ANALYSIS AND IMPLEMENTATION OF DECIMAL ARITHMETIC HARDWARE IN NANOMETER CMOS TECHNOLOGY

By

IVAN DARIO CASTELLANOS

Bachelor of Science in Electrical Engineering
Universidad de los Andes
Bogotá, Colombia
2001

Master of Science in Electrical Engineering
Illinois Institute of Technology
Chicago, Illinois
2004

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 2008

ANALYSIS AND IMPLEMENTATION OF DECIMAL ARITHMETIC HARDWARE IN NANOMETER CMOS TECHNOLOGY

Dissertation Approved:

Dr. James E. Stine

Dissertation Adviser

Dr. Louis Johnson

Dr. Sohum Sohoni

Dr. Nohpill Park

Dr. A. Gordon Emslie

Dean of the Graduate College

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

1. INTRODUCTION

Many scientific, engineering and commercial applications call for operations with real numbers. In many cases, a fixed-point numerical representation can be used. Nevertheless, this approach is not always feasible since the range that may be required is not always attainable with this method. Instead, floating-point numbers have proven to be an effective approach as they have the advantage of a dynamic range, but are more difficult to implement, less precise for the same number of digits, and include round-off errors.

The floating-point numerical representation is similar to scientific notation differing in that the radix point location is fixed usually to the right of the leftmost (most significant) digit. The location of the represented number's radix point, however, is indicated by an exponent field. Since it can be assigned to be anywhere within the given number of bits, numbers with a "floating" radix point have a wide dynamic range of magnitudes that can be handled while maintaining a suitable precision.

The IEEE standardized the floating-point numerical representation for computers in 1985 with the IEEE-754 standard [1]. This specific encoding of the bits is provided and the behavior of arithmetic operations is precisely defined. This IEEE format minimizes calculation anomalies, while permitting different implementation possibilities. Since the 1950's binary arithmetic has become predominantly used in computer operations given its simplicity for implementation in electronic circuits. Consequently, the heavy utilization of binary floating-point numbers mandates the IEEE binary floating-point standard to be

required for all existing computer architectures, since it simplifies the implementation. More importantly, it allows architectures to efficiently communicate with one another, since numbers adhere to the same IEEE standard.

Although binary encoding in computer systems is prevalent, decimal arithmetic is becoming increasingly important and indispensable as binary arithmetic can not always satisfy the necessities of many current applications in terms of robustness and precision. Unfortunately, many architectures still resort to software routines to emulate operations on decimal numbers or, worse yet, rely on binary arithmetic and then convert to the necessary precision. When this happens, many software routines and binary approximations could potentially leave off crucial bits to represent the value necessary and potentially cause severe harm to many applications.

1.1 Importance of Decimal Arithmetic

Decimal operations are essential in financial, commercial and many different Internet based applications. Decimal numbers are common in everyday life and are essential when data calculation results must match operations that would otherwise be performed by hand [2]. Some conventions even require an explicit decimal approximation. A study presented in [3] shows that numeric data in commercial applications, like banking, insurance and airlines is predominantly decimal well up to 98%. Furthermore, another study discussed in [4] shows that decimal calculations can incur a 50% to 90% processing overhead.

One of the main causes for decimal's performance cost is that binary numbers cannot represent most decimal numbers exactly. A number like 0.1, for example, would require an infinite recurring binary number, whereas, it can be accurately represented with a

decimal representation. This implies that it is not always possible to guarantee the same results between binary floating point and decimal arithmetic. This is further illustrated in the following table where the number 0.9 is continuously divided by 10.

Table 1. Numerical differences between decimal and binary floating-point numbers.

Decimal	Binary
0.9	0.9
0.09	0.089999996
0.009	0.009
0.0009	9.0E-4
0.00009	9.0E-5
0.000009	9.0E-6
9E-7	9.0000003E-7
9E-8	9.0E-8
9E-9	9.0E-9
9E-10	8.9999996E-10

It is, therefore, considerably difficult to develop and test applications that require this type of calculations and that use exact real-world data like commercial or financial values. Even legal requirements, like the Euro (€) currency regulations, dictate the working precision and rounding method to be used for calculations in decimal digits [5]**Error! Reference source not found.**. These requirements can only be met by working in base 10, using an arithmetic which preserves precision.

Typically, decimal computations are performed on binary hardware through software emulation and mathematical approximations, since requirements specific to decimal numbers cannot always be met in pure binary form. These requirements may include arithmetic that preserves the number of decimal places (including trailing zeroes or unnormalized coefficients) and decimal rounding among others. In all cases, any scaling, rounding, or exponent has to be handled explicitly by the applications or the programmer, a complex and very error-prone task. Since binary computations for decimal arithmetic tend to be slow, significant performance improvements may result

from using decimal floating-point hardware. Native (hardware) decimal floating-point arithmetic will make programming far simpler and more robust, and produce a significantly better performance in computer applications. The impact of this type of hardware can improve decimal floating-point calculations speed by **two or three orders** of magnitude compared to a software approach and is further highlighted with IBM's release of the Power6 processor, the first UNIX microprocessor able to calculate decimal floating-point arithmetic in hardware [7].

As an example, shown in [4], division of a JIT (Java Just-In-Time compiled) 9-digit BigDecimal number type, takes more than *13,000* clock cycles on an Intel® Pentium™ processor, while a *9*-digit decimal addition requires more than *1,100* clock cycles. On the other hand, binary arithmetic takes *41* cycles for integer division and *3* cycles for an addition on the same processor. Dedicated decimal hardware would be comparable to these values, if available.

1.2 The Decimal Floating-Point Standard

The increasing importance of decimal arithmetic is highlighted by the specifications being included in the current revision draft of the IEEE-754 standard for floating-point arithmetic or IEEE-754R [8]. Decimal floating-point numbers are in a format similar to scientific notation:

$$(-1)^{S} \times \textit{Coefficient} \times 10^{(\textit{Exponent} - \textit{Bias})},$$

where *S* is either 1 or 0 and determines the sign of the number. The exponent is biased to avoid negative representations. In other words, all exponents are represented in relation to a known value given to exponent zero. For example, if the bias is 127 numbers below 127 are negative and above 127 are positive. To illustrate a specific

example, suppose an exponent of 125 is utilized with a bias of 127, this exponent represents a value of -2 according to the IEEE standard.

The current draft specifies the representation of three decimal number types: decimal32, decimal64 and decimal128 encoded in *32*, *64* and *128*-bits respectively. The value of the number is encoded in four different fields. An illustration of this representation for decimal64 numbers is shown in Table 2, taken from [9].

Table 2. Decimal floating-point format

Length (bits)	1	5	8	50
Description	Sign	Combination Field	Exponent Continuation	Coefficient Continuation

Decimal64 numbers are comprised of a 16 digit coefficient and a 10-bit biased exponent. The sign bit indicates the sign of the number as indicated earlier, in the same way as binary floating-point numbers. Both exponent and coefficient are encoded with part of their value given in the combination field: the two Most Significant Bits (MSBs) for the exponent and the Most Significant Digit (MSD) of the coefficient. The combination field also determines if the number represented is a finite number, an infinite number or a *NaN* (Not-a-Number). Quiet and signaling *NaN*s (in which case an exception is triggered or signaled) are determined by the first bit of the exponent continuation field. Table 3, shown in [9], illustrates the combination field which depends if the number is Infinity, a *NaN* or a finite number. The combination field is encoded differently as well if the finite number's MSD is greater than or equal to 8 or if the number is less as illustrated in the first two entries of the table. The exponent is therefore a biased unsigned 10-bit binary number and the coefficient is given by a specific 10-bit per 3 decimal-digits encoding representing a 16 decimal digits number. An additional important characteristic of

decimal floating-point numbers is that they are not normalized, as opposed to binary floating-point numbers.

Table 3. Decimal FP Combination Field

Combination Field (5 Bits)	Type	Exponent MSBs (2-bits)	Coefficient MSD (4-bits)
a b c d e	Finite < 8	a b	0 c d e
1 1 c d e	Finite > 7	c d	1 0 0 e
1 1 1 1 0	Infinity	- -	- - - -
1 1 1 1 1	NaN	- -	- - - -

For example, suppose a programmer wants to encode the number -8.35 into decimal64. The first step is to break the number into its coefficient and exponent, which produces 835 (with 13 leading zero decimal digits given that coefficients are 16 digits long) and –2 respectively, i.e. -835×10^{-2}. For decimal64 numbers, where the bias value is of 398_{10}, an exponent of –2 becomes 396_{10} ($01\ 1000\ 1100_2$). The combination field for -835×10^{-2} contains the two most significant bits or MSBs of the exponent (01 in this example) and the most significant digit (MSD) of the coefficient (4-bits, 0000 in this case since the MSD is zero). According to Table 3, for finite numbers with the most significant digit value below 8, the 5-bit combination field *abcde* decodes *ab* as the Exponent's MSBs and 0*cde* as the MSD. To illustrate an example, the number –8.35 becomes 01 | 000. The remaining 8-bits of the exponent, 0x8C, are arranged in the exponent continuation field. Finally, the coefficient is given in the coefficient continuation field using Densely Packed Decimal (DPD) encoding [10]. DPD encoding provides an efficient method of storing and translating 10-bit / 3 decimal digits into BCD representation and vice versa by using simple Boolean expressions. A more detailed explanation of DPD can be found in the Appendix.

The DPD codification of the three BCD decimal digits into 10-bits is called compression and it depends on the size of each digit, *small* or *large* (3-bit for less than or equal to 7, and 4-bits for greater than 7). A specific mapping is used in each situation: when all digits are small, left digit is small, middle digit is large, etc [10]. The three digits 835 are given in BCD as bits *abcd efgh ijkm* $(1000\ 0011\ 0101)_2$. Bits *a*, *e* and *i* are used to indicate if the numbers are *large* or *small*. For this specific case, in which left digit is a *large* number, the mapping used for the encoding has the form [*jkd fgh* 1 10 *m*] or 0x23D (see second table in Appendix A). Therefore, the decimal64 representation for –8.35 is, in hexadecimal, A2 30 00 00 00 00 02 3D.

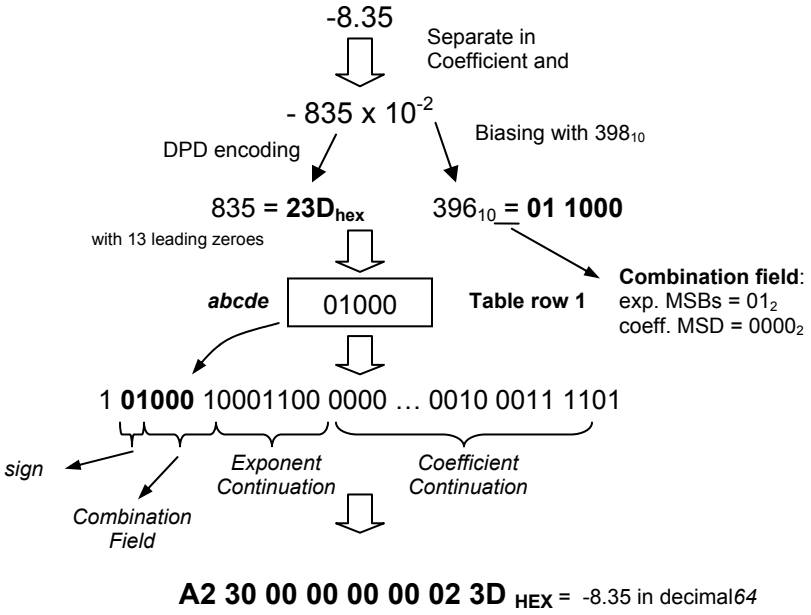


Figure 1. Example: decimal floating-point representation of number -8.35.

1.3 A case for Decimal Arithmetic in General-Purpose Computer Architectures

Decimal arithmetic has long been studied in computer architectures, however, most silicon implementations of digital logic suffered due to area requirements. By the year

2010, processors with 2 billion transistors are expected to be developed [11]. Therefore, the large number of transistors available within silicon implementations and the increased sophistication of design tools gives designers the ability to include new and important features, such as decimal arithmetic. Previous implementations in decimal arithmetic include high-speed multipliers [12][13][14], algorithms for decimal adders [15][16][17] and multi-operand addition [18][19], and algorithms for decimal partial product generation [14][19][20]. Although there has been a large amount of interest and research interest into decimal arithmetic architectures, many of the architectures fail to produce designs that are targeted at real implementations, especially at designs below 180nm. This dissertation attempts to study these designs by offering possible solutions and implementations in decimal arithmetic and, in some cases, how they possibly can be combined with binary arithmetic to produce combined binary/decimal arithmetic units.

2. BACKGROUND

Decimal arithmetic operations were significantly researched in the 1950's and the latter part of the 20th century, but nonetheless binary arithmetic hardware took over computer calculations. The reasoning behind this came after Burks, Goldstine, and von Neumann published a preliminary study on computer design [12]. They argued that for scientific research, simplicity was the major advantage of binary hardware and therefore increasing its performance and reliability. Furthermore, decimal numbers would need to be stored in binary form requiring extra storage space (bits) to maintain the same precision as binaries and require more circuitry than operations performed in pure binary form. Nevertheless if conversions from decimal to binary and vice-versa are needed then it is significantly more efficient to perform operations in decimal hardware [4].

In many cases, techniques developed for binary arithmetic hardware can be applied to some extent to decimal hardware. It is therefore important to explore relevant approaches and research for binary arithmetic since an invaluable insight into solving decimal arithmetic problems can be gained.

2.1 Binary Comparison

An important element in general purpose and application specific architectures is the comparator [22]. The design of high speed and efficient comparators aids in the performance of these architectures. The idea of designing efficient comparators however is not new as seen from previous studies in [23], [23], [23], [25]. Nevertheless further

gains in area usage and power can be obtained by designing a comparator that can handle different data-types using an efficient compatible comparison method.

The work on [26] presents the design and implementation of a high performance comparator capable of handling 32-bit and 64-bit two's complement numbers and single and double precision binary floating-point numbers. This type of design is especially useful to reduce costs in processors, since it allows the same hardware to be used to compare multiple data types. A novel approach to the magnitude comparison problem was utilized with a comparator module that has logarithmic delay. This design can also be easily extended to support 128-bit binary floating point numbers and can accommodate pipelining to improve throughput.

The IEEE 754 standard [1] specifies floating-point comparisons where the relation between two numbers is given greater than, less than, equal or unordered. When either of the operands compared are Not-a-Number or *NaN* the result of the comparison is unordered. If the *NaN* is a signaling *NaN* then an Invalid exception flag bit is asserted. The result of the comparison is represented by Floating-point Condition Codes or *FCC* [27].

Table 4 shows the *FCC* representation of the comparison result in this design. Note that bit *FCC*[1] is analogous to a greater than flag (*GT*) and *FCC*[0] is analogous to a less than flag (*LT*). When both flags are zero the numbers are equal and when both are one the numbers are unordered.

Table 4. Floating-point Condition Codes.

FCC [1] (GT)	FCC [0] (LT)	Relation
0	0	A = B
0	1	A < B
1	0	A > B
1	1	Unordered

As illustrated in [26], the combined comparator is composed of three blocks. The 2-bit *Sel* signal indicates the type of operands being compared. The first block converts 32-bit operands to 64-bit operands so that all operand sizes are handled by the same hardware. Like most floating-point implementations, 32-bit numbers are converted to 64-bit numbers to simplify the logic. The second block performs a magnitude comparison. Finally, the third block takes care of exceptions and special cases according to the IEEE 754 standard. It also correctly handles the signs of the input operands.

2.1.1 Magnitude Comparator Design

The magnitude comparator devised in [26], and shown in Figure 2, is the core of the comparator module. The two operands *A* and *B* are compared in stages. The first stage compares corresponding 2-bit pairs from each operand. Two output bits, *GT* (greater than) and *LT* (less than), from each element indicate if the result of the compared pair is greater than, less than or equal as shown in Table 5. If the bit pairs of *A* and *B* are denoted by $A[2i+1, 2i]$ and $B[2i+1, 2i]$ then the values for $GT[i]_1$ and $LT[i]_1$ (where the subscript 1 indicates the first stage of the comparison) are given by:

$$GT[i]_1 = A[2i+1] \cdot \overline{B[2i+1]} + A[2i+1] \cdot A[2i] \cdot \overline{B[2i]} + A[2i] \cdot \overline{B[2i+1]} \cdot \overline{B[2i]},$$

$$LT[i]_1 = \overline{A[2i+1]} \cdot B[2i+1] + \overline{A[2i+1]} \cdot \overline{A[2i]} \cdot B[2i] + \overline{A[2i]} \cdot B[2i+1] \cdot B[2i].$$

for ($0 \le i \le \lceil n/2 \rceil - 1$) where *n* is the operand size, in this case 64.

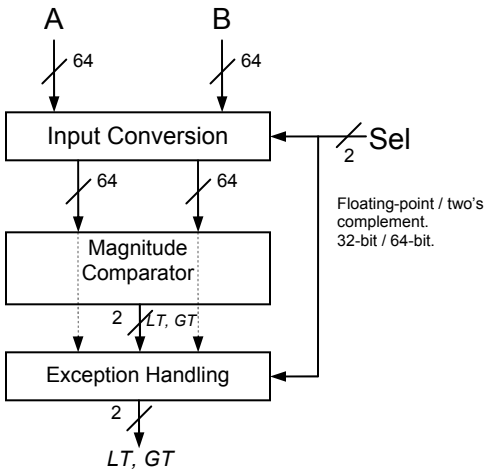


Figure 2. Block diagram of the binary portion of the comparator, taken from [26].

In subsequent stages the same process is used except the $GT[i]_j$ signals replace the $A[i]$ signals and $LT[i]_j$ replace $B[i]$ where *j* denotes the comparator stage. There is however an additional reduction possible in subsequent stages since *GT* and *LT* can not be equal to 1 at the same time and therefore for *j* > 1 the equations are simplified to:

$$GT[i]_{j+1} = GT[2i+1]_j + GT[2i]_j \cdot \overline{LT[2i+1]_j},$$

$$LT[i]_{j+1} = LT[2i+1]_j + \overline{GT[2i+1]}_j \cdot \overline{LT[2i]}_j.$$

A total of $k = \lceil \log_2(n) \rceil$ stages are required to obtain the final result given by $GT[0]_k$ and $LT[0]_k$. In the case of this implementation with 64-bits, *n* = 64 and *k* = 6 stages are required.

Table 5. BCD Magnitude Comparison

GT[i]	*LT[i]*	Result
0	0	*A[2i+1,2i] = B[2i+1,2i]*
0	1	*A[2i+1,2i] < B[2i+1,2i]*
1	0	*A[2i+1,2i] > B[2i+1,2i]*
1	1	invalid

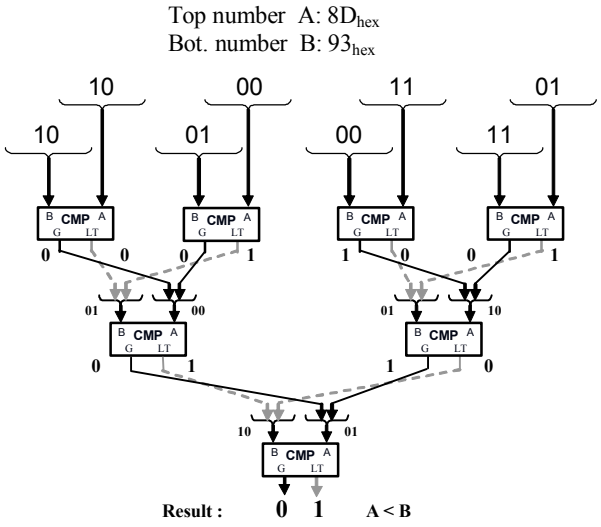


Figure 3. Magnitude comparator example, A = 0x8D and B = 0x93. *n*= 8, *k* = 3 stages necessary.

Figure 3 illustrates an example using this logarithmic tree magnitude comparator. The operand size in this case is 8-bits and therefore only 3 stages are necessary (*k*=3). The comparison to be computed is *A* to *B* where *A* = 0x8D and *B* = 0x93. Each number is separated in bit pairs and each corresponding pair is compared individually. Subsequent stages group *LT* and *GT* signals together as shown. The final stage yields *GT* = 0 and *LT* = 1 as expected giving *A* < *B*.

2.1.2 Two's complement and binary floating-point comparator

The magnitude of the operands however is not the only characteristic considered when comparing numbers. To compare two's complement or floating-point numbers the sign should also be considered. The third stage shown in Figure 2 sets the *LT* output to one in any of the following four cases:

1) *A* is negative and *B* is positive.
2) *A* and *B* are positive and the magnitude of *A* is less than the magnitude of *B*.
3) *A* and *B* are negative two's complement numbers and the magnitude of *A* is less than the magnitude of *B*.
4) *A* and *B* are negative floating point numbers and the magnitude of *A* is greater than the magnitude of *B*.

To minimize the complexity of the other predicates like Greater Than (*GT*) and Equal to (*EQ*), logic is asserted based on whether the input operands are *LT* or *not LT* given that *LT*, *GT*, and *EQ* cannot all be asserted simultaneously. This translates to simple logic for both 32 and 64-bit numbers. In order to make sure the values for the cases listed above are produced correctly for the implementation presented in this paper, only $LT[0]_6$ and $GT[0]_6$ are computed utilizing the logic since $EQ[0]_6$ can be produced by the following equation:

$$EQ[0]_6 = \overline{LT[0]_6 + GT[0]_6}$$

The subindex $_6$ denotes the sixth level of the magnitude comparator, or the final stage given that the operands considered are 64-bits, i.e. $LT[0]_6$ and $GT[0]_6$ are the outputs of the magnitude comparator module.

Consequently, the values of *LT*, *EQ*, or *GT* for the whole design can be produced for two's complement numbers as:

$$EQ = EQ[0]_6 \cdot (A[63] \cdot B[63] + \overline{A[63]} \cdot \overline{B[63]})$$

$$LT = (A[63] \cdot \overline{B[63]} + \overline{B[63]} \cdot LT[0]_6 + A[63] \cdot LT[0]_6) \cdot \overline{EQ}$$

$$GT = \overline{LT + EQ}$$

Floating-point comparisons on the other hand are complicated because of the incorporation of exceptions which are mandated by the IEEE 754 standard. The major exception that should be detected with comparisons is if the operands are *Unordered*. According to the IEEE 754 standard, values are unordered if either operand is a *NaN* and a floating-point comparison is being performed. The hardware for detecting unordered may vary from one processor to the next, because the standard allows discretion in defining specific *signaling* and *quiet NaN*'s bit patterns. The IEEE 754 standard also states that comparisons must also output an *Invalid Operation* exception if either operand is a *signaling NaN* [1]. Furthermore, a final test must be performed to make sure +0 and −0 compare *Equal*, regardless of the sign.

In summary, the floating-point comparisons must be able to handle *Invalid* operations, both types of *NaN*'s, and not differentiate between both types of zeroes. As with two's complement numbers, the comparator is simplified by computing whether the two operands are *Equal* or *Less* than each other. Once these two outputs are known, it is simple to produce *Greater than* output. For the combined unit, the *Less than* comparison utilizes the same cases tabulated previously accounting for a floating-point operation. On the other hand, floating-point comparisons for *Equal* need to be modified to account for

either equal operands or the comparison of zeroes. Therefore, the combined comparator (two's complement and floating-point) handles two cases for determining whether the operands are *Equal*:

1) The operand magnitudes are equal AND the operands' signs are equal.
2) The operand magnitudes are zero AND the operands are floating point numbers.

The final equations for the combined comparator are given below, where *Azero* represents a literal testing of whether *A* is +0 or −0, *fp* represents a literal specifying a floating-point operation and *UO* represents a literal indicating unordered operands:

$$UO = (A_{NaN} + B_{NaN}) \cdot fp$$

$$EQ = EQ[0]_6 \cdot (A[63] \cdot B[63] + \overline{A[63]} \cdot \overline{B[63]} + Azero \cdot fp) \cdot \overline{UO}$$

$$LT = (A[63] \cdot \overline{B[63]} + \overline{A[63]} \cdot \overline{B[63]} \cdot LT[0]_6 + A[63] \cdot B[63] \cdot LT[0]_6 \cdot \overline{fp} + A[63] \cdot B[63] \cdot \overline{LT[0]_6} \cdot fp) \cdot \overline{EQ} \cdot \overline{UO}$$

$$GT = \overline{LT + EQ + UO}$$

For these equations, logic is saved by only testing whether *A* is zero since $EQ[0]_6$ already indicates if the operands are equal making a test of *B* equal to zero redundant.

Sign extension for 32-bit two's complement numbers is implemented by sign extending the 32nd bit into the upper 32-bits of the comparator. IEEE single-precision numbers do not need to be converted to double-precision numbers, since the two formats have the same basic structure and the exponents are biased integers. The logic to detect *NaN*s and *zeros* for the two floating-point formats differs slightly, since single precision numbers have smaller significands and exponents than double precision numbers.

2.2 Addition

Addition is a fundamental arithmetic operation and the design of efficient adders aids as well in the performance and efficiency of other operation units like multipliers and dividers. Decimal addition has been researched but not as heavily as binary addition and only a handful of research papers can be found on the topic. Nevertheless, binary arithmetic is important for the decimal case since decimal numbers are represented in binary and many concepts and techniques developed for binary can be applied to some extent as well. An overview of some relevant binary addition concepts is therefore necessary.

2.2.1 Binary addition

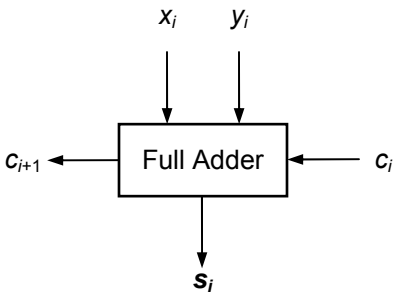
One of the most basic elements in addition is the Full Adder (*FA*) or 3:2 counter. Adders are sometimes called counters, because they technically count the number of inputs that are presented at their input [28]. The *FA* takes three single bit inputs, x_i, y_i and c_i and produces two single bit outputs s_i and c_{i+1} corresponding to [29]:

$$x_i + y_i + c_i = 2 \cdot c_{i+1} + s_i \,,$$

where c_i is commonly referred to as the carry-in and c_{i+1} the carry-out. The logic equations for the Full Adder cell are given by:

$$s_i = x_i \oplus y_i \oplus c_i$$

and $c_i = x_i y_i + x_i c_i + y_i c_i$.



x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 4. Full adder cell and Truth Table.

The full adder cell can be utilized to create *n*-bit operand adders as shown in the next figure. This simple approach, called Ripple Carry Adder or Carry Propagate addition (*RCA*/*CPA*), has the disadvantage of a significant time-consuming delay due to the long carry chain as the carry propagates from c_0 to c_1 all the way until the MSB, in this case s_3.

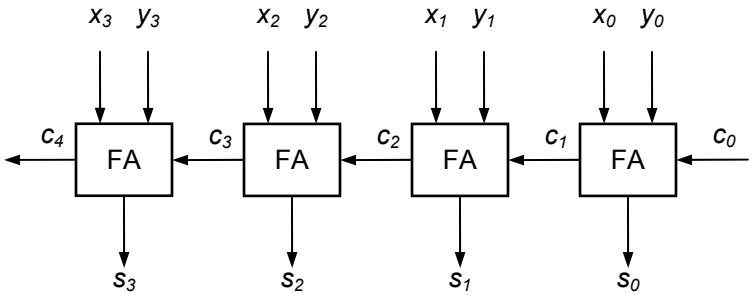


Figure 5. 4-bit Ripple Carry Adder.

In order to speed up this process, certain aspects of the addition in each cell can be exploited, as is the case for the Carry Look-ahead Adder (*CLA*). If a carry is present at the FA's carry-in from the previous significant bit it is said to *propagate* if either x_i or y_i

are equal to 1. On the other hand if a carry is generated within the FA cell, when both x_i and y_i are 1, the cell is said to *generate* a carry-out. Logic equations for *generate* and *propagate* signals (*g* and *p*) as well as an equation describing when a carry-out takes place can therefore be determined from the inputs:

$$g = x_i \cdot y_i \,,$$

$$p = x_i + y_i \,,$$

$$c_{i+1} = g_i + c_i \cdot p_i$$

The last equation can be utilized to determine the carry-out of the next significant bit *FA*:

$$c_{i+2} = g_{i+1} + c_{i+1} \cdot p_{i+1} = \boldsymbol{g_{i+1} + (g_i + c_i \cdot p_i) \cdot p_{i+1}} \,.$$

Showing that c_{i+2} can be obtained exclusively with the operands inputs without the need of the carry c_{i+1} as in Figure 5. This provides a method of obtaining the carry-out result for each bit position without the need of a carry chain and hence speeding up significantly the process.

As the operand size is increased, however, the complexity of each new bit's carry-out logic grows significantly making the method impractical for operands of more than 4-bits, depending on the technology used. In this case, further techniques allow carry generate and propagate signals to be obtained for an n-bit block and improve the adder's implementation.

2.2.2 Carry save addition (CSA)

Carry-save addition is the idea of utilizing addition without carries connected in series as in the Ripple Carry Adder but instead to count and hence avoid the ripple carry chain. In

this way multi-operand additions can be carried out without the excessive delay resulting from long carry chains. The following example shows how a 4-bit *CSA* accepts three 4-bit numbers and generates a 4-bit partial sum and 4-bit carry vector, avoiding the connection of each bit adder's carry-out to the carry-in of the next adder.

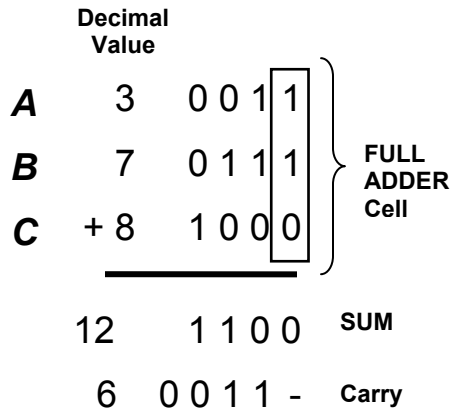


Figure 6. Full adder cells used for carry-save addition.

The example shown demonstrates how performing addition in a given array (each column in the figure) produces an output with a smaller number of bits; in this case form 3 bits to 2. This process is called *reduction* and is very useful during multiplication.

2.2.3 4:2 Compressors

One particular useful carry-save adder is the 4:2 compressor presented in [30]. The main reason for using compressors is that their carry-out (c_{out}) is no longer dependent on the c_{in}, as shown in Figure 7. This gives compressors a significant advantage over traditional carry-save adder trees implemented with 3:2 counters in that it can expedite processing the carry chain while still maintaining a regular structure.

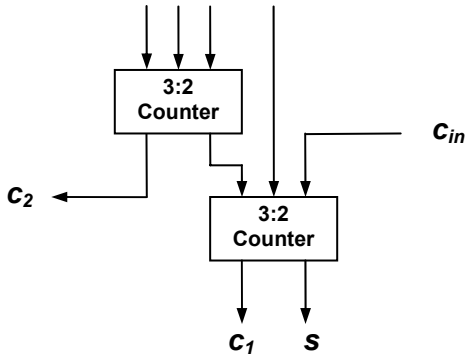


Figure 7. Weinberger 4:2 Binary Compressor.

2.2.4 Decimal excess-3 addition

As stated earlier, usually decimal numbers are stored as Binary Coded Decimals (BCD). BCD numbers have 6 unused combinations, from 1010_2 to 1111_2, and this complicates addition and subtraction for the decimal case. Furthermore, negative numbers can not be represented in two's complement fashion which is a common method for subtraction for the binary case.

A different coding for decimal numbers, called Excess-3, is important since it has many useful properties for subtraction and addition. Excess-3 code can be generated by just adding a binary 3 to the common BCD code, as shown in Table 6.

Table 6. Excess-3 Code.

Decimal Value	Excess-3 Code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

Except for some corrections necessary during addition/subtraction, common binary techniques can be applied for arithmetic operations. Most importantly the addition of two numbers creates a decimal carry which is available by using the carry output of the most significant binary bit. This occurs because the addition of two excess-3 digits creates a result in excess-6 which already eliminates the unwanted 6 binary combinations. Furthermore, the code is self-complementing [31]. This implies that a subtraction or negative number addition can be obtained by inverting all bits of the digit and adding a binary ulp, in the same way as two's complement binary numbers.

The following equation, taken from [31], shows the operation result of two Excess-*3* numbers added together, where the underline represents a digit in BCD:

$$SUM = D_1 + D_2 = \underline{D_1} + 3 + \underline{D_2} + 3 = \underline{D_1} + \underline{D_2} + 6$$

There are two possibilities to consider for the sum result. When $\underline{D_1} + \underline{D_2} < 10$ then no carry to the next higher digit is needed and the Excess-6 result can be corrected by just subtracting 3 from the sum. This can be easily accomplished by adding 13 and ignoring the carry output, which effectively subtracts 16. When $\underline{D_1} + \underline{D_2} \geq 10$ a decimal carry should be signaled to the digit in the next place. This can be accomplished by sending the Carry out signal of the most significant bit. Nevertheless this sends a carry of 16 (6 too much) and hence by adding 3 the result is restored into Excess-3 code. Note that in both cases the correction requires the addition of *3* or *13* which can be accomplished by a simple inverter on the LSB output. Figure 8 shows an implementation of an Excess-*3* adder.

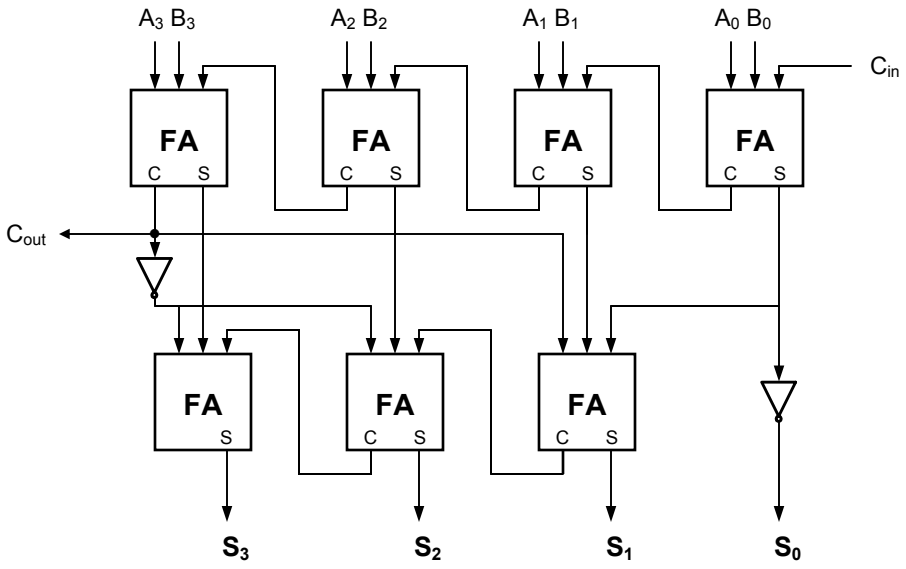


Figure 8. Adder for Excess-3 code, taken from [31].

2.2.5 Direct decimal addition

The use of Excess-3 code permits the addition of two decimal numbers by using a correction method to that corrects the six unwanted values in BCD code after the operation takes place (1010_2 to 1111_2). Regardless, a different approach proposed in [15] presents logic that performs *direct decimal addition* where a combinational element has as inputs two 4-bit BCD numbers x_i and y_i and a carry-in $c_i[0]$ and outputs a 4-bit BCD digit s_i and a 1-bit carry-out $c_{i+1}[0]$ satisfying:

$$(c_{i+1}, s_i) = x_i + y_i + c_i[0] ,$$

where c_{i+1} represents ten times the weight of s_i . The following are the logic equations that describe the direct decimal adder [12]:

$$g_i[j] = x_i[j] \cdot y_i[j] \qquad 0 \le j \le 3 \text{ “}\textit{generate}\text{”}$$

$$p_i[j] = x_i[j] + y_i[j] \quad 0 \le j \le 3 \text{ “}propagate\text{”}$$

$$h_i[j] = x_i[j] \oplus y_i[j] \quad 0 \le j \le 3 \text{ “}addition\text{”}$$

$$k_i = g_i[3] + (p_i[3] \cdot p_i[2]) + (p_i[3] \cdot p_i[1]) + (g_i[2] \cdot p_i[1])$$
$$l_i = p_i[3] + g_i[2] + (p_i[2] \cdot g_i[1])$$
$$c_i[1] = g_i[0] + (p_i[0] \cdot c_i[0])$$

$$s_i[0] = h_i[0] \oplus c_i[0]$$
$$s_i[1] = ((h_i[1] \oplus k_i) \cdot \overline{c_i[1]}) + ((\overline{h_i[1] \oplus l_i}) \cdot c_i[1])$$

$$s_i[2] = (\overline{p_i[2]} \cdot g_i[1]) + (\overline{p_i[3]} \cdot h_i[2] \cdot \overline{p_i[1]}) + ((g_i[3] + (h_i[2] \cdot h_i[1])) \cdot \overline{c_i[1]})$$

$$(((\overline{p_i[3]} \cdot \overline{p_i[2]} \cdot p_i[1]) + (g_i[2] \cdot g_i[1]) + (p_i[3] \cdot p_i[2])) \cdot c_i[1])$$

$$s_i[3] = ((\overline{k_i} \cdot l_i) \cdot \overline{c_i[1]}) + (((g_i[3] \cdot \overline{h_i[3]}) + (\overline{h_i[3]} \cdot h_i[2] \cdot h_i[1])) \cdot c_i[1])$$
$$c_{i+1}[0] = k_i + (l_i \cdot c_i[1])$$

These equations describe a decimal full adder that can be utilized for either carry-save or carry propagate addition.

2.2.6 Decimal Floating-Point Adder

To the author’s knowledge the only published work to date of an arithmetic module compliant with the IEEE-754 current revision draft is the decimal floating-point adder published by Thompson, Karra and Schulte in [16] and hence its inclusion in this section is of significance. This design differs from previous decimal adders in that it is fully compliant with the standard including special value cases and exception handling, and

that it is capable of generating a complete result in a single cycle instead of a single digit per cycle.

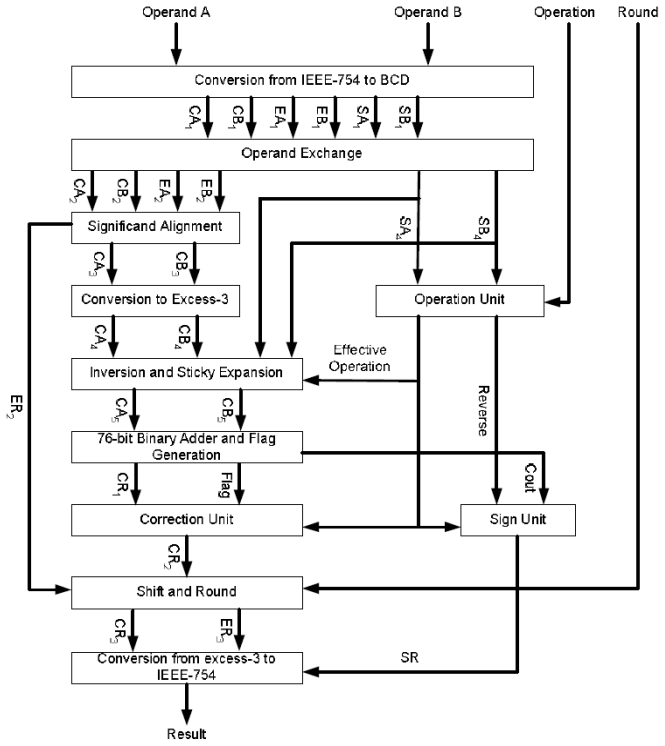


Figure 9. Decimal Floating-Point adder, from [16].

Figure 9 shows a block diagram of the adder design. Initially the two IEEE-754 decimal numbers are decoded into their sign bits, coefficient (BCD) and Exponent fields (two's complement binary). The operand exchange block orders the coefficients according to which number's exponent is greater followed by the operation unit which determines the actual operation to be performed (addition or subtraction) depending on the signs of the operands. The coefficients, or significands, are aligned and a conversion into Excess-3 format follows for their respective binary addition and flag bits determination. The result is finally corrected, depending on the previously set flags, shifted and rounded allowing it to be encoded back into IEEE-754 decimal format. The adder presented in this work also

allows up to 5 stages of pipelining which improves its critical path delay. Also it is of importance since it explores the advantage of using Excess-3 coding for decimal addition.

2.3 Binary Multiplication

Since decimal operations are performed on binary circuits, understanding how binary multiplication is achieved aids in the application and development of new techniques for the decimal case. A brief overview on its most significant implementations is given in this Section for that purpose.

The multiplication of two binary numbers, multiplier X (x_{N-1}, x_{N-2}, …, x_1, x_0) and multiplicand Y (y_{M-1}, y_{M-2}, …, y_1, y_0) is determined by the following equation [32]:

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j} \ .$$

This is illustrated in Figure 10 for the case of 6-bit operands:

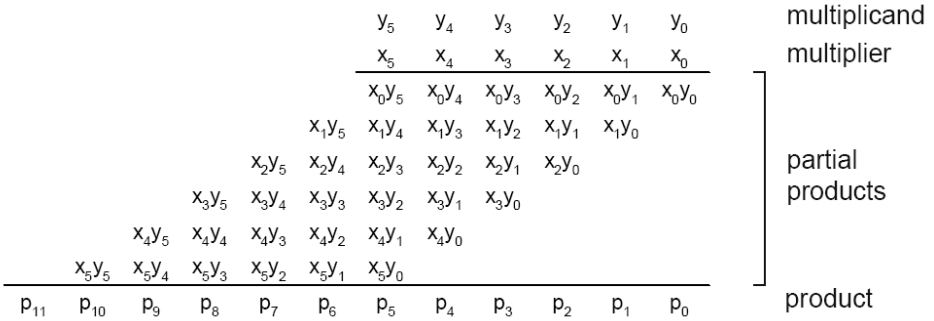


Figure 10. Partial products during multiplication, taken from [32].

Each partial product bit position can be generated by a simple AND gate between corresponding positions of the multiplier and multiplicand bits as shown in figure 3. All partial products are reduced or “compressed” by addition into a single product result.

A direct implementation of this method is given in the Carry Save Array Multiplier or CSAM. In this type of multiplier the partial product array shown in Figure 10 is skewed into a square shape so that its implementation is more efficient for VLSI. The compression is performed by the use of full adders (*FA*s / 3:2 counters) and half adders (*HA*s). Figure 11 shows this array for the multiplication of two 8-bit operands. *MFA* and *MHA* cells represent full adders and half adders with an additional *AND* gate input to generate the partial product. The highlighted arrow shows the critical path of the circuit, the longest carry propagation chain. In Figure 10, which has 6-bit operands instead of 8, this would correspond to the addition of the 6^{th} column (partial products x_0y_5 to x_5y_0) plus the carry propagation through the last adder that generates the product, from p_5 to p_{11}. This long carry chain limits significantly the performance of the multiplier and is even more considerable as the operand size is incremented.

One of the most significant works that addressed this problem was proposed by Wallace [33]. Wallace suggested the use full adders and half adders in a recursive fashion adding three elements at a time in a carry propagate free way. In this manner, the partial product array can be reduced in stages subsequently to two numbers without carry propagation. When the resulting two numbers are obtained, a Carry Propagate Addition (*CPA*) takes place to obtain the final result [34].

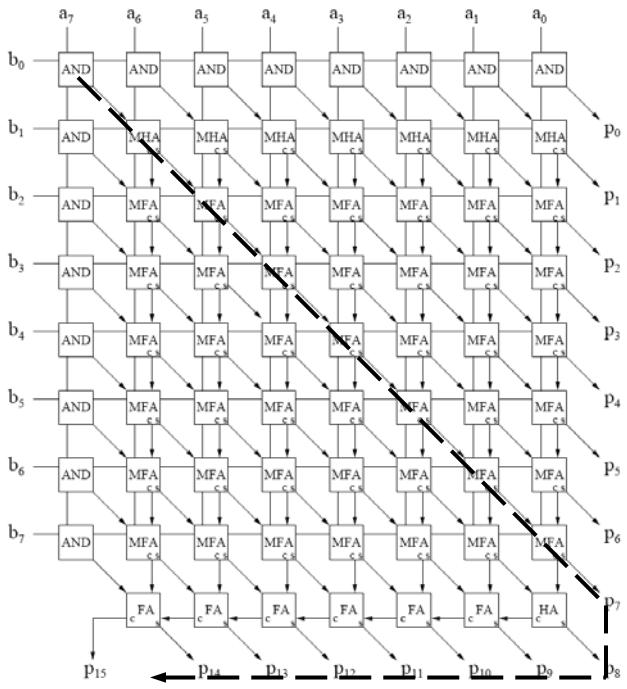


Figure 11. Carry Save Array Multiplier, CSAM.

This is shown in the example in Figure 12. The diagram illustrates a 4-bit multiplication where the resulting partial products are shown at the top, analogous to the 6-bit multiplication of Figure 10. Each dot represents a partial product bit position (e.g. x_0y_5, x_3y_2, etc.) The second step shows the partial product array reorganized in a triangular shape where the oval around the dots represents a full adder (3 inputs) or a half adder (2 inputs). The result of each addition produces two elements, a sum and a carry-out to its next significant bit position (column to its left). The process is repeated again until at the final stage only two bit array numbers are left, and a reduced size carry propagate addition is required to produce the final result.

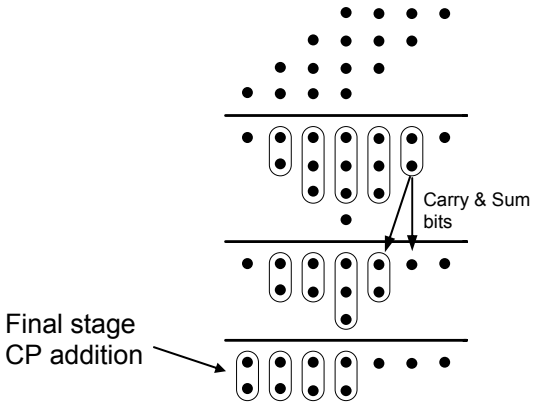


Figure 12. Wallace Tree multiplier reduction for two 4-bit operands. Carry and sum bits for the Half Adder shown.

Figure 13 illustrates how a Wallace tree for 6-bit operands is implemented using *FA*s and *HA*s. In this case the partial products corresponding to the 5th column are detailed. A partial product column array of 5-bits feeds a *FA* and a *HA*. The carry-out bits produced are the inputs for the 2nd stage *FA* on the next column. The output sum bits are passed directly to the next stage *FA*, within the same column. In this manner a partial product reduction tree can be formed.

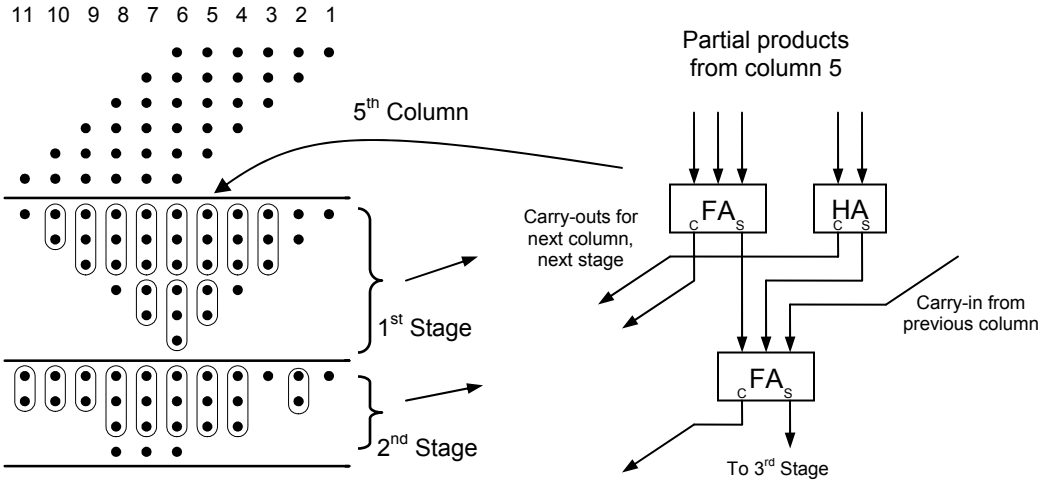


Figure 13. Implementation detail for Wallace tree, 5th column (only 2 stages are shown).

As can be seen from the example shown in Figure 13, the resulting Wallace reduction tree is not regular and hence causes difficulties when the circuit layout is implemented. Nevertheless, the use of 4:2 compressors (exposed in Section 2.2.3), can be organized into efficient interconnection networks for reducing the partial product matrix in a matter that is regular and more suitable for implementation. However, careful attention has to be placed when organizing these compressor trees, because the carry terms within the 4:2 compressor have a weight that is one more than its sum, corresponding to the next significant bit (column to its left). This means that compressor trees must be built according to the following:

- The column sum output sum for any compressor tree utilizes the current weight of its column.

- The column carry output for any compressor tree must utilize the previous weight of the current column.

Therefore, although compressor trees are traditionally drawn as binary trees, they must be organized carefully so that the counter outputs are summed together properly. Figure 14 shows an example of an 8-bit compressor tree for three columns. It can be seen that the carry-in for each element comes from its previous column.

2.4 Decimal Multiplication

Decimal multiplication is considerably more involved and has not being researched as heavily as its binary counterpart. There are however certain studies and ideas from the 1950's, when decimal arithmetic was researched significantly, that are worth mentioning

and that sometimes have aided in more modern developments but nevertheless there are only a very few modern papers on the topic.

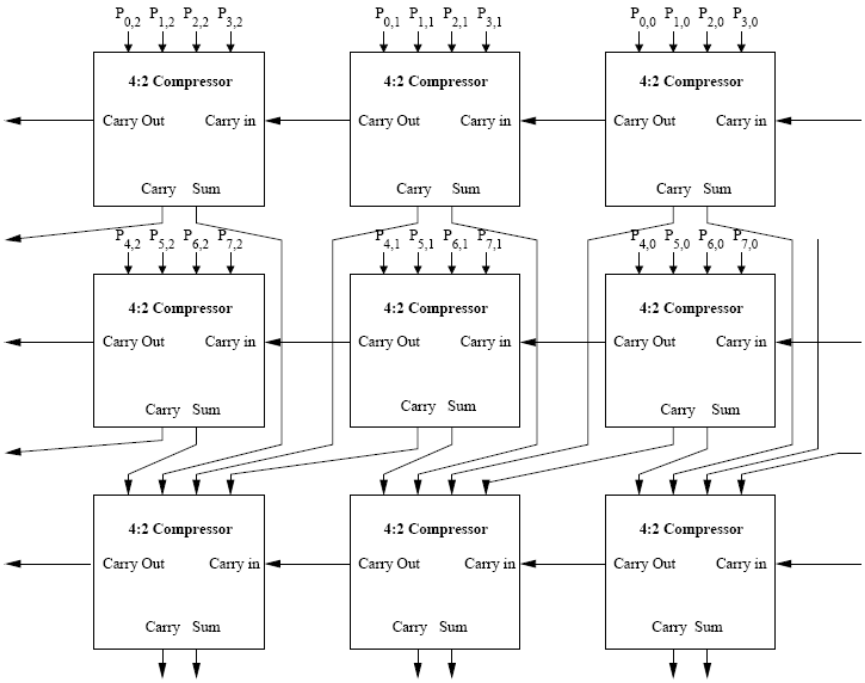


Figure 14. Three column 8-bit 4:2 binary compressor tree

One of the main difficulties lies in the generation of the partial products. In the binary case this could be accomplished by a simple AND gate which produced a single bit per digit result. In the decimal case however the inputs are not single bits but decimal numbers usually coded in BCD which implies two 4-bit inputs per digit multiplication. The result is in decimal as well and therefore a 4-bit output is produced.

One possible form of implementing the multiplication algorithm is to follow the pencil-and-paper approach, as shown in [31]. In this method a multiplication table is known beforehand and the result of the multiplication of each multiplicand digit with a multiplier can be determined by table lookup or combinational logic. A performance improvement

might be obtained if the resulting number is considered separately and divided into left digit (tens) and right digit (units). An addition accumulator can be used for each digit and the final result computed at then end. Table 7 shows the decimal multiplication table used for each component and Figure 15 shows an example of the algorithm.

Table 7. Decimal Multiplication Table, from [31].

Left Digit Component

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

Right Digit Component

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

Nevertheless, with this approach a performance constraint is that the number of additions required to perform multiplication is one greater than the number of digits in the multiplier and hence slow when compared to other methods.

An alternative approach, proposed on [31] as well, attempts to generate the partial product digits by addition instead of a table lookup. This is accomplished by over-and-over addition where the digits of the multiplier are checked one by one and the multiplicand is added an equivalent number of times to an accumulator. This approach however can be very time consuming as the number of additions required is significant.

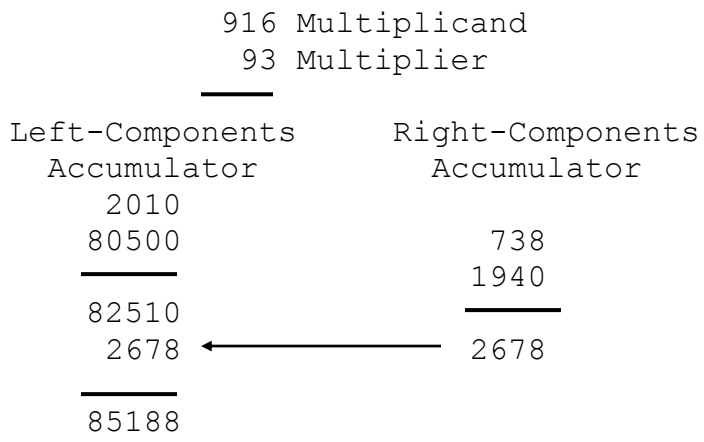


Figure 15. Decimal multiplication table Left and Right algorithm example, taken from [31].

To reduce the number of additions and speed up the multiplication a technique called Doubling and Quintupling can be used. With Doubling the value of twice the multiplicand is calculated before the actual accumulation takes place. This allows a faster accumulation as it reduces the number of additions necessary. If the multiplier digit is 5 for example the number of additions required is 3 (2+2+1) instead of 5. The value of twice the multiplicand can be obtained by adding the number to itself with a decimal adder. An important speedup can be accomplished however since the number is **only** added to itself and some combinations are never present which simplifies significantly the functions for the output result and does not require a full decimal adder. Quintupling on the other hand can also be used and it consists on calculating five times the multiplicand, 5*M*, before hand. This can be accomplished by noting that a multiplication by 5 can be performed by a multiplication by 10 (decimal left shifting) and a division by 2 (binary right sift) with certain corrections. In this way a value of 5*M* can be used for the accumulation and further reduce the number of additions required for the multiplication.

The discussed ideas have been implemented utilizing mostly a serial or sequential approach as shown in [35], [36], [37] and [38]. Two proposals however are significant and are detailed below.

2.4.1 High frequency decimal multiplier

One of the recent research papers on decimal multiplication worth mentioning is the one by Kenney, Schulte and Erle in [13]. The design proposed presents an iterative multiplication that uses some of the ideas exposed above, doubling and quintupling. In this design a set of multiplicand multiples are computed using combinational logic. In this way the values of 2*M*, 4*M*, and 5*M* are obtained and then divided into two sets of multiples: {0*M*, 1*M*, 4*M*, 5*M*} and {0*M*, 2*M*, 4*M*}. Depending on the value of the multiplier digit, a selector picks a multiple from each set and in that way their addition produces any value from 0*M* to 9*M* in a single operation. The design is further improved by allowing a two stage pipeline increasing its operating frequency and by utilizing a new decimal representation for intermediate products which speeds up the process. This representation, called overloaded decimal, permits the complete use of all 4-bits comprising the decimal digit and hence the numbers from A_{16} to F_{16} are allowed. In this way the correction back into decimal is avoided in each iteration's addition. The process continues until all digits in the multiplier operand are consumed. In the final product each digit is corrected from overloaded decimal back into BCD by adding 6_{10} when a digit lies in the range of A_{16} - F_{16}, which is easily accomplished with two level logic. A carry of value 1 is also added to the next order digit.

2.4.2 Multiplication with efficient partial product generation

A different iterative multiplication approach is proposed by Erle, Schwarz and Schulte in [14]. In this design the partial products are calculated in a digit-by-digit multiplier creating a digit-by-word (multiplicand multiple) signed-digit partial product.

The multiplier operand is examined digit by digit from least significant digit to the most significant digit and as each partial product is obtained it is accumulated with previous results to obtain the product. The most significant characteristic in this design is the recoding of the multiplier and restricting the range of each digit by utilizing a redundant representation from -5_{10} to 5_{10}. In this way the digit multiplier is simplified since there are no longer two input numbers with ten possibilities each but two inputs with values ranging from 0 to 5. The sign of the product is obtained simply by looking at the signs of the input digits. The multiples of 0 and 1 correspond to trivial multiplication results and therefore the range of input digits considered is virtually restricted to just the numbers from 2 to 5. This significantly speeds up the process since the possible input combinations are reduced from 100 to only 16 but nevertheless complicates the final product calculation since the result needs to be recoded back into BCD from a redundant representation. Table 8 shows the multiplication table for the recoded operand digit values.

Table 8. Restricted range, signed magnitude products, from [14].

x	2_{10}	3_{10}	4_{10}	5_{10}
2_{10}	04_{10} $00,0100_2$	$1\bar{4}_{10}$ $01,1100_2$	$1\bar{2}_{10}$ $01,1010_2$	10_{10} $01,0000_2$
3_{10}	$1\bar{4}_{10}$ $01,1100_2$	$1\bar{1}_{10}$ $01,1001_2$	12_{10} $01,0010_2$	15_{10} $01,0101_2$
4_{10}	$1\bar{2}_{10}$ $01,1010_2$	12_{10} $01,0010_2$	$2\bar{4}_{10}$ $10,1100_2$	20_{10} $10,0000_2$
5_{10}	10_{10} $01,0000_2$	15_{10} $01,0101_2$	20_{10} $10,0000_2$	25_{10} $10,0101_2$

The problem however with these two last propositions, [13] and [14], is that they have limited parallelization and hence are difficult to use in a pipelined system. In other words, the computation of the multiplication in both cases is highly sequential since the partial products are added by accumulation one by one as they are obtained. This forces the multiplier to be busy and unavailable for further computations in a pipeline until a result is computed. Only then it can accept a new operation which is unacceptable in most of today's floating-point units. It is therefore desirable to research methodologies which allow multiplication to be as parallel as possible as, for example, the CSA Multiplier and the Wallace tree for the binary case.

3. DECIMAL FLOATING-POINT COMPARATOR

As stated earlier, a comparator is an important element in general purpose and application specific architectures. The design of an efficient and high speed decimal comparator aids in the performance of these architectures. This design proposes a high performance 64-bit decimal floating point comparator, compliant with the current draft of the IEEE-754R standard for floating-point arithmetic. This is the first implementation of a decimal floating-point comparator compliant with the draft standard. The design can also be easily extended to support 128-bit decimal floating point numbers and even though it is not pipelined, it can accommodate pipelining to improve throughput.

3.1 Decimal floating-point comparison

Floating point comparisons are specified by the IEEE 754 standard [1]. The comparator proposed accepts two 64-bit decimal floating point numbers. In the same way as binary comparisons, the relation between the two numbers is given by four mutually exclusive conditions: greater than, less than, equal and unordered. The numbers are unordered when either one or both operands compared are Not-a-Number or *NaN*. If the *NaN* is specified as a signaling *NaN* then an Invalid exception flag bit is asserted. The result of the comparison is represented by Floating-point Condition Codes or FCC and presented in Table 4. Again, bit FCC[1] is analogous to a greater than flag (GT) and FCC[0] is analogous to a less than flag (LT).

The design however differs significantly from its binary counterpart mainly because the current IEEE 754 revision specifies that decimal floating-point numbers are not normalized and, therefore, the representation is **redundant** in nature. This implies for example that numbers 125×10^{-5} and 1250×10^{-6} are both representable and should be recognized as **equal** during comparison. Binary floating-point numbers on the other hand do not allow redundancy. Without redundancy numbers can be compared as pure binary integer numbers (since biased exponents are used) without the necessity of separating exponent and coefficient and perform alignment as proposed in [26].

The core of the comparison lies on the magnitude comparator used for the coefficients. A usual scheme to approach the comparison of the two coefficients is to subtract them. Taking into account the signs of the operands, the sign of the result determines if the comparison is greater than, less than or equal when the subtraction result is zero. This type of approach is advantageous in a system in the sense that the existing decimal floating-point adder/subtractor hardware can be utilized also for this purpose without an area increment.

A decimal comparator however is only reasonable if it provides a significant speed improvement at the cost of a small area overhead when compared to a floating-point subtraction approach. To its advantage, however, the comparator can benefit from the fact that the difference between both numbers is not required and that the result of the subtraction does not need to be rounded and recoded into decimal floating-point standard again. Furthermore, an adder/subtractor requires a greater working digit precision (extra guard bits for example) than what can be represented in the format to account for rounding and normalization [29].

Before subtraction takes place, since the floating-point numbers to be compared may have different exponents, their coefficients need to be aligned. Once alignment is performed, and taking into account the signs of the operands, the sign of the subtraction result determines if the comparison is greater than, less than or equal when the subtraction result is zero. This type of approach is advantageous in a system in the sense that the existing addition/subtraction hardware can be utilized also for this purpose without an area increment. The example in Figure 16 illustrates the classic approach to comparison where the number 3149 x 10^{23} is compared to 90201 x 10^{16}.

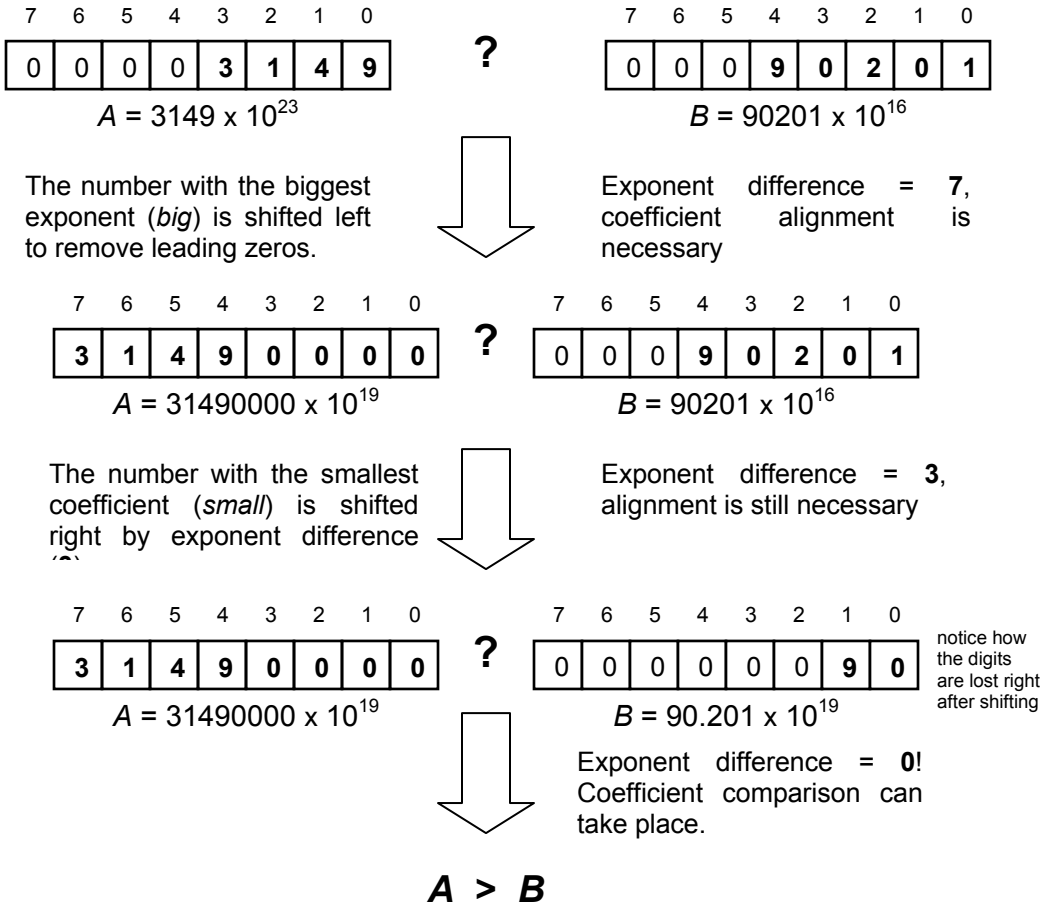


Figure 16. Classic approach to floating-point comparison.

The comparator proposed however utilizes a scheme that avoids subtraction for the coefficient comparison and instead uses a faster approach. It also avoids the use of extra digits. Only a working precision of 16 decimal digits, as in the standard, is used.

3.2 Comparator Design

An overview of the design of the decimal floating-point comparator is given in Figure 17. A decoding module, IEEE 754 decoder, converts the decimal64 operands (A and B) into a format that can be utilized for comparison. The combination field is processed and the IEEE 754 decoder outputs the number sign, exponent in unsigned 10-bit binary form, coefficient as a 64-bit BCD encoded number and tells if the number is infinite, a quiet *NaN* or a signaling *NaN*.

Since alignment is needed, the value of the difference between the operands' exponents is necessary and subtraction is, therefore, required. The 10-bit unsigned binary exponents are compared in the Exponent Comparison module. This module contains a 10-bit Carry Look-ahead Adder for fast operation and performs subtraction to determine which exponent is smaller or if they are equal and the amount of shifting necessary for the coefficient alignment. This amount is passed to the Coefficient Alignment module.

Alignment is performed by left shifting the coefficient of the number with the greatest exponent and, thus, reduce its exponent magnitude. Since the representation allows for 16 digits, shifting is limited from 0 (or no shift) to 15 digits. Larger alignment needs are evaded by treating them as special case scenarios. Consequently, this aids in maintaining the coefficient digit size (i.e. working precision) restricted to 16 digits allowing the coefficient magnitude comparison module to yield a result faster given that

its delay and complexity grows logarithmically (log_4). These special cases or scenarios will be treated in the following subsections.

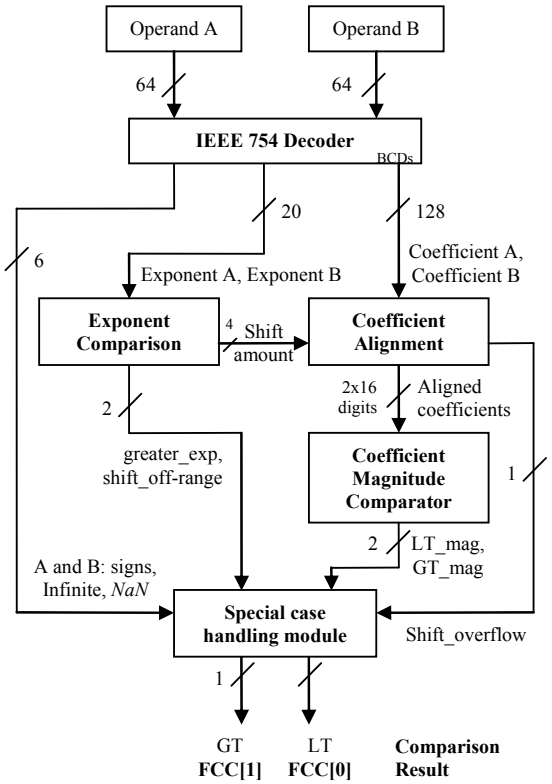


Figure 17. Decimal floating-point comparator design.

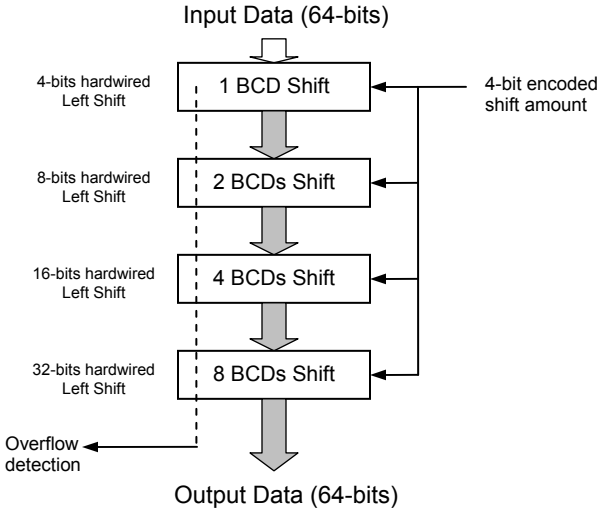


Figure 18. Coefficient Aligner, logarithmic barrel shifter.

3.3 Coefficient magnitude comparison

Once the 64-bit/16-digit coefficients are correctly aligned their magnitude can be compared. The magnitude comparator designed is based on the comparator proposed in [26] and exposed in Section 2.1 with significant performance modifications tailored specifically for BCD number comparison.

The two operands *A* and *B* are compared in stages. The first stage compares corresponding 4-bits BCD digits from each operand. Sixteen of these elements are used in parallel to process the complete 16-digit coefficients. Two output bits (*GT* and *LT*) from each of these elements indicate the result as greater than, less than or equal.Table 9 shows the magnitude comparison where $GT[i]_j$ and $LT[i]_j$ represent greater than and less than flags respectively as in Section 2.1 and *A[i]* and *B[i]* represent the digit *i* of the operand coefficients. The subscript *j* indicates the stage of the comparison.

Table 9. BCD Magnitude Comparison

GT[i]	*LT[i]*	Result
0	0	*A[i] = B[i]*
0	1	*A[i] < B[i]*
1	0	*A[i] > B[i]*
1	1	invalid

The first stage's elements (one per digit pair compared) have 8 input bits (two BCDs) and 2 single bit outputs each, producing a truth table of 256 possibilities per output. Since the numbers compared are BCD encoded, and not binary, the truth table is simplified by ignoring all entries where the 4-bit BCD numbers are greater than 9. This reduces the cases to be considered from 256 to 100 (the rest are don't cares) and reduces significantly the minimized sum-of-products expressions for $LT[i]_1$ and $GT[i]_1$.

Subsequent stages compare the results of the previous stage in the same manner forming a logarithmic tree, comparing *4*-bit $GT[i]_j$ sets to corresponding $LT[i]_j$ sets of the result where $GT[i]_j$ replaces *A[i]* and $LT[i]_j$ replace *B[i]* signals. These elements are further optimized given that $GT[i]_j$ and $LT[i]_j$ are never both asserted at the same time as shown in Table 9. The truth table cases are further reduced from *100* to *82* producing fast and simplified minimized sum-of-products expressions for $LT[i]_{j+1}$ and $GT[i]_{j+1}$ considering it is a 4-bit 2 number comparison:

$$\mathbf{LT[i]_{j+1}} = LT[i+3]_j + (GT[i+3]_j' \cdot LT[i+2]_j) + (GT[i+3]_j \cdot GT[i+2]_j' \cdot LT[i+1]_j) + (GT[i+3]_j' \cdot GT[i+2]_j' \cdot GT[i+1]_j' \cdot LT[i]_j)$$

$$\mathbf{GT[i]_{j+1}} = GT[i+3]_j + (GT[i+2]_j \cdot LT[i+3]_j') + (GT[i+1]_j \cdot LT[i+3]_j' \cdot LT[i+2]_j') + (GT[i]_j \cdot LT[i+3]_j' \cdot LT[i+2]_j' \cdot LT[i+1]_j')$$

The number of comparator stages is given by $k = log_4(4n)$ where *n* is the coefficient digit size. With *n* = *16* digits a total of $log_4(4x16) = 3$ stages are needed to yield the resulting *LT* and *GT* for the magnitude comparison of the coefficients.

Figure 19 illustrates an example using this logarithmic tree magnitude comparator. The operand size is reduced to 4 digits instead of 16 for clarity. The comparison to be computed is *A* to *B* where $A = 2713_{10}$ and $B = 2195_{10}$. Each number is separated in digits and each corresponding digit pair is compared individually. Subsequent stages group *LT* and *GT* signals together as shown. The final stage yields *GT* = 1 and *LT* = 0 as expected giving *A* > *B*.

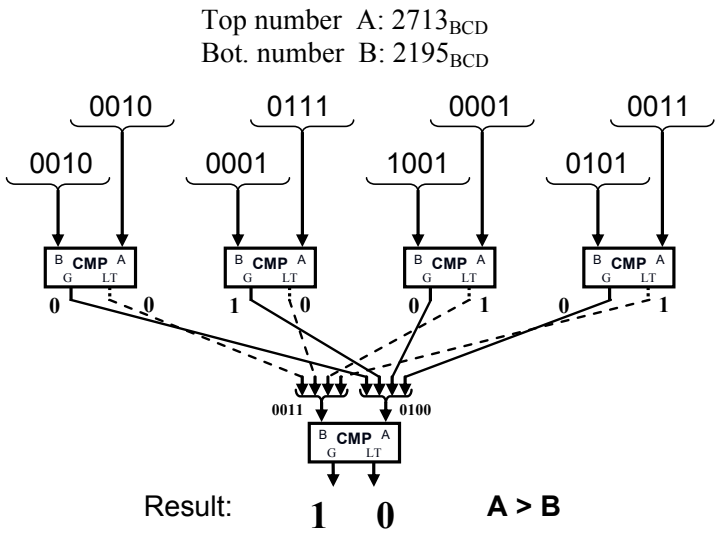


Figure 19. BCD magnitude comparator.

3.4 Special case scenarios

The result of the comparison of the two operands cannot always be obtained by a magnitude comparison of the aligned coefficients. It is possible for either of the operands to be a *NaN* (in which case the result should be *unordered)*, plus/negative infinity or plus/negative zero which are both representable. These possibilities are treated as special cases and can be determined early in the decoding phase.

The coefficients of the operands cannot always be correctly aligned within the 16-digit working precision for the magnitude compare module. There are two possibilities that can arise that are also considered as special cases. The first one occurs when the absolute value of the exponent difference between operands is greater than 15 and the second when the alignment of the coefficient produces a digit shifted out (overflow). This is treated along with the cases for *NaNs*, infinities and zeros and their respective signaling flags.

Each of the possible comparison scenario cases sets its own output signals ***LT*** and ***GT*** which affect the FCC comparison result. Five different mutually exclusive enable flag bits are used to indicate when each scenario occurs. The special case handling module, at the bottom of Figure 17, is the one responsible for determining the result according to the given situation.

3.4.1 One or both numbers is infinite

If one or both numbers are infinite the comparison result can be obtained right away by examining the sign of the operands. If the operands are *A* and *B* then: *A* is **less** than *B* if *A* is negative infinity and *B* is positive infinity OR if *A* is negative infinity and *B* is not infinity OR if *B* is positive infinity and *A* is not infinity.

$$\mathbf{LT\_inf} = (inf\_A \cdot sign\_A \cdot sign\_B') + (inf\_A \cdot sign\_A \cdot inf\_B') + (inf\_A' \cdot inf\_B \cdot sign\_B')$$

A is **greater** than *B* if *A* is positive infinity and *B* is negative OR if *A* is positive infinity and *B* is not infinity OR if *A* is not infinity and *B* is negative infinity.

$$\mathbf{GT\_inf} = (inf\_A \cdot sign\_A' \cdot sign\_B) + (inf\_A \cdot sign\_A' \cdot inf\_B') + (inf\_A' \cdot inf\_B \cdot sign\_B);$$

Note that if both numbers are positive infinite or negative infinite the result is 1-1 for *GT_inf* and *LT_inf* signaling an unordered comparison. The signaling flag that indicates this scenario is given by:

$$infinite\_flag = infinity\_A + infinity\_B$$

where *infinity_A/B* is 0 if numbers are finite and 1 if infinite.

3.4.2 Both operands are zero

In the IEEE-754 current draft the value of zero in decimal is indicated by a zero value for the number coefficient as opposed to a zero exponent in binary floating-point. If both operands are zero then both numbers are always equal regardless of their sign taking into account that +0 and -0 are both equivalent as specified by the standard.

The bits ***LT*** and ***GT*** remain unmodified (both are zero indicating equality) guarded by the flag enable bit *zero_flag* which prevents all other scenario modules to affect the result (except for infinite numbers). This flag is given by:

$$zero\_flag = (A\_zero \cdot B\_zero)\ \&\ infinite\_flag'$$

where *A*/*B_zero* is 0 if the number is non-zero and 1 if it is zero.

3.4.3 Exponent difference off-range

If the absolute value of the exponent difference between operands is greater than 15 (indicated by the signal *shift_off-range* in Figure 17) then the coefficients cannot be aligned since the working precision allows 16 digits. This means that one of the numbers is evidently greater in magnitude than the other (e.g. 512×10^{40} and 123×10^{-3}). The comparison result can be obtained by knowing which of the number's exponent is greater and by examining the signs of the numbers. The signal *greater_exp* indicates which exponent is greater.

A is **less** than *B* if exponent *B* is greater than exponent *A* (*greater_exp* = 1) and both numbers are positive OR if *A* is negative and *B* is a positive number OR if both numbers are negative and exponent *A* is greater, (*greater_exp* = 0). Numbers are never equal.

$$\textbf{LT\_off-range} = (\text{greater\_exp} \cdot \text{sign\_A'} \cdot \text{sign\_B'}) + (\text{sign\_A} \cdot \text{sign\_B'}) + (\text{sign\_A} \cdot \textit{sign\_B} \cdot \textit{greater\_exp'}).$$

A is **greater** than *B* if both numbers are positive and exponent *A* is greater than exponent *B* (*greater_exp* = 0) OR if *A* is positive and *B* is negative OR if both numbers are negative and exponent *B* is greater (*greater_exp* = 1).

$$\textbf{GT\_off-range} = (\text{greater\_exp'} \cdot \text{sign\_A'} \cdot \text{sign\_B'}) + (\text{sign\_A'} \cdot \text{sign\_B}) + \textit{(sign\_A} \cdot \textit{sign\_B} \cdot \textit{greater\_exp)}$$

The signaling flag for this case is:

$$\textbf{\textit{exp\_flag}} = \textit{shift\_offrange} \bullet \textit{zero\_flag'} \bullet \textit{infinite\_flag'},$$

where *shift_offrange* is determined by calculating the exponents difference in the exponent compare module (>15).

3.4.4 Alignment shift-out, overflow

If the exponent difference is within range (<15), alignment of the coefficient takes place by left shifting. If a digit is shifted out (*shift_overflow*) then the comparison result can be determined by knowing the signs of the numbers.

The greatest exponent determines which of the two numbers compared is the one being aligned with respect to the other. When *greater_exp* = 0, exponent A is the one aligned and vice-versa. If alignment overflow occurs *A* is **less** than *B* if: *A* is negative and *B* is positive OR if *A* alignment overflows (magnitude of number *A* is greater) and it is negative OR if *B* alignment overflows (magnitude of number *B* is greater) and *B* is positive.

$$\textbf{\textit{LT\_of}} = (sign\_A \cdot sign\_B') + (greater\_exp' \cdot sign\_A) + (greater\_exp \cdot sign\_B')$$

A is **greater** than B if: *A* is positive and *B* is negative OR if *A* alignment overflows and it is positive OR if *B* alignment overflows and it is negative.

$$\textbf{\textit{GT\_of}} = (sign\_A' \cdot sign\_B) + (greater\_exp' \cdot sign\_A') + (greater\_exp \cdot sign\_B)$$

The equation for the signaling flag of this scenario is:

$$\textbf{\textit{align\_flag}} = shift\_overflow \cdot exp\_flag' \cdot zero\_flag' \cdot infinite\_flag'$$

where *shift_overflow* is asserted during alignment if a digit is shifted out.

3.4.5 Coefficient comparison

If the exponent difference is within range and no shift overflow occurs after alignment then this indicates that the coefficients are correctly aligned and their comparison can be executed by the dedicated coefficient magnitude comparator module discussed in Section 3.3. The output signals from this module (*LT* and *GT*) are renamed to avoid confusion as *LT_mag* and *GT_mag*. Nevertheless, the final result of the comparison is not yet determined as the relationships of greater than, less than or equal do not only depend on which number's magnitude is greater but also on their signs. The result of the comparison in this scenario is then given by the following conditions.

A is **less** than *B* if: *A* is negative and *B* is positive OR *A* and *B* are positive and the magnitude of *A* is less than the magnitude of *B* OR *A* and *B* are negative and the magnitude of *A* is greater than the magnitude of *B*.

$$\textbf{\textit{LT\_cmp}} = (sign\_A \cdot sign\_B') + (sign\_A' \cdot sign\_B' \cdot LT\_mag) + (sign\_A \cdot sign\_B \cdot GT\_mag)$$

A is **greater** than *B* if: *A* is positive and *B* is negative OR if *A* and *B* are both positive and the magnitude of *A* is greater than *B* OR if both numbers are negative and the magnitude of *A* is less than *B*.

$$\textbf{GT\_cmp} = (sign\_A' \cdot sign\_B) + (sign\_A' \cdot sign\_B' \cdot GT\_mag) + (sign\_A \cdot sign\_B \cdot LT\_mag)$$

Mag_flag determines if the outputs *GT_cmp* and *LT_cmp* should affect the final result and it is given by:

$$\textbf{mag\_flag} = align\_flag' \cdot exp\_flag' \cdot zero\_flag' \cdot inifinite\_flag'$$

In other words, the magnitude comparison of the coefficients is only valid (through *mag_flag*) if none of the previous enable flags was triggered.

The final equations for the comparator considering all possible scenarios are:

$$\textbf{GT} = unordered + ((GT\_inf \cdot infinite\_flag) + (GT\_off\text{-}range \cdot exp\_flag) + (GT\_of \cdot align\_flag) + (GT\_cmp \cdot mag\_flag)),$$

$$\textbf{LT} = unordered + ((LT\_inf \bullet infinite\_flag) + (LT\_off\text{-}range \bullet exp\_flag) + (LT\_of \bullet align\_flag) + (LT\_cmp \bullet mag\_flag)).$$

The signal unordered is asserted when either of the operands is a *NaN*. If this occurs the result is overridden and is always unordered (*GT*=1, *LT*=1) as specified in Table 4.

The signals *GT* and *LT* are finally produced in the special case handling module. It is the one that receives the flags indicating the different scenarios and is responsible for the handling of the different comparison cases described in this section.

3.5 Combined binary floating-point, two's complement and decimal floating-point comparator

Given the similarity of approaches of the decimal comparator described and its binary counterpart exposed in Section 2.1, a single design capable of handling 32-bit and 64-bit two's complement numbers, single and double precision binary floating-point and 64-bit decimal floating-point numbers is interesting since it would result especially useful to reduce costs in processors, by allowing the same hardware to be used to compare all three data types.

The main difference between the binary and the decimal comparator schemes is that decimal floating-point representation is redundant, as stated before, and therefore requires alignment while binary does not. Binary 32-bit floating-point numbers only require a sign extension given that 32-bit and 64-bit number formats can both be handled by the same hardware. The logic that can be shared between both cases however is the core of the comparators, the magnitude comparison module which, in the decimal case, comes into effect after alignment.

The magnitude comparator logarithmic tree for the decimal case is composed of comparator elements that handle 4-bit BCD digits. Each element had two digit inputs (two 4-bit BCDs) as opposed to handling 2-bit pairs as in the binary case (Section 2.1). Optimization of the BCD element was possible since in BCD no numbers are encoded after 9_{hex} and hence A_{hex}, B_{hex}, C_{hex}, D_{hex}, E_{hex} and F_{hex} can be ignored providing further simplification. An additional benefit is the fewer number of stages necessary since 4-bit digits are compared instead of 2-bit pairs. Nevertheless the binary pair comparator element can be used for the decimal case instead of the BCD and provide a way of saving hardware since it would be used by both formats. Tests and simulations were run

however to see the impact of using the bit pair comparison module from the previous section for the decimal comparator and the results justified the joint design of the module proposed. Furthermore, less area would be required when implemented on a system since the decoding of the IEEE 754 decimal floating-point can be handled by the decimal floating-point unit potentially already existent in the system.

An overview of this combined design is shown in Figure 20. The 3-bit signal *Sel* determines the format type of the operands, 32-bit or 64-bit two's complement, binary floating-point or decimal floating-point. If the operands are binary then the sign extension module sign extends 32-bit numbers into 64-bit so that the same hardware can be utilized. In the decimal case the input for the magnitude comparator is obtained after decoding and coefficient alignment. The exception and special case module handles the result of the magnitude comparator taking into account the signs of the numbers, the data type and the flags for overflow, off-range and others exposed in Section 3.4, for the decimal case.

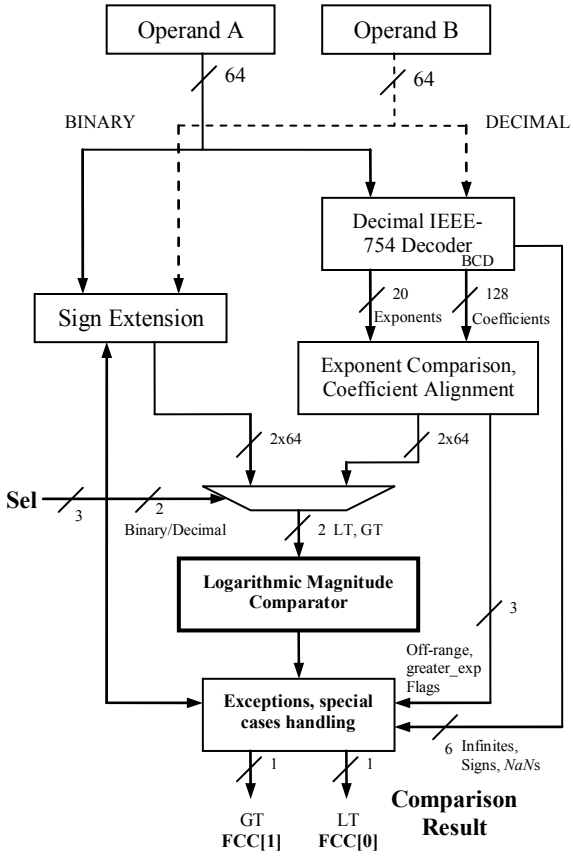


Figure 20. Combined two's complement, binary and decimal floating-point comparator.

4. EXPERIMENTS FOR DECIMAL FLOATING-POINT DIVISION BY RECURRENCE

One method widely used for division is performed by recurrence or sequentially. In this method, the quotient is represented by a chosen radix and a digit is produced after each iteration. The quotient digit can also be selected from a redundant digit set as this approach has noteworthy speed and cost advantages.

The main difficulty using a digit recurrence algorithm lies in the quotient digit selection function or QDS. Several studies have been made to simplify or improve this function. The Kornerup study presented in [40] shows an accepted analytical approach to determine a minimum number of digits required for the QDS function. This theory, however, is specific to the binary case and, hence, requires modification to be applied to the decimal case. This study attempts to provide an insight into the implementation feasibility of a decimal digit recurrence divider utilizing the recurrence division theory.

4.1 Decimal Division by Digit Recurrence Theory

As discussed previously, when implementing division by recurrence, the quotient digit of radix *r* lies within a symmetric redundant selection set of consecutive integers given by:

$$q_j \in D = \{\bar{a},...,\bar{1},0,1,...,a\} \; \forall \; a \geq \frac{r}{2}, \tag{1}$$

such that $\bar{a} = -a$. The redundancy factor or measure of redundancy for a digit set is defined by:

$$\rho = \frac{a}{r-1} \text{ with } \frac{1}{2} < \rho \leq 1. \tag{2}$$

The main equation when implementing division by recurrence for a dividend, x, and divisor, d, is given by [41]:

$$w[j+1] = rw[j] - dq_{j+1} \quad , \tag{3}$$

where r denotes the quotient radix, q_{j+1} the selected quotient digit and $w[j]$ the partial remainder in iteration j. Naturally, in our case, the radix is decimal or $r = 10$.

In order for the recurrence in (3) to be valid through all iterations and guarantee a result, two basic conditions for the QDS should be met: **containment** and **continuity**. And, the value of the quotient digit q_{j+1} is given by the selection function:

$$q_{j+1} = \text{SEL}(rw[j], d). \tag{4}$$

The **containment** condition specifies that the quotient digit selected must maintain the next partial remainder bounded to satisfy convergence of the algorithm, or:

$$-\rho \cdot d \leq w[j] \leq \rho \cdot d \, . \tag{5}$$

This is summarized in Robertson's diagram, shown in Figure 21, where the limits on the vertical axis for $w[j+1]$ are noted by the horizontal doted lines. The selection interval of $rw[j]$ for which it is possible to select $q_{j+1} = k$ and keep the next residual bounded is defined as $[L_k(d), U_k(d)]$. Each interval, as shown in Figure 21, defines a specific interval for a given quotient digit (e.g. $q_{j+1} = k-1$ produces the interval given by $[L_{k-1}(d), U_{k-1}(d)]$).

Expressions for $L_k(d)$ and $U_k(d)$ can be obtained from Robertson's diagram defined by [41][29]:

$$L_k(d) = (k - \rho) \cdot d \text{ and } U_k(d) = (k + \rho) \cdot d \,. \tag{5}$$

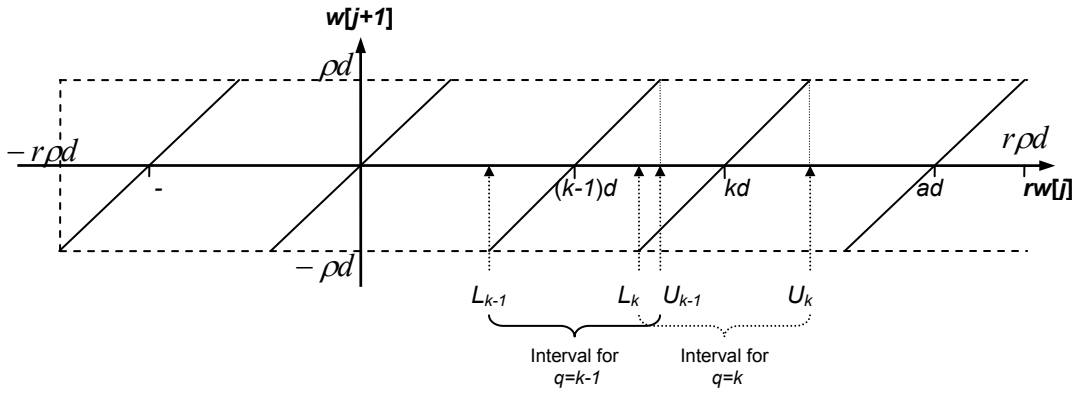


Figure 21. Robertson's diagram showing selection intervals for *q=k-1* and *k*.

The **continuity** condition ensures that for any possible *rw[j]* (horizontal axis in Robertson's diagram) there exists a selectable quotient digit *k* (i.e. *rw[j]* lies always within a selection interval $[L_k, U_k]$), otherwise a quotient digit would not be selectable. Therefore, the overlap present between two consecutive intervals must exist or be equal to zero, at minimum, or $L_k \leq U_{k-1}$. This condition is imposed by:

$$L_k \leq S_k \leq U_{k-1}\,, \tag{6}$$

where S_k denotes the partition points within the selection interval $[L_k, U_{k-1}]$ such that the QDS returning q_{j+1} may be defined by [40]:

$$S_k \leq rw[j] < S_{k+1} \Rightarrow q_{j+1} = k\,. \tag{7}$$

4.2 Quotient Digit Selection

The overlap is critically important, since it allows an inexact value of the divisor and the partial remainder to determine a suitable quotient digit. In this sense, only a limited number of leading digits of both divisor and partial remainder are required. With this consideration, the truncated partial remainder, $r\hat{w}[j]$, is defined as:

$$r\hat{w}[j] = rw[j] + \varepsilon_{rw} \tag{8}$$

where ε_{rw} denotes the truncation error. Carry-save representations are often utilized for division by recurrence algorithms when computing (3), because carry-propagate adders would lengthen the critical path excessively. The truncation error, using a carry-save representation, is defined by [40]:

$$0 \le \varepsilon_{rw} < 2 \cdot ulp(r\hat{w}[j]) . \tag{9}$$

In a similar way, the truncated divisor is given by:

$$d = \hat{d} + \varepsilon_d \text{ with } 0 \le \varepsilon_d < ulp(\hat{d}) . \tag{10}$$

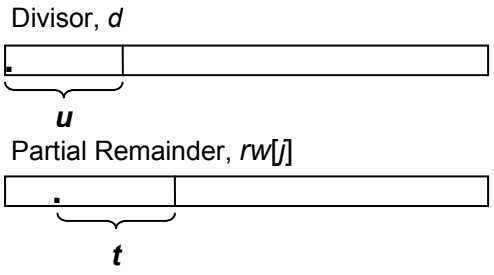


Figure 22. Truncated inputs to the QDS function.

Figure 22 illustrates the truncation of both divisor and partial remainder. The number of digits to the right of the radix point is given by ***u*** and ***t,*** respectively. Since both divisor and partial remainder utilize decimal representations or radix 10, it follows that:

$$ulp(r\hat{w}_i) = 10^{-t}, \tag{11}$$

$$ulp(\hat{d}) = 10^{-u}, \tag{12}$$

where *ulp* indicates unit in the last place (less significant digit). Therefore, the truncated divisor can be represented as an integer multiple *m* of an $ulp(\hat{d})$:

$$\hat{d} = m \times 10^{-u}. \tag{13}$$

Division by digit recurrence theory is often implemented for fractional normalized divisors [41]. Figure 23 illustrates the reasoning behind this, since a non-normalized divisor would require an infinite precision as *d* approaches zero. Hence, the values of *d* are normalized and bounded by:

$$1/10 \le d < 1, \tag{14}$$

which implies, due to (13), that:

$$10^{u-1} \le m < 10^{u}. \tag{15}$$

Following the analysis approach in [40], Figure 23 shows the partition of the interval [L_k ,U_{k-1}] by S_k, defined in (6), as a function of $\hat{d}$ and not *d*. Below S_k, the value of the selected quotient digit is $q = k-1$, and above S_k, $q = k$. $S_k(\hat{d})$ is now a staircase function due to the truncation of both divisor and partial remainder and indicates rectangles

where a given quotient digit can be selected due to quantization. The value of $S_k(\hat{d})$ can also be expressed as an integer multiple of $ulp(r\hat{w}_i)$ by constants given by k and m, or $s_{k,m}$:

$$S_k(\hat{d}) = s_{k,m} \times 10^{-t} . \tag{16}$$

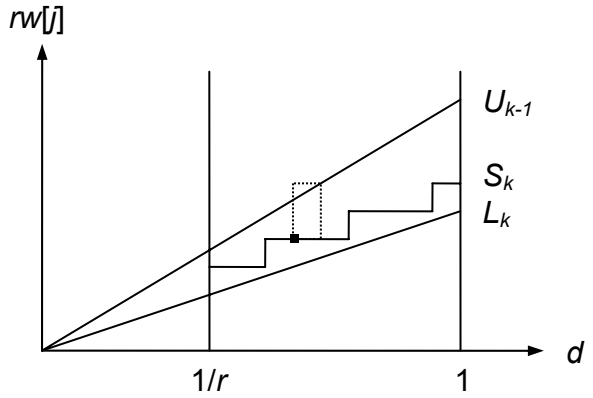


Figure 23. PD Diagram, Sk selection points as a function of truncated d, [40].

The dotted rectangle in Figure 23 has its lower left hand corner at $(\hat{d}, S_k(\hat{d}))$ and is limited by:

$$\hat{d} \le d < \hat{d} + ulp(\hat{d}) \tag{17}$$

and $$S_k(\hat{d}) \le rw_i < S_k(\hat{d}) + 2 \cdot ulp(r\hat{w}_i) . \tag{18}$$

The study by Kornerup defines boundary conditions on this rectangle to determine the minimum amount of digits after truncation for the partial remainder and the divisor; ***t*** and ***u*** respectively. The rectangle should lie above S_k and, therefore, the boundary condition on its right hand corner yields:

$$L_k(\hat{d} + ulp(\hat{d})) = (k - \rho)(\hat{d} + ulp(\hat{d})) \le S_k(\hat{d}) . \quad (19)$$

It follows, using (5), (12) and (16):

$$\left\lceil (k - \rho)(m \cdot 10^{-u} + 10^{-u}) \right\rceil = s_{k,m} \times 10^{-t} , \quad (20)$$

$$\left\lceil 10^{t-u}(k - \rho)(m + 1) \right\rceil = s_{k,m} . \quad (21)$$

The height of the rectangle is of $2 \cdot ulp(r\hat{w}_i)$. Nevertheless, consecutive rectangles aligned vertically are spaced by one $ulp(r\hat{w}_i)$ (resolution of S_k). Overlapping rectangles from the bottom should have a value of $k - 1$ and, therefore, the midpoint on the left edge should lie under U_{k-1}. This boundary condition, combined with (5), yields:

$$S_k(\hat{d}) + ulp(r\hat{w}_i) \le U_{k-1}(\hat{d}) = (k - 1 + \rho) \cdot \hat{d} . \quad (22)$$

Again, using (11) and (16) on this inequality gives:

$$s_{k,m} \le \left\lfloor 10^{t-u}(k - 1 + \rho)m - 1 \right\rfloor . \quad (23)$$

Combining (21) and (23) yields floor and ceiling expressions for the possible values of S_k:

$$\left\lceil 10^{t-u}(k - \rho)(m + 1) \right\rceil \le \left\lfloor 10^{t-u}(k - 1 + \rho)m - 1 \right\rfloor . \quad (24)$$

Rearranging terms results in an expression of the form:

$$\left\lceil Am + B \right\rceil \le \left\lfloor (A + C)m \right\rfloor , \quad (25)$$

with $\boldsymbol{A = 10^{t-u}(k-\rho)}$, $\boldsymbol{B = 10^{t-u}(k-\rho)+1}$ and $\boldsymbol{C = 10^{t-u}(2\rho-1)}$. For the nontrivial solution, where the quotient selected is zero and given condition (2), it is seen that $A \geq 0$, $B \geq 1$ and $C > 0$ for $k \geq 1$. For condition (25) to withstand, it is necessary that $C \cdot m \geq B$. Nevertheless, the stronger condition $C \cdot m - B \geq 1$ allows a minimum of one integer between the floor and ceiling functions yielding:

$$10^{t-u}(2\rho-1)m - 10^{t-u}(k-\rho) - 1 \geq 1 . \tag{26}$$

This condition should hold at the extreme case for the values of m and k in order for (25) to be valid. This occurs at the maximum value for the quotient digit $k = a$ and, due to (15), at the minimum case when $m = 10^{u-1}$. Rearranging terms gives:

$$10^{-t} \leq \frac{10^{-1}(2\rho-1) - 10^{-u}(a-\rho)}{2} , \tag{27}$$

which produces a minimal value for t for a known u. Furthermore, it is clearly seen that the numerator in (27) should be positive enabling a minimum u to be obtained as:

$$10^{-u} < \frac{2\rho-1}{10(a-\rho)} , \tag{28}$$

4.3 Considerations for the IEEE-754 Decimal Case

The application of the previous analysis, with radix = 10, to the IEEE 754 revision draft for decimal floating-point requires some considerations. Most significantly, the revision draft specifies that decimal floating-point numbers are not normalized and, therefore, the representation is redundant in nature. This implies for example that numbers 125 x 10^{-5} and 1250 x 10^{-6} are both representable and are equal in magnitude. The standard also specifies that the mantissa is represented as an integer and not a fraction with a leading

binary 1, as in the binary case (i.e. $0.1xxxx\ldots_2$). This complicates the algorithm application, since both the divisor and dividend operands in the binary case are limited in range to $0.1_2 \leq x < 1_2$.

Unsigned integer division has operands $0 \leq x < r^n - 1$ and $0 \leq d < r^n - 1$. The division result produces a quotient q such that [41][29]:

$$q = \lfloor x / d \rfloor. \tag{29}$$

As mentioned previously, basic integer division algorithms require full-precision for the QDS function. To apply fractional division theory, the divisor d should first be normalized, by shifting, so that the most-significant bit is a nonzero digit. With a shift of p digits, the normalized divisor d^* is:

$$d^* = 10^p d\,. \tag{30}$$

Consequently, the resulting quotient is:

$$q = \lfloor x / d \rfloor = 10^p \lfloor x / d^* \rfloor. \tag{31}$$

To use the algorithm, fractional operands are modified as defined by:

$$x_f = x \times r^{-n}, \tag{32}$$

$$d_f = d^* \times r^{-n}. \tag{33}$$

Expressions (27) and (28) can be used to obtain minimum bounds for the number of decimal digits of the partial remainder and divisor, t and u respectively. There is a

choice, however, in the value of *a*, or the amount of redundancy as shown in (1). In the decimal case *a* can vary from 6 to 9.

As redundancy is incremented (*a* increases), the overlap described in (6) is augmented thus simplifying the QDS function by allowing for selection of the quotient digit and consequently a smaller look-up table. On the other hand, a greater value of *a* complicates the generation of the quotient digit multiples ($q_{j+1}d$) needed for the iterative algorithm (3). For example, with *a* = 9 the possible divisor multiples required are (–9*d,* –8*d*, …, –1*d*, 0, 1*d*, …, 8*d*, 9*d*). Nevertheless, as *a* is decremented and the possible quotient multiples are reduced, the additional digit inputs to the QDS function are incremented as more precision is required. Since each digit is a decimal digit the size of a look-up table for the QDS would increase by an order of magnitude with each additional digit required. Therefore, the smallest look-up table size is achieved with *a* = 9 and, hence, a maximum redundant digit set with ρ = 1, from (2). In this case, (28) and (27) yield: $\boldsymbol{u \geq 2}$ and $\boldsymbol{t \geq 2}$ implying that only 2 decimal digits are required for the divisor and the partial remainder.

The shifted partial remainder, however, can still be within the allowed boundary (5) but be greater than 1 in which case integer digits are needed. Since the divisor is normalized (30) its range is limited to $1/10 \leq d < 1$, this observation is also shown in Figure 23 with the vertical line at $1/r$. The possible range for the shifted partial remainder is then given by:

$$\left|rw[j]\right| = \left|10 \cdot w[j]\right| \leq 10 \cdot \rho d < 10\,, \tag{34}$$

due to the containment condition given in (5). This implies that at most a single integer digit is required. The total number of digit inputs to the QDS function is **5 digits**, 2

decimals for the divisor (. *xx*) and 2 decimals and an integer for the partial remainder (*x*. *xx*). A table based QDS function will then have 5 decimal inputs and a decimal output. Considering a 5-bit encoding for the signed decimal quotient digit output the total number of QDS entries in the table would be:

$$\mathbf{10^5 \times 5\ bits = 100{,}000 \times 5\ bits}.$$

The division by recurrence algorithm requires a subtraction and a multiplication of the truncated divisor by the quotient digit which can be positive or negative. Since the numbers treated are decimal this complicates significantly the arithmetic operations involved.

Furthermore, a significant complication of using a maximal redundant quotient digit set is the generation of extra divisor multiples (–9*d*, …, 0, …, 9*d*), as discussed previously. The proposed scheme utilizes ideas from decimal multiplication presented in [31]. The divisor multiples ($q_{j+1}d$ product) generation starts by computing a priori the multiples 0, *d*, *2d, 4d* and *5d* which can be added in combination to create all possible divisor multiples from 1*d* to 9*d*. Figure 24 shows an overall scheme of the decimal division by recurrence design.

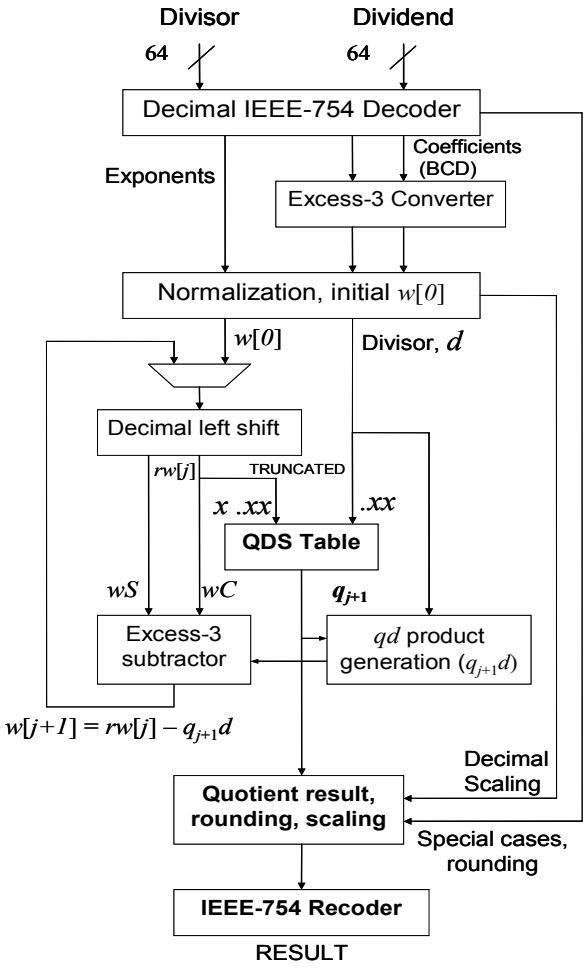


Figure 24. Decimal division by digit recurrence implementation.

5. DECIMAL PARTIAL PRODUCT REDUCTION

A few of the approaches to decimal multiplication were described in Section 2.4. The different methods discussed included serial or iterative accumulate addition to obtain the multiplication result. Parallel multiplication however is used extensively in binary floating-point hardware ([42], [43]) and is of importance if performance is to be extended to the decimal case.

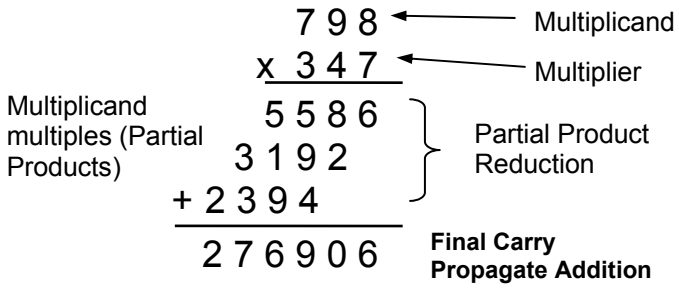


Figure 25. Multiplication Algorithm

Parallel multiplication can be divided in three main steps, as illustrated in Figure 25. The first step entails partial product generation where the multiplicand multiples are obtained. Then, partial product reduction occurs using a fast addition scheme to reduce the partial products to two. Finally, a carry propagate addition is necessary to obtain the final result.

The overall performance of the multiplier, therefore, is closely related to the individual performance for these stages. However, improvement in partial product

reduction for example, often increases complexity in partial product generation. This is the reasoning behind binary methods, like Booth encoding, where the number of partial products to be added is reduced at the expense of more complex multiplicand multiple generation through recoding [44][45]. Unfortunately, this condition can offset the gains in performance of the resulting multiplier.

As discussed in Section 2.3, binary multiplication with tree multipliers typically use carry-save adders to repeatedly reduce the partial product matrix until only two rows remain which are then added using a fast carry-propagate adder to form the final product. Although tree multipliers are typically much faster algorithmically than array multipliers (see Section 2.3), they produce an irregular structure which can affect their performance.

Traditional decimal codes are different than binary codes in that more information per bit has to be coded into the digital logic. The most common decimal encoding is 4-bit Binary Coded Decimal (BCD) which represents decimal codes 0 through 9. This code is also referred to as BCD-8421 where the numbers 8421 represent the weight of each bit in the encoding. BCD-8421 has the advantage that each decimal number is represented in a common binary number system and, hence, some of the binary operations can be performed with regular binary logic structures [31].

Although BCD-8421 codes are straightforward, they have two distinct disadvantages. First, the binary representation of ten through fifteen has no meaning and must be eliminated. Another major disadvantage is that BCD-8421 is not self-complementing, whereas, a self-complementing BCD code is one where the 9's complement of the decimal digit may be obtained by changing the 1's to 0's and 0's to 1's (bit inversion) [31]. The 9's complement operation is necessary to perform subtraction in much the

same way as two's complement numbers are used to perform subtraction with binary numbers.

Although these two disadvantages make BCD-8421 more challenging to work with, simple Boolean logic can be used to obtain its 9's complement:

$$C_0 = \overline{T_0}$$

$$C_1 = T_1$$

$$C_2 = T_1\overline{T_2} + \overline{T_1}T_2$$

$$C_3 = \overline{T_1 + T_2 + T_3}$$

where the letters *T* and *C* refer to true and complement digit, respectively.

Efficient addition of decimal numbers is an interesting topic since it is not only used as a standalone operation but also required in other calculations like multiplication and division. Multi-operand addition, where more than two numbers are added, is of particular importance since it can be used to reduce the partial products that occur in a multiplication operation (see Figure 10). To avoid the time consuming carry propagation in multi-operand addition, two different schemes used in binary radix numbers are also applicable to the decimal case. These are signed-digit addition and carry-save addition, explored in this work.

5.1 Decimal Carry-Save Addition

One of the most common elements within partial product reduction is the full adder cell or the 3:2 counter, as stated in Section 2.2.1. As with the binary case, decimal partial product reduction can also utilize 3:2 counters except that each digit is 4-bits long to accommodate the BCD-8421 encoding. In this case, four 3:2 counters are used for each bit of the BCD digit as shown in Figure 26. Figure 27 shows how this scheme would be implemented with a block level diagram.

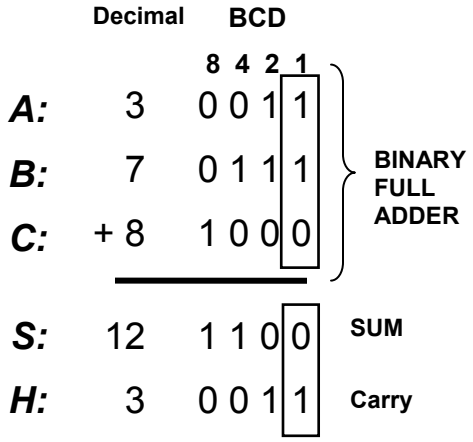


Figure 26. Binary full adder cells used for decimal carry-save addition.

The result of the addition is obtained by adding the sum and the shifted carry vector. Each carry bit in *H*, in Figure 26, represents the carry input for the next significant bit. The addition of both vectors assumes that *H* is shifted left one bit or, in binary, multiplied by two.

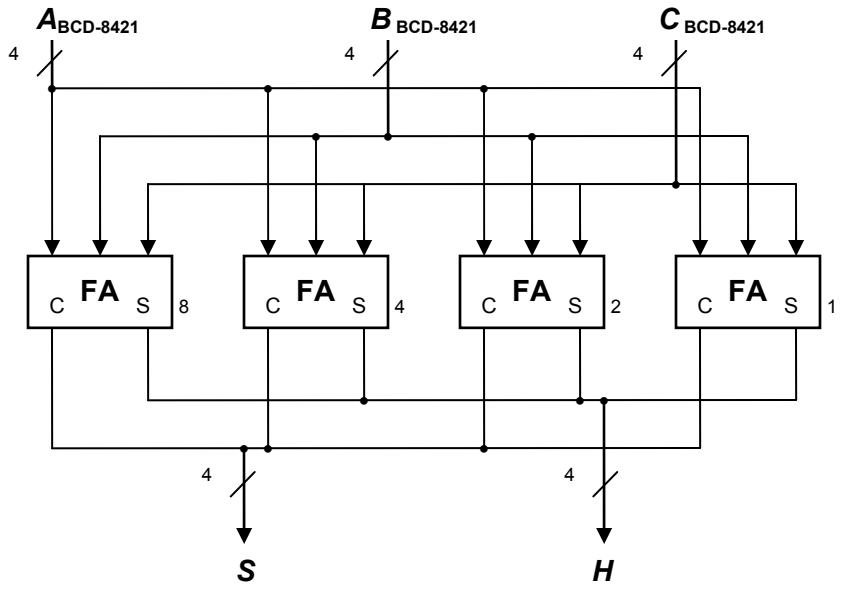


Figure 27. Block diagram for full adder cells used for decimal Carry-Save addition.

The function of the 3:2 counter or full adder can be summarized using the following equation:

$$A + B + C = S + 2 \times H$$

where *A*, *B*, *C*, *S* and *H* are decimal numbers with

$$A = \sum_{i=0}^{3} a_i \times r_i$$

such that a_i is the bit value of *A* at position *i* and r_i the corresponding weight ($r_3 r_2 r_1 r_0$ = 8421 for BCD-8421). Although using binary 3:2 counters is relatively simple, it has one major disadvantage for BCD-8421 coding. As seen in Figure 28, the sum vector *S* is out of range since the sum *S* = 1100_{BCD} is not a valid decimal representation and the addition of *S* and *H* yields a result in binary as opposed to correctly displaying a valid

BCD-8421 number. Consequently, the conversion to BCD would require additional logic and, hence, it is not efficient.

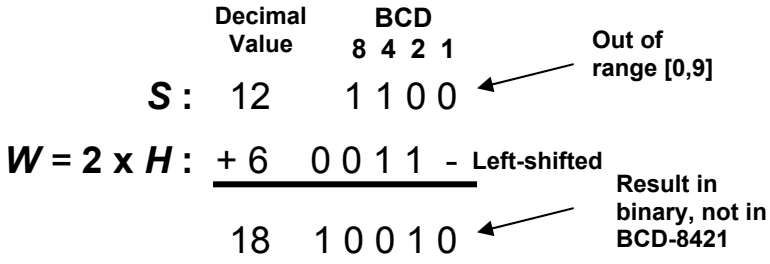


Figure 28. Result of the addition, carry vector shifted left.

One solution can be obtained by using a different weighted BCD representation during the summation process. That is, the bits are recoded to another valid decimal representation which allows the carries to be generated correctly when a shift occurs. To find the best BCD representation to utilize for the recoding, it is important to look at an important property of adding two numbers. In order to avoid the out of range result, decimal codes are employed such that the sum of the weights of each 4-bit decimal code is equal to 9. In the previous example, since the code BCD-8421 was used, it was possible to represent digits with a value greater than 9 (10 through 15, since we had 4-bits BCD). If the maximum possible value was 9 instead (since the sum of the weights of all 4-bits is 9) then the result will never be out of range. This property is also utilized with binary signed digit adders to avoid carry-propagation [29].

The following BCD codes: 5211, 4311, 4221, and 3321 can satisfy this condition. Additionally, an advantage of these encodings is that they are self-complementing, meaning that the 9's complement of the number can be obtained by a simple bit inversion of each of the 4-bits [31]. Using this trivial simplification allows the ten's

complement of the number to easily be obtained in much the same way as the two's complement for binary numbers is performed. Table 10 shows some of the mentioned codes.

Table 10. 8421, 4221 and 5211 BCD representations.

Decimal	BCD-8421	BCD-4221	BCD-5211
0	0000	0000	0000
1	0001	0001	0001 | 0010
2	0010	0010 | 0100	0100 | 0011
3	0011	0011 | 0101	0101 | 0110
4	0100	1000 | 0110	0111
5	0101	1001 | 0111	1000
6	0110	1010 | 1100	1001 | 1010
7	0111	1011 | 1101	1100 | 1011
8	1000	1110	1110 | 1101
9	1001	1111	1111

Furthermore, since the value of 2x*H* (the carry vector) is required to obtain the final sum, a BCD coding that facilitates a multiplication by 2 is desirable to use. This is because, in binary, a multiplication by 2 is easily accomplished by a bitwise left shift. However, in the self-complementing codes shown above the weights do not increase in powers of two and, hence, shifting cannot be applied directly. Nevertheless, if the number is re-encoded this might be accomplished in a simpler way.

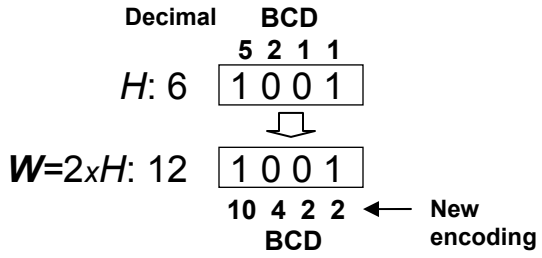


Figure 29. Multiplication by 2, recoded result.

One potential solution for recoding the carries into a self-complementing output is to use the 5211 code, shown in Table 10. By using this code, a multiplication by 2 would result in a 10, 4, 2, 2 encoding, as seen in Figure 29 without the need to even modify the bits. Although this encoding is not useful, since it does not satisfy the two conditions described earlier (self-complementing and sum in-range), the most-significant bit can be passed to the next significant digit or column. The resulting BCD code will be 4221 as illustrated in Figure 30.

In summary, if a BCD-5211 recoding is utilized, a shift asserts a carry-out for the 10-weight and allows the decimal code to be transformed to BCD-4221, which is self-complementing. The bit shifted out is the carry out to the next decimal digit and the space opened in the least significant bit position after the shift would be available to receive a carry input from the previous significant digit. For the least-significant decimal digit, it is assumed that this value is 0.

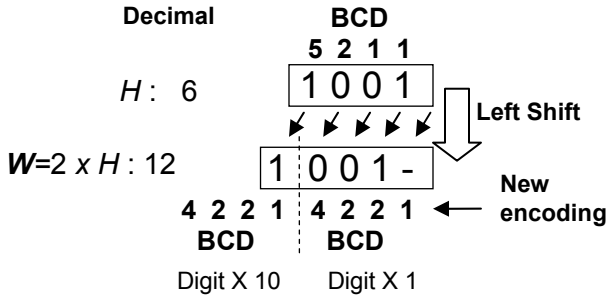


Figure 30. Multiplication by 2, result in BCD-4221.

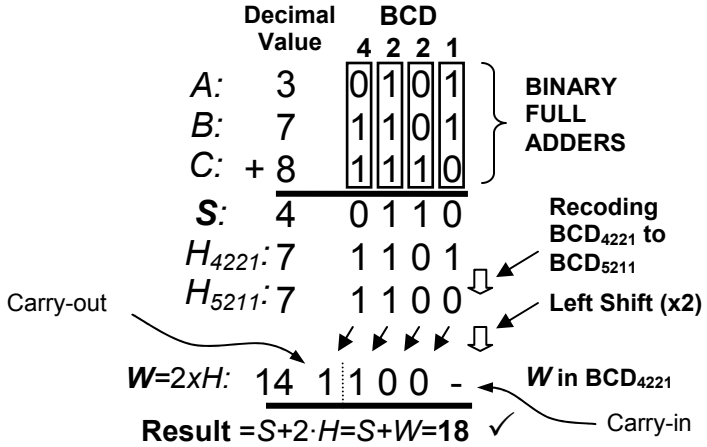


Figure 31. Decimal 3:2 counter with BCD recoding example.

Figure 31 also shows the multiplication by 2 operation on *H*. *H* is shifted left after recoding allowing a carry-in to be accepted ("-" in the example) and generating a carry-out ("1" in the figure). The value of 2x*H* is then composed of a decimal digit and a single bit decimal carry-out. The decimal digit is *W* with a value of $1000_{BCD\text{-}4221}$ or 4_{10} considering a carry-in of zero to its LSB. The single bit carry-out of 1 represents a value of 1 in the tens place and, together with *W* = 4, represents 14. *W* has the same arithmetic weight as *S*. The operation is summarized as:

$$A + B + C = S + 2 \times H = \boldsymbol{S} + (\textbf{Carry-out} \times 10) + \boldsymbol{W} + \textbf{Carry-in}.$$

The following figure shows the implementation of the decimal 3:2 counter:

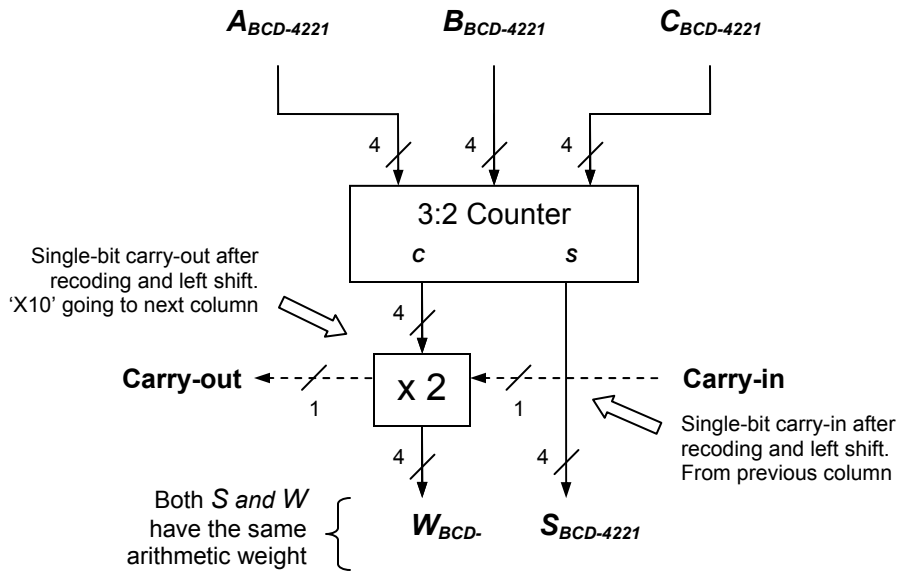


Figure 32. Block diagram for decimal 3:2 counter [44].

Although this recoding works nicely, a conversion is necessary to convert the input of the 3:2 counter into the desired decimal code (4221). Using Table 10, a BCD-8421 number can be converted into BCD-4221 through simple logic utilizing the following Boolean equations:

$$H0 = D1$$

$$H1 = D1 + D3$$

$$H2 = D3$$

$$H3 = D2 + D3,$$

where $D_3D_2D_1D_0$ represent the BCD-8421 digit and $H_3H_2H_1H_0$ is the BCD-4221 decimal digit. The counter then takes three 4-bit decimal numbers as inputs in BCD-4221 decimal code. These binary full adders perform the bit-wise addition which results in a

sum vector *S* and a carry vector *H*, both in BCD-4221. When *H* is recoded into BCD-5211, this allows the left shift to obtain the correct decimal value in BCD-4221.

It is important to see that by using recoding the correct result is obtained. Previously, without recoding, binary counters produced an incorrect result. By using a scheme that allows the carry-out to be in the correct notation (i.e. BCD-4211 code) the result is produced correctly.

Recoding from BCD-4221 to BCD-5211 can be accomplished using the following Boolean logic expressions, derived from Table 10 [44]:

$$w[3] = h[3] \cdot (h[2] + h[1] + h[0]) + h[2] \cdot h[1] \cdot h[0]$$

$$w[2] = h[2] \cdot h[1] \cdot \overline{(h3 \oplus h[0])} + (h[3] \cdot \overline{h[0]}) \oplus h[2] \oplus h[1]$$

$$w[1] = h[2] \cdot h[1] \cdot \overline{(h[3] \oplus h[0])} + h[3] \cdot \overline{h[0]} \cdot \overline{(h[2] \oplus h[1])}$$

$$w[0] = (h[2] \cdot h[1]) \oplus h[3] \oplus h[0],$$

where $w_3w_2w_1w_0$ represent the BCD-5211 digit and $h_3h_2h_1h_0$ the digit in BCD-4221. This logic is implemented in the block that performs the multiplication of the carry vector by two ("x2" in Figure 32).

5.2 Delay analysis of the decimal 3:2 counter by recoding

An insight to the performance and flow structure of the 3:2 decimal counter proposed in [44] (shown in Figure 32) can gained by utilizing the Δ delay model frequently used for binary arithmetic delay analysis. The use of Δ delays aid in considering design trade-

offs in a circuit. Consider the case of a regular 1-bit Full Adder or 3:2 counter. Typically a 3:2 counter can be implemented using nine logic gates as shown in the following figure:

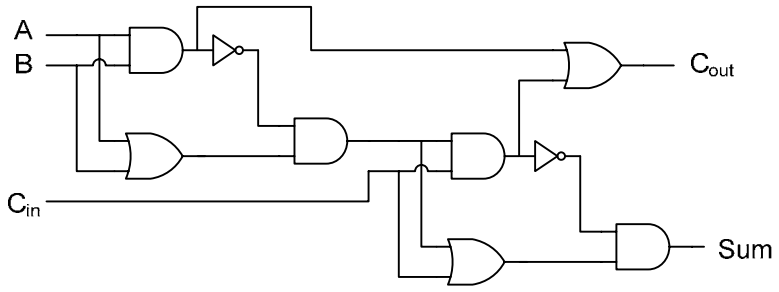


Figure 33. Nine gates Full Adder / 3:2 counter cell.

If the delay through a logic AND or OR gate is estimated to be of 2Δ delays and the delay through an inverter gate of 1Δ then we have the delay of the paths, from inputs to outputs:

$$A, B \rightarrow Sum = 10\Delta$$
$$A, B \rightarrow C_{out} = 9\Delta$$
$$C_{in} \rightarrow Sum = 5\Delta$$
$$C_{in} \rightarrow C_{out} = 4\Delta$$

i.e. it takes 10Δ delays for the signal to propagate from *A* or B to the *Sum* output for example. In the same way, the delay through the "x2" block can be obtained by looking at the logic equations used to implement it. This block consists of a recoding from BCD-4221 to BCD-5211 and a single bit hardwired left shift. The left shift does not incur a delay while the BCD recoding can be implemented using simple two level logic. Its delay therefore consists of 2 gate delays and an inverter delay used to create the logic predicates, resulting in 5Δ. Consequently the delay of the complete decimal 3:2 counter

of Figure 32, and considering the delay model for the binary 3:2 counter shown above, it requires:

$$A, B \rightarrow S = 10\Delta$$

$$A, B \rightarrow W = 9\Delta + 5\Delta = 14\Delta$$

$C \rightarrow S = 5\Delta$, through carry-in path,

$C \rightarrow W = 4\Delta + 5\Delta = 9\Delta$, through carry-in path.

This analysis can be used to evaluate the feasibility of the proposed partial product reduction scheme described in the next section, using decimal 4:2 compressors.

5.3 Decimal 4:2 Compressor Trees

Decimal partial product reduction trees that use the BCD recoding technique are different than their binary counterpart in the way the weights are handled. Binary partial product reduction trees always take the carry and pass it onto the next column as shown in Figure 14. This occurs because the carry-out is at a more significant weight than the sum. However, using the coding discussed in the previous section, it is apparent that for the decimal case this is completely different as it produces a carry-out digit at the same weight as the sum and a carry-out bit to the next significant digit.

The use 3:2 counters for partial product reduction is common in binary arithmetic. In [44], partial product reduction is handled by using a tree of decimal 3:2 counters, in a similar fashion to a Dadda tree [46]. This is shown in Figure 34 where the X2 block represents the recoding logic discussed earlier.

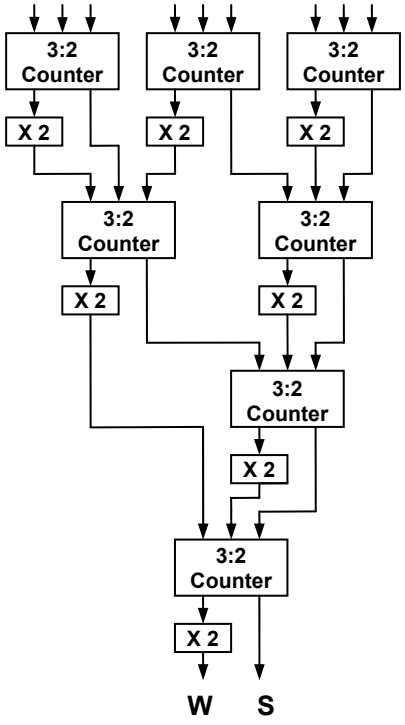


Figure 34. Decimal 9:2 counter, adapted from [44].

In [44], partial product reduction is handled by using a tree of decimal 3:2 counters, in a similar fashion to a Dadda tree [46]. This is shown in Figure 34 where the X2 block represents the recoding logic discussed earlier where a multiplication by 2 is obtained through BCD recoding. It is apparent however from Figure 34 that the resulting circuit is very irregular difficulting its implementation as its interconnection is complex and most likely affecting its performance and area consumption, in part due to unpredictable long wiring and its general layout.

Regularity is one of the reasons why in binary arithmetic the Weinberger 4:2 compressor [30] is used for partial product reduction. Its extension to the decimal case is therefore an interesting problem as it can provide performance enhancements for partial product reduction.

Apart from its improved regular structure, the proposed decimal 4:2 compressor has the advantage that the carry-in is no longer dependent on the carry-out, as shown in Figure 35, breaking the carry chain path. This gives compressors a significant advantage over traditional carry-save adder trees in that it can expedite processing the carry chain while still maintaining a regular structure. Although compressor trees are different than traditional counters, they can be organized into efficient interconnection networks for reducing the partial product matrix. This allows partial product reduction trees to be built easily and with regularity.

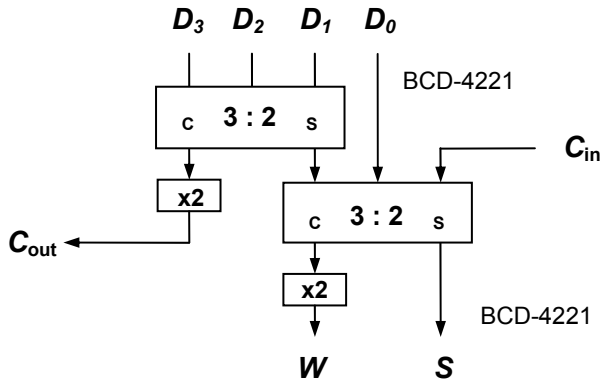


Figure 35. Proposed decimal 4:2 compressor. All signals are decimal (4-bits).

The new decimal compressor's structure is similar to the binary case except for the fact that in binary the signals C_{out} and *W* are both passed to the next column since their weight corresponds to the next significant bit, unlike *S*. In decimal however both C_{out} and *W* are composed of 2-digits, a "tens" digit and a "units" digit where a BCD number represents the "units" weight and a single bit the "tens" value. The "tens" single bit is generated in the X2 blocks during shifting as a carry-out and, at the same time, a carry-

in is accepted from the previous column. Figure 35 does not show the “tens” single bit generated at the X2 blocks for clarity (carry-out) or the single bit carry-in.

Utilizing the delay statements shown in Section 5.2 allows Δ delay analysis of the decimal compressor:

$$D_3, D_2, D_1, D_0 \rightarrow S = 10\Delta + 10\Delta = 20\Delta$$

$$D_3, D_2, D_1, D_0 \rightarrow W = 10\Delta + 14\Delta = 24\Delta$$

$$D_3, D_2, D_1, D_0 \rightarrow C_{out} = 14\Delta$$

$$C_{in} \rightarrow S = 5\Delta$$

$$C_{in} \rightarrow W = 4\Delta + 5\Delta = 9\Delta$$

This model is important since it determines the feasibility of higher order compressors that use this block since, as opposed to the binary case, the C_{out} and the W signals incur an additional delay due to the X2 block logic.

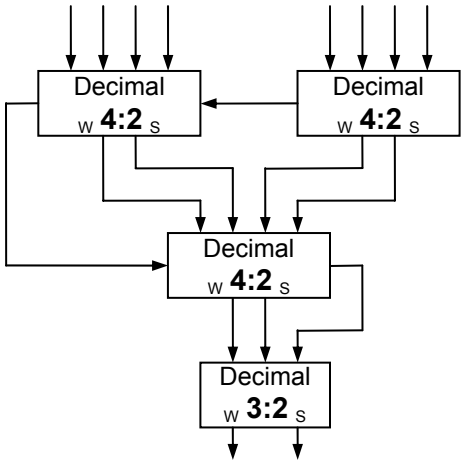


Figure 36. Decimal 8:2 compressor, all signals are decimal (4-bits).

Figure 36 shows an 8:2 compressor tree that performs the same function as Figure 34 yet is regular. Assuming that the inputs are given in time 0, and using the Δ delay expressions determined for the 4:2 compressor the Δ delay for each path can be determined. The time when the corresponding signal is available is shown in the following figure for the 8:2 compressor.

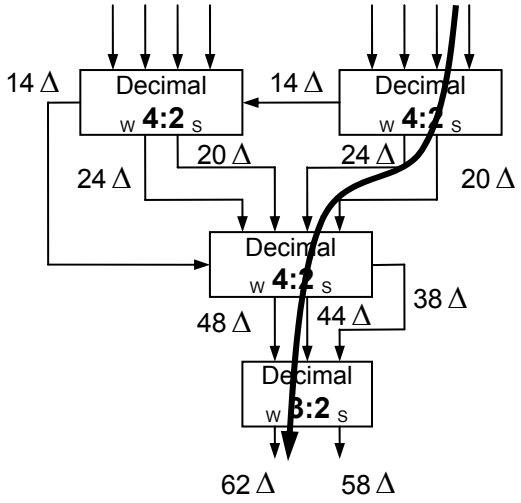


Figure 37. Decimal 8:2 compressor. Critical path and Δ delays for signals are shown.

The C_{out} signal only requires 14Δ as shown and is fast enough that it does not delay further the subsequent 4:2 compressor where its carry-out is attached. In other words, the delay is determined by the 24Δ alone required for the *W* signal in each compressor. If this was not the case, it will imply that the delay will accumulate with each stage affecting performance, even more significantly as the size of the compressor is increased (Ex: 16:2, 32:2, etc).

Subsequent column heights (digits) compressors can easily be built with $log_4(n)$ levels, where *n* represents the number of digits to be compressed (8 digits as shown, 16, etc).

Figure 38 illustrates a 16-digit (64-bit) decimal compressor tree using the same technique used to build the 8-digit tree from Figure 36.

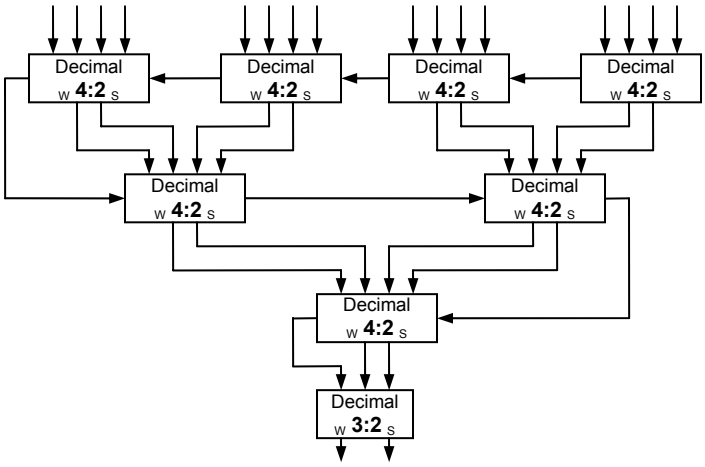


Figure 38. Decimal 16:2 compressor, all signals are decimal (4-bits).

6. PARTIAL PRODUCT GENERATION SCHEMES

Due to its complexity, decimal multiplication was previously implemented using sequential methods [31]. These designs generally required the iteration of each of the multiplier digits and, depending on its value, the addition of the corresponding multiple of the multiplicand to the result. This value has to be pre-computed and stored before the process takes place and is typically performed through costly lookup tables [12]. Enhancements to this approach included the reduction of the multiplier digit range through recoding to reduce the number of combinations, a method similar to Booth multiplication. Digit recoding allows the computation on the fly of multiplicand multiples to avoid their storage and the use of decimal carry save or sign-magnitude addition to speed the partial product reduction process [13], [14].

Currently there are only two propositions in the literature of decimal parallel multipliers, [44] and [20], with similar implementation architectures but significantly different multiplicand multiple generation schemes. Although these implementations are noteworthy, there are still many enhancements that require clarification which this section addresses.

6.1 Multiplier Recoding

As stated above typical decimal multiplication is more involved than binary requiring the need to generate the radix-10 multiplicand multiples 0*x*, 1*x*, 2*x*, …, 9*x*, as shown in Figure 39 (copied from Section 5 for clarity). Multiples like 3*x* and 7*x* are considered

"hard multiples", since their generation with binary logic is not straight forward and usually require a carry propagate addition which is slow. Previous methods for generating multiples utilized either this approach or costly look-up tables.

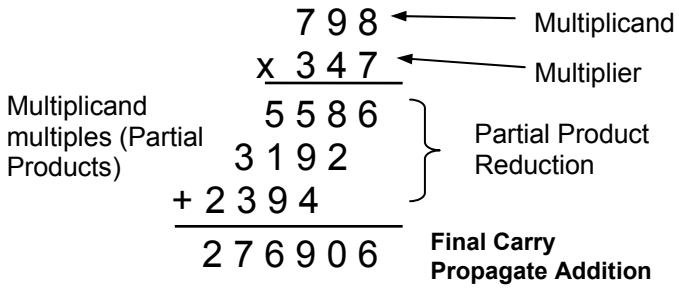


Figure 39. Multiplication Algorithm

The common multiplication algorithm is given by:

$$p = x \cdot y = \sum_{i=0}^{n-1} x \cdot y_i r^i \ ,$$

where *x* and *y* denote an *n* digit BCD multiplicand and multiplier, respectively, and *r* =10 denotes the radix and $y_i \in [0, 9]$. Recoding of the multiplier is an efficient technique for reducing its implementation since it permits a different set of multiples to be utilized avoiding the need for generating "hard multiples". One method recodes each multiplier digit $y_i \in [0, 9]$ to $y_i = y_{Hi} + y_{Li}$ where $y_{Hi} \in \{0, 5, 10\}$ and $y_{Li} \in$ {-2, -1, 0, 1, 2}. In this manner only the multiples 2*x* and 5*x*, which can be generated without excessive overhead, are required to create all other decimal multiples. This is illustrated in Table 11 in the radix-5 columns. Similarly, multiple 10*x* is produced with a simple 4-bit wired left shift (1-digit shift).

In [44], a radix-5 multiplier using this approach is described. A block diagram of this digit-recoding scheme is shown in Figure 40. Hot-one coded multiplexors are used avoiding the need of an extra '0*x*' input. The selection signals are determined directly from BCD-8421 input digits, y_i:

$$y_{Hi}10x = y_i[3]$$

$$y_{Hi}5x = y_i[2] + y_i[1] \cdot y_i[0]$$

$$y_{Li}2x = \overline{y_i[2]} \cdot y_i[1] + y_i[3] \cdot \overline{y_i[0]} + y_i[1] \cdot y_i[0]$$

$$y_{Li}1x = y_i[2] \cdot \overline{y_i[0]} + \overline{y_i[2]} \cdot \overline{y_i[1]} \cdot y_i[0]$$

$$sign = y_i[3] + y_i[2] \cdot \overline{y_i[1]} \cdot \overline{y_i[0]} + \overline{y_i[2]} \cdot y_i[1] \cdot y_i[0].$$

Table 11. Radix 5/4 digit recoding.

Decimal	Radix-5		Radix-4	
y_i	y_{Hi}	y_{Li}	y_{Hi}	y_{Li}
0	0	0	0	0
1	0	1	0	1
2	0	2	0	2
3	5	-2	4	-1
4	5	-1	4	0
5	5	0	4	1
6	5	1	4	2
7	5	2	8	-1
8	10	-2	8	0
9	10	-1	8	1

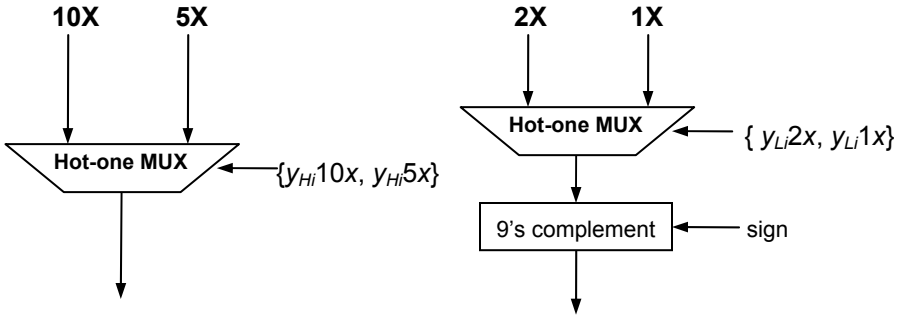


Figure 40. Digit recoding for radix-5, [44].

The second recoding approach, similar to the radix-5 case, recodes each multiplier digit $y_i \in [0, 9]$ to $y_i = y_{Hi} + y_{Li}$ where $y_{Hi} \in \{0, 4, 8\}$ and $y_{Li} \in$ {-2, -1, 0, 1, 2}. It is named radix-4 recoding in [19] and is also shown in Table 11. In this approach, the hard multiples are avoided and instead 2*x*, 4*x* and 8*x* are required to generate all others. The multiple 4*x* can be generated by cascading two consecutive 2*x* modules. An additional 2*x* module yields 8*x* implying a latency three times as large to that required to obtain 2*x*. The logic equations for the digit recoding of y_i are:

$$y_{Hi}8x = y_i[3] + (y_i[2] \cdot y_i[1] \cdot y_i[0])$$

$$y_{Hi}4x = y_i[2] \oplus (y_i[1] \cdot y_i[0])$$

$$y_{Li}2x = y_i[1] \cdot \overline{y_i[0]}$$

$$y_{Li}1x = y_i[0]$$

$$sign = y_i[1] \cdot y_i[0].$$

6.2 Multiplicand Multiples Generation

The recoding schemes discussed in the previous section avoid the computation of complicated multiples of *x*. Instead, in the case of radix-5, only 2*x* and 5*x* modules are required as all other multiples can be generated from them. Two main approaches for the generation of these multiples can be identified in available literature. The first method for obtaining 2*x* and 5*x* is studied in [31] with a conventional binary logic approach and utilized in [12] and [20]. A more recent technique proposed in [19] utilizes BCD recoding and shifting instead.

6.2.1 2x and 5x with Conventional Binary Logic

The multiples 2*x* and 5*x* can be produced rapidly as opposed to other multiples since in both doubling and quintupling of BCD-8421 numbers no carry is generated beyond the next significant digit. Logic equations for doubling and quintupling of BCDs are given in [12]. When doubling takes place, the Least Significant Bit (LSB) of each decimal BCD digit is initially zero. When the digit value is in the range of [5-9] a carry-out of at most one is produced (9 x 2 = 18). Therefore, it will not further propagate since the LSB of the next digit zero as well. The equations for doubling each multiplicand digit can be formed as follows:

$$2x_i[0] = (a_{i-1}[2] \cdot a_{i-1}[1] \cdot \overline{a_{i-1}[0]}) + (a_{i-1}[2] \cdot a_{i-1}[0]) + a_{i-1}[3])$$

$$2x_i[1] = (\overline{a_i[3]} \cdot \overline{a_i[2]} \cdot a_i[0]) + (a_i[2] \cdot a_i[1] \cdot \overline{a_i[0]}) + (a_i[3] \cdot \overline{a_i[0]})$$

$$2x_i[2] = (a_i[1] \cdot a_i[0]) + (\overline{a_i[2]} \cdot a_i[1]) + (a_i[3] \cdot \overline{a_i[0]})$$

$$2x_i[3] = (a_i[2] \cdot \overline{a_i[1]} \cdot \overline{a_i[0]}) + (a_i[3] \cdot a_i[0])$$

On the other hand, when a number is odd and quintupling takes place, its value becomes initially five and when the number is even it becomes zero. Quintupling produces a carry out of at most four (9 x 5 = 45). Since the initial value is zero or five, adding the carry results in at most 9, preventing the carry to propagate any further. Therefore, by checking the next significant digit LSB ($a_i[0]$) to check if the digit is 0 or 5, equations for 5*x* can be determined as follows:

$$5x_i[0] = (a_i[0] \cdot \overline{a_{i-1}[3]} \cdot \overline{a_{i-1}[1]}) + (\overline{a_i[0]} \cdot a_{i-1}[1]) + (a_i[0] \cdot a_{i-1}[3])$$

$$5x_i[1] = (\overline{a_i[0]} \cdot a_{i-1}[2]) + (a_i[0] \cdot \overline{a_{i-1}[2]} \cdot a_{i-1}[1]) + (a_{i-1}[2] \cdot \overline{a_{i-1}[1]})$$

$$5x_i[2] = (a_i[0] \cdot \overline{a_{i-1}[3]} \cdot \overline{a_{i-1}[1]}) + (a_i[0] \cdot \overline{a_{i-1}[2]} \cdot a_{i-1}[1]) + (a_i[0] \cdot \overline{a_{i-1}[2]} \cdot a_{i-1}[1]) + (\overline{a_i[0]} \cdot a_{i-1}[3])$$

$$5x_i[3] = (a_i[0] \cdot a_{i-1}[2] \cdot a_{i-1}[1]) + (a_i[0] \cdot a_{i-1}[3])$$

The generation of the remaining “hard multiples”, as described in the previous section and Table 11, requires the addition of negative 1*x* or 2*x* which can be implemented by obtaining the nine’s complement of the number. As stated in Section 5, the nine’s complement of a BCD-8421 digit can be determined from simple two-level logic. This allows the complete multiple digit set to be obtained.

6.2.2 2x and 5x using BCD recoding

In [19], doubling and quintupling are performed using BCD recoding. Doubling or multiplying by 2 is explained in Section 5.1 and Figure 30 and is implemented for multiple generation in the same way.

Quintupling is performed in a similar fashion in [19]. In this case, the digit starts in BCD-8421 code, as shown in Figure 41. A simple 3-bit hardwired left shift performs the multiplication assuming the result to be in BCD-5421 code.

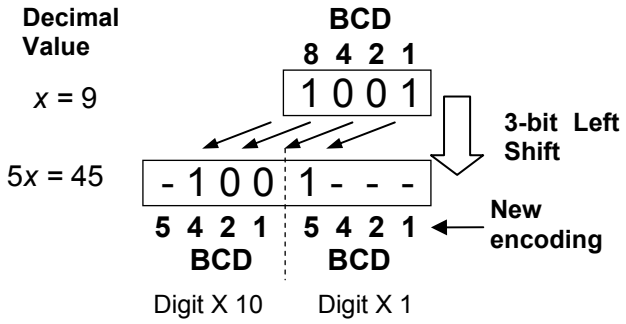


Figure 41. Quintupling through BCD recoding.

As opposed to BCD-8421 code, an advantage of using BCD-4221 is that, since the sum of the weights of all 4-bits is equal to 9, the representation is self-complementing as shown in Chapter 5. This means that the 9's complement of the number can be obtained by a simple bit inversion of each of the bits. In this manner, the ten's complement of the number for subtraction can be obtained in much the same way as the two's complement for binary numbers, a bit inversion followed by an addition of a unit or one in the least significant place (*ulp*) without requiring additional logic.

The BCD recoding approach is convenient as well as simple, since the partial product reduction technique presented in Section 5 receives the partial products in BCD-4221 code and hence no recoding back from BCD-4221 to BCD-8421 is required in the doubling scheme. One disadvantage, however, is that the conversion from BCD-8421 to 5211 for doubling and from BCD-5421 back to 4221 after quintupling is not straight forward and requires dedicated logic that incurs a slight overhead. Its latency, however, is comparable with the doubling and quintupling logic equations presented earlier in Section 6.2.1.

6.3 Partial Product Generation Architectures

Partial product generation schemes for parallel multipliers were proposed in [19] and [20]. In [20], a radix-5 multiplier is implemented using the binary techniques discussed in Section 6.2.1. An overview of this architecture is shown in

Figure 42. A BCD digit input recoder determines the appropriate multiple selection signals for the multiplexors and binary logic is used to implement the logic equations described for doubling and quintupling. Since the code used, BCD-8421, is not self-complementing a dedicated 9's complement unit is required to create the negative multiples. Complementing modules, containing the logic to perform the equations given at the beginning of Chapter 5, are hence required as shown in the diagram.

The design also includes a specialized radix-10 carry-save adder that is used to add both multiples to generate the corresponding partial product before the reduction tree. This procedure effectively halves the number of partial products and simplifies the partial product reduction tree. In the case of 16 digits multiplier operands the number of partial products would be reduced from 32 to 16. The equations for the radix-10 carry-save adder are detailed in Section 2.2.5. Nevertheless the cost in area and delay of performing this addition during partial product generation is significant and outweighs the benefits when utilizing a reduction scheme like the one proposed in Chapter 5. This is explored in Chapter 7 where implementations for different architectures possibilities are explored.

A similar architecture to Figure 42 is presented in [19] for the radix-5 case but instead BCD recoding is utilized for multiplicand doubling and quintupling. Additionally, hot-one logic for digit recoding and for the multiplexors is used, saving digital logic as shown in

Figure 43. The radix-4 case utilizes the same architecture, but instead the multiples 4*x* and 8*x* are created with consecutive 2X BCD recoding modules. In both cases, radix-4 and radix-5, two partial products per multiplier digit are produced (refer to Table 11). For the case of 16 digit operands, as is the case for decimal64 numbers in the IEEE-754R, this translates into 32 partial products requiring a 32 level partial product reduction tree.

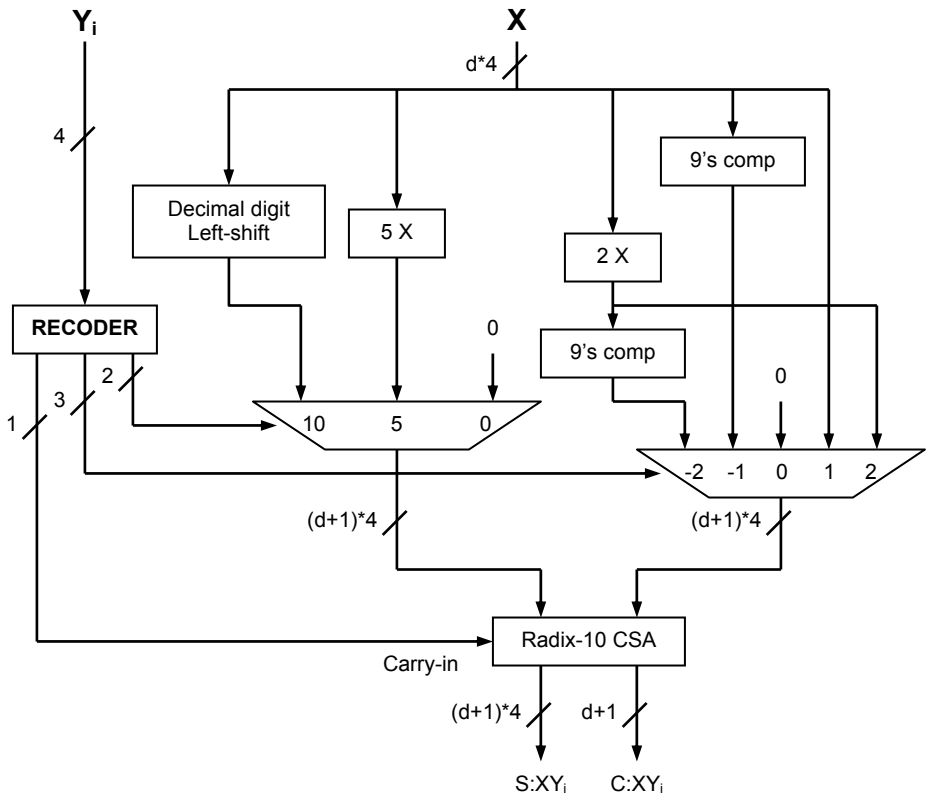


Figure 42. Lang-Nannarelli radix-10 partial product generation.

An additional enhancement is that since the BCD code utilized is self-complementing a dedicated 9's complement unit, like shown in Figure 42, is not necessary. A simple bit inversion performs this operation and can be activated on command with an XOR gate saving logic. The resulting architecture is fast and efficient; each multiplicand digit is

recoded to select the appropriate multiple and two (d+1) digits operands are produced as shown in Figure 43. For operands of 16 decimal digits in size the resulting number of partial products is 32.

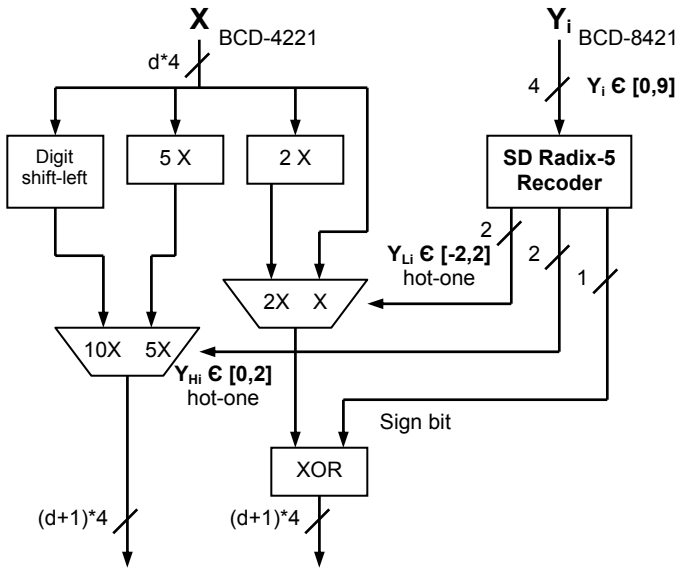


Figure 43. Vásquez-Antelo radix-5 partial product generation.

An additional insight into the performance of these architectures propositions can be obtained by combining the methods discussed previously for doubling and quintupling, specifically through direct Boolean logic or through BCD digit recoding. For this reason two hybrid versions were implemented: the radix-5 Vásquez-Antelo using binary logic for the 2x and 5x modules instead of the BCD recoding scheme and the Lang-Nannarelli architecture replacing the doubling and quintupling modules for their BCD recoding counterparts. The results of the original designs and these hybrid propositions were explored and shown in Chapter 7, Implementations.

7. Results

As technology improves and microchip minimum feature size is reduced more functionality can be included in a single chip. However, with CMOS designs moving to 90nm, 45nm and smaller technologies, a number of new physical effects have to be considered by the designer due to the short-channel nature of the transistor. This work utilized these technologies to portray the nature of decimal arithmetic and its impact upon different measures of performance. Consequently, the results will allow designers as well to choose the best suitable decimal units for nanometer-scale integration, rather than to extrapolate the results of research performed with older technologies. This chapter presents the results of the different implementations and comparisons of the proposed arithmetic hardware: the decimal floating-point comparator, the combined decimal/binary floating point comparator, the partial product generation architectures and the partial product reduction schemes for decimal multiplication.

One of the goals of this work was to create high-quality professional nanometer implementations. Therefore, having a well-defined design flow or sequence of steps within design tools that utilizes the maximum degree of design automation is of great importance. To help illustrate the designs within this dissertation, several commercially developed Electronic Design Automation (EDA) tools are used with VLSI design flows, including OSU's TSMC 180nm Standard Cell Library [47], Cadence Design Systems GPDK 90nm and NCSU/OSU FreePDK 45nm [48]. Many of this flows and libraries, such as the OSU 180nm and FreePDK 45nm are developed for use in this dissertation.

Specifically, the FreePDK 45nm is developed with Synopsys' Cadabra to create a full standard cell library with automatic transistor layout (ATL) to generate the library automatically. Figure 44 shows the design flow utilized for the arithmetic hardware presented in this work.

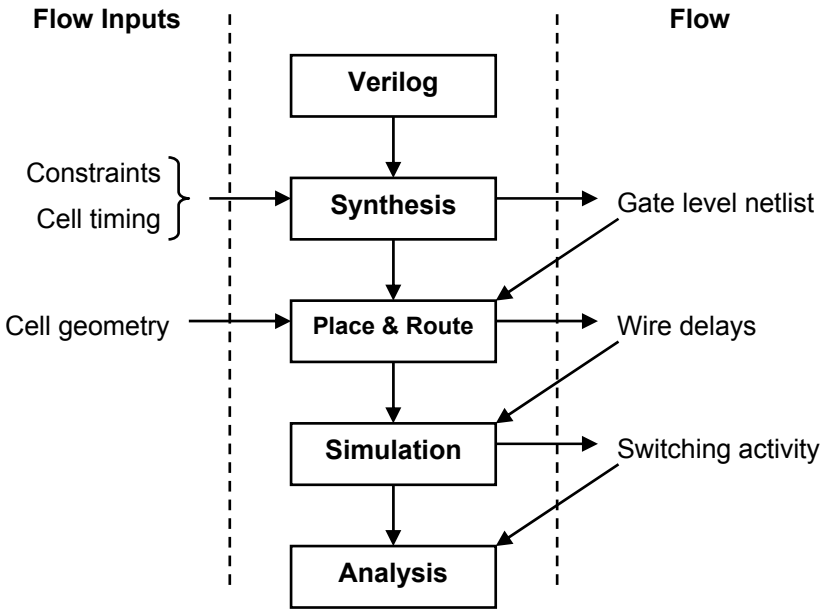


Figure 44. Design Flow methodology.

Each unit was first modeled using Verilog hardware descriptive language (HDL). Extensive testing of each functional unit was done to verify its correct operation using a copious number of test vectors in each design. Synthesis optimized for delay was performed using one of different tools: Cadence Design Systems Build Gates, RTL Compiler or Synopsys Design Compiler. The placement of cells and the routing of interconnect was performed with Cadence Encounter SoC which provides area, power and delay estimates using back-annotated layout.

7.1 Decimal comparator design

Since there are no other previous decimal floating-point comparators published, the delay and area results cannot be compared against other designs. An insight to how good the decimal comparatorproposed performs can be gained by comparing it to the only IEEE-754R [8] draft compliant decimal floating-point adder found, published in [16], even though the delay and area results for this design are given for a different implementation technology. Table 12 gives the post-synthesis area and delay results for both designs.

Table 12. Area and delay estimates

Design	Technology of implementation	Area (mm^2)	Delay (ns)
Decimal FP Comparator	TSMC 0.18um	0.070	2.73
Decimal FP Adder	LSI Logic 0.11um	0.148	3.83

Given that a common approach to comparison is to subtract both numbers and check the result's sign (if result is zero the numbers are equal, if result is negative then the first operand is less than the second) a dedicated floating-point comparator is only justifiable if its performance enhancement is considerable and its implementation does not incur in a significant area overhead.

The delay estimate presented is faster than that of the floating point adder and the area utilized is significantly smaller despite larger feature size of the implementation (0.18μm as opposed to 0.11μm). A rough estimate of scaling this design to 0.11μm technology by $1/S^2$ yields a ~5 times smaller area requirement than the adder and about twice the speed with a factor of 1/S. The delay and area estimates after place and routing were 4.328ns and 0.073mm^2 respectively.

The performance difference can also be in part attributed to the adder's necessity for a precise rounded result as opposed to the comparator. On the other hand, a decimal floating-point comparator would generally only be added to an architecture with decimal floating-point hardware. The presented implementation included decimal floating-point decoding logic and hence this portion would be removed making its area impact even smaller than the results shown.

7.2 Decimal/Binary Combined Comparator

Again, just as in the decimal floating-point comparator from the previous section an important step to validate the proposed combined comparator is to weigh it against the common approach method for comparisons in which an adder/subtractor scheme is used. For this purpose a combined comparator was implemented using a high speed 64-bit Carry Look-Ahead adder (CLA) to compare the operands' magnitudes instead of the proposed logarithmic tree described in Chapter 3. This is illustrated in Figure 45.

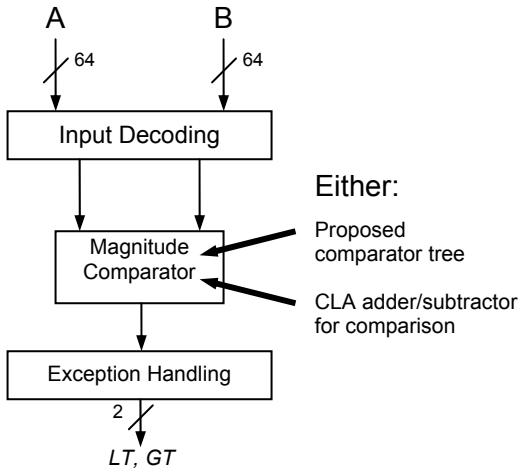


Figure 45. Concept block diagram for implementation comparison.

Synthesis was performed using Synopsys Design Compiler and the Carry Look-Ahead adder, DW01_add1, was implemented using Synopsys' DesignWare Intellectual Property library [50]. Previous designs, the combined two's complement and binary floating-point comparator given in [26] and the decimal floating-point comparator presented in this work in chapter 3 and implemented in section 7.1, were also synthesized and Place and Routed under the same conditions and included for comparison purposes in terms of area, delay and power consumption. Dynamic power figures were obtained using Encounter's power analysis through simulation utilizing a set of 10000 random generated input test vectors for each device under test. Figure 46-48 illustrate the results.

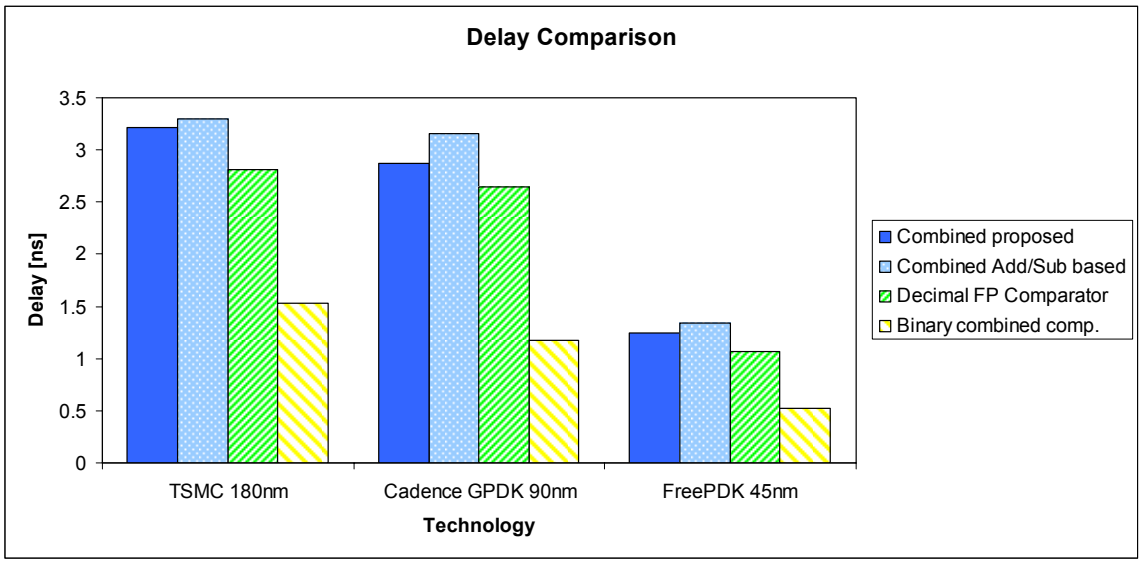


Figure 46. Delay estimates for comparator designs.

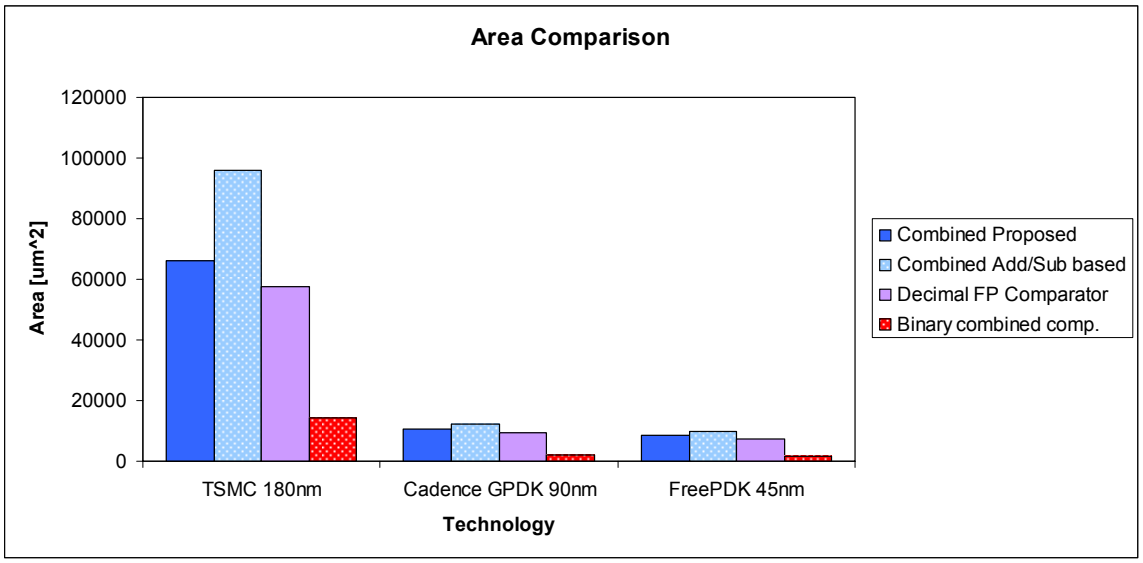


Figure 47. Area estimates for comparator designs.

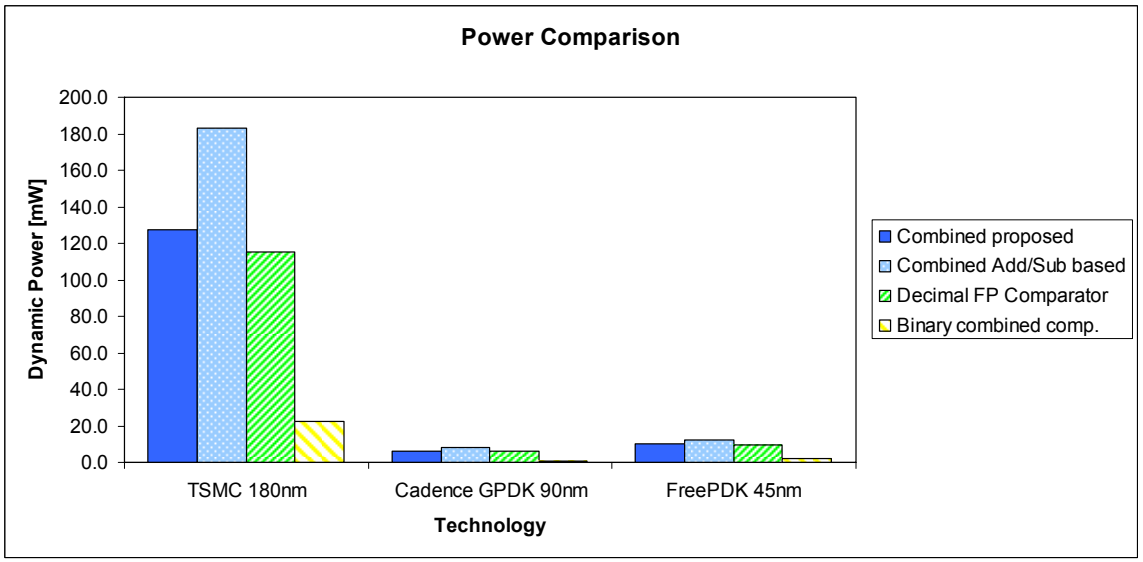


Figure 48. Dynamic power consumption for comparator designs.

As expected, the performance figures are generally improved as the technology utilized progresses, from 180nm to 45nm.

The results suggest an efficient method for comparisons with a small area requirement, improved speed and significant power reduction when weighed against comparators based on the addition/subtraction hardware approach. The design is also comparable in performance with the decimal only design making it an attractive solution since it can handle binary, two's complement and binary and decimal floating-point numbers.

7.3 Partial Product Reduction Schemes

In [19], the results presented for the partial product reduction using 9:2, 16:2 and 32:2 counter trees were determined using an area-delay model for CMOS based on logical effort [51]. Delay is given in terms of FO4 units (delay of a 1x sized inverter driving four 1x inverters) and area in minimum area NAND gate units. This method however is inaccurate and does not present the results of a real implementation. Furthermore, it makes comparison of their design against new proposals difficult. This is the reason why the Váquez - Antelo - Montuschi counter tress in [19], or VAM trees, were implemented in Verilog and went through the same design flow process as the proposed compressor trees to obtain accurate comparison results between the two architectures.

Figure 49 through 51 document the results for the proposed 8:2, 16:2 and 32:2 decimal compressor trees compared to the counter based trees in terms of area, delay and power. For small operand size inputs, the delay figures for both counter and compressor style reduction trees are similar and do not provide a substantial benefit. As the operand size increases, however, the compressor outperforms comparable counter trees in delay but sometimes gains a small area overhead in comparison. It is important however to

notice that the design proposed, shown in Figure 36 and Figure 38, is highly regular when compared to the counter tree in Figure 34. This facilitates its implementation and improves its performance but this characteristic can mainly be exploited through a custom-layout implementation. Nevertheless a standard cell layout implementation as developed provides an insight into the compressor trees performance although the benefits of regularity are not fully exploited.

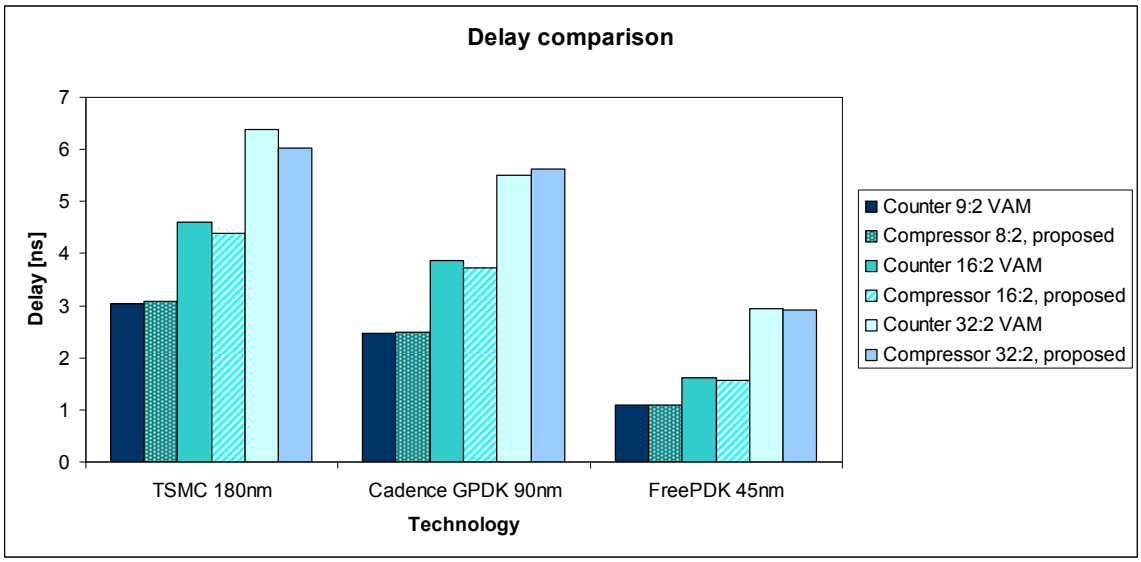


Figure 49. Delay estimates for compressor trees vs. counter trees designs.

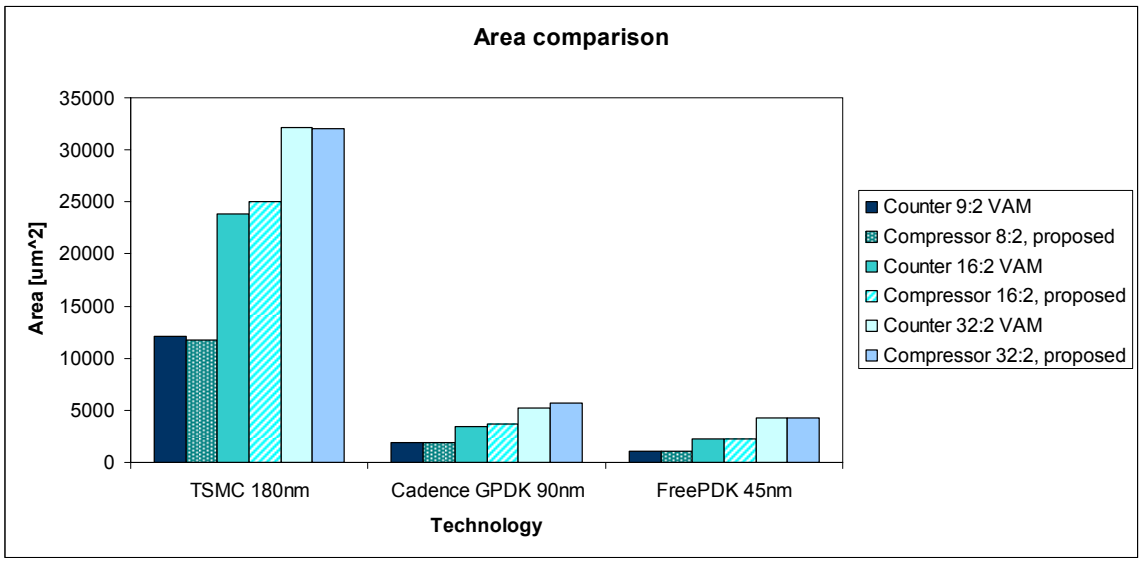


Figure 50. Area estimates for compressor trees vs. counter trees designs.

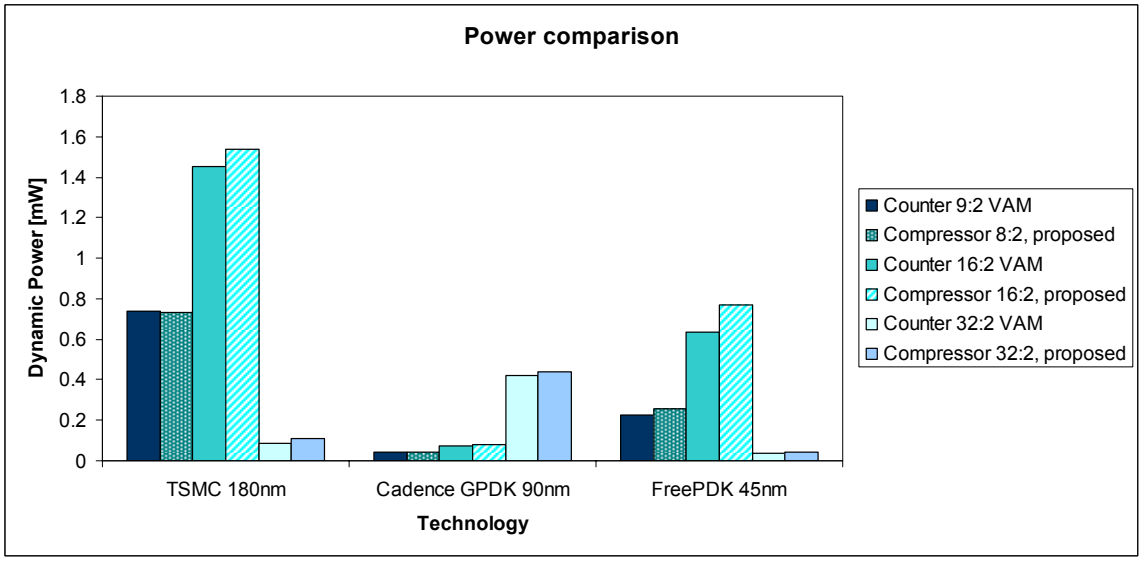


Figure 51. Dynamic power estimates for compressor trees vs. counter trees designs.

7.4 Partial Product Generation Architectures

The radix-5 and radix-4 Vásquez-Antelo-Montuschi (VAM) architectures proposed in [19], and shown in Figure 43, were implemented and its results given in the first two entries of Figure 52 through 54. The implementations are based on a 16-digit operand size, corresponding to decimal64 numbers (64-bit decimal floating-point). In the results presented, each partial product generation scheme handles a single digit of the multiplier operand, y_i, (see Figure 43) times the 16 digits of the multiplicand, X in this case. For a complete 16 digit multiplier, the partial product generation architecture would be replicated 16 times to scan over all 16 digits of the multiplier operand y_i, with $i \in \{0, 15\}$.

The proposed radix-5 design from Lang-Nannarelli in [20] (LN Radix-5) shows a significant delay overhead when compared to the first two implementations. This occurs mainly due to the radix-10 CSA adder included in the design, shown in Figure 42. To provide a fair comparison to the previous schemes this adder is removed and, hence, the partial product generator outputs 2 partial products per multiplier digit, in the same manner as the VAM generators. This is shown in the third and fourth entries in the tables where, without the radix-10 CSA the LN design presents attractive performance estimates.

As stated in section 6.3, two hybrid versions were implemented combining elements of both propositions: the radix-5 VAM using binary logic for the 2x and 5x modules instead of the BCD recoding and the LN radix-5 replacing the doubling and quintupling modules for their BCD recoding counterparts. Results for these two new designs are presented in the last two entries of Figure 52 through 54 and demonstrate and attractive alternative for decimal partial product generation.

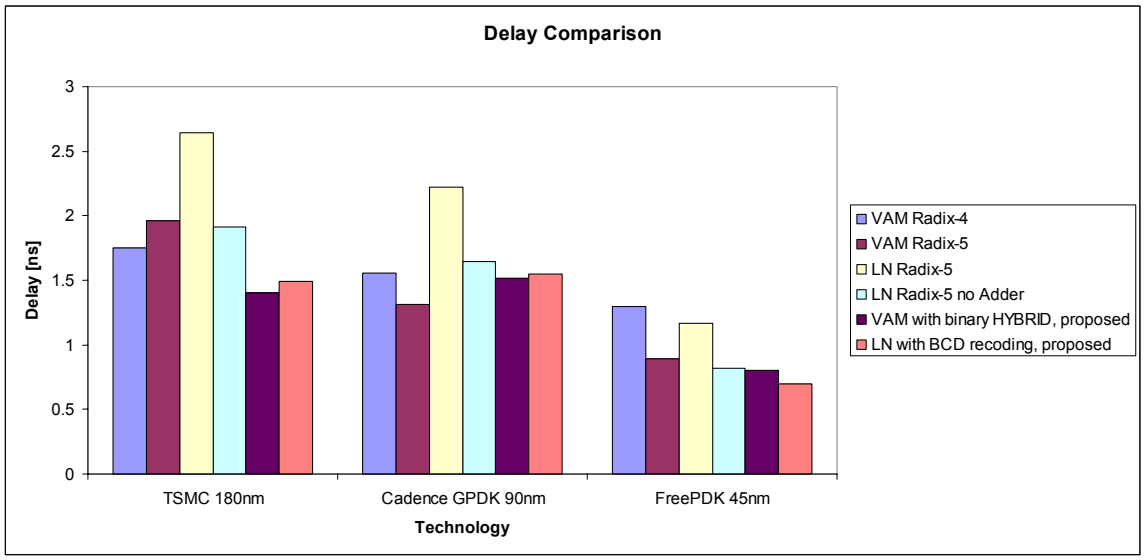


Figure 52. Delay results, partial product generation architectures.

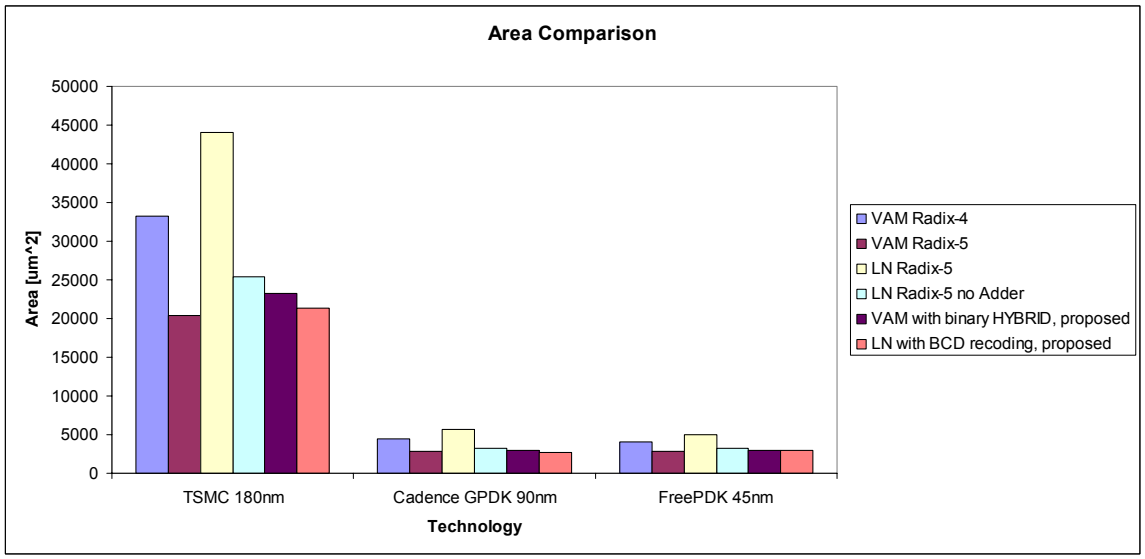


Figure 53. Area comparison for partial product generation architectures.

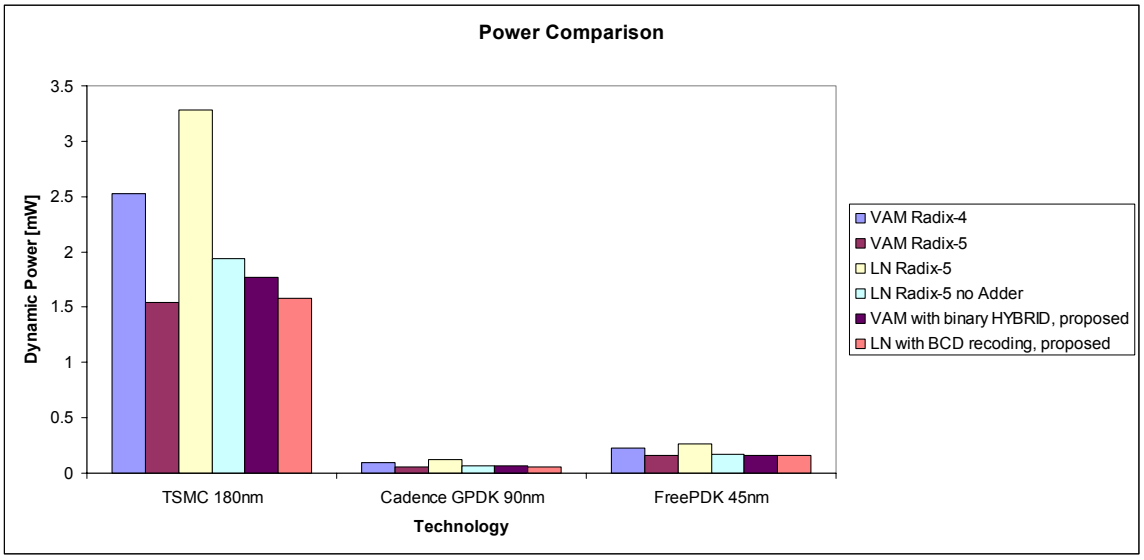


Figure 54. Dynamic power comparison, partial product generation.

8. CONCLUSIONS

The increasing importance of decimal arithmetic is evident from the current ongoing research in this field and the efforts to establish the IEEE-754R standard draft for floating-point arithmetic with specifications for the decimal case. Even more with IBM's new Power6 processor, the first of its kind to include decimal floating-point hardware, [52][53].

Multiple designs for decimal arithmetic were presented throughout this work. The comparator results suggest an efficient method for comparisons with a small area requirement, improved speed and significant power reduction when weighed against comparators based on the addition/subtraction hardware approach. The combined comparator, capable of handling decimal and binary floating-point numbers, two's complement numbers and binary numbers also presents performance improvements with a slight area overhead making it an attractive option for the comparison operation. Table 13 and Table 14 present the results.

Table 13. Area and delay results for comparator designs.

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]
Combined proposed	66033	3.21	10556	2.87	8525	1.25
Combined Adder/Subtractor based DW	95766	3.30	12271	3.16	9815	1.34
Decimal FP Comparator	57520	2.81	9253	2.65	7356	1.07
Binary, 2's, FP Comparator	14270	1.53	2020	1.17	1716	0.52

Table 14. Dynamic and Static power results for comparator designs.

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]
Combined proposed	127.8	0.12	6.36	42.44	10.27	33.83
Combined Adder/Subtractor based DW	182.8	0.18	7.86	52.64	12.43	42.25
Decimal FP Comparator	115.1	0.11	6.31	40.68	9.51	30.50
Binary, 2's, FP Comparator	22.3	0.03	1.01	8.21	1.71	6.42

An analytical study was developed for the design of a decimal division by recurrence functional unit. Utilizing previous formalizations of finding a minimal digit set and appropriate QDS unit, it was demonstrated that the design of such a unit is viable.

In decimal multiplication a previously proposed specialized recoding structure is studied and integrated within a compressor tree allowing decimal reduction to occur efficiently. Moreover, the decimal compressor trees proposed have structures which lend themselves better to custom-level VLSI designs. Results in a submicron standard cell library indicate high performance efficient designs. However, for low operand sizes, decimal compressor trees perform similarly than previously proposed decimal counter trees. Table 15 and Table 16 show the results summarized in Figure 49 to 51.

Table 15. Comparison results for proposed compressor trees vs. counter trees in [19].

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]
Counter 9:2 VAM	12059	3.03	1888	2.46	1046	1.10
Compressor 8:2, proposed	11729	3.09	1873	2.50	1089	1.10
Counter 16:2 VAM	23832	4.61	3465	3.86	2196	1.62
Compressor 16:2, proposed	25030	4.38	3686	3.73	2209	1.57
Counter 32:2 VAM	32189	6.39	5222	5.51	4297	2.95
Compressor 32:2, proposed	32007	6.03	5702	5.63	4228	2.91

An interesting degrading outcome however of technology scaling is the increasing importance that static power presents increasing by one or two orders of magnitude, as shown in Table 14. This is due to the important effects present in deep sub-micron

technologies that could be previously ignored like leakage current due to sub-threshold conduction, gate oxide tunneling and Gate Induced Drain Leakage (GIDL).

Table 16. Dynamic and static power comparison results for proposed compressor trees vs. counter trees in [19].

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]
Counter 9:2 VAM	0.738	0.024	0.045	11.9	0.223	6.2
Compressor 8:2, proposed	0.734	0.023	0.044	10.1	0.256	6.3
Counter 16:2 VAM	1.453	0.047	0.076	21.6	0.634	11.9
Compressor 16:2, proposed	1.540	0.050	0.079	22.3	0.767	12.3
Counter 32:2 VAM	0.085	0.064	0.424	25.1	0.038	21.6
Compressor 32:2, proposed	0.107	0.065	0.440	26.3	0.043	21.3

This work also presents comparisons for two recent approaches of decimal partial product generation in terms of delay, area and power consumption. Two additional hybrid architectures are developed and present attractive performance metrics versus previous designs. Table 17 and Table 18 present the results of these implementations, given in the last two entries, and contrast them with previous designs.

Table 17. Area and delay results for VAM [19], LN [20] architectures and hybrid partial product generation architectures.

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]	Area [um^2]	Delay [ns]
VAM Radix-4	33286	1.75	4482	1.56	4015	1.30
VAM Radix-5	20465	1.96	2813	1.31	2836	0.89
LN Radix-5	44108	2.64	5652	2.22	5047	1.17
LN Radix-5 no Adder	25453	1.91	3202	1.65	3178	0.82
VAM with binary HYBRID, proposed	23253	1.40	3019	1.52	2981	0.80
LN with BCD recoding, proposed	21400	1.49	2688	1.55	2972	0.70

Table 18. Dynamic and static power consumption for partial product generation architectures.

Technology	TSMC 180nm		Cadence GPDK 90nm		FreePDK 45nm	
Design	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]	Dynamic [mW]	Static [uW]
VAM Radix-4	2.525	0.071	0.091	21.8	0.224	19.7
VAM Radix-5	1.542	0.043	0.053	17.0	0.159	14.5
LN Radix-5	3.278	0.086	0.125	27.8	0.264	23.1
LN Radix-5 no Adder	1.941	0.049	0.067	15.3	0.168	13.6
VAM with binary HYBRID, proposed	1.766	0.044	0.063	17.0	0.159	12.9
LN with BCD recoding, proposed	1.582	0.043	0.056	11.6	0.162	14.1

Future work beyond the scope of this dissertation can be focused in different areas pertaining decimal arithmetic. One of these areas includes further research of complex multiple generation (2x, 5x, etc.) to improve partial product generation. Digit recoding can also be explored and the possibility of applying a Booth's style algorithm to reduce the number of partial products generated. Reduction and compressor trees can be further explored through the use of recoding, with emphasis on the applicability of binary techniques for Carry-Save addition to the decimal case. For division, on the other hand, an implementation and analysis of a quotient digit by recurrence divider utilizing the results of the study developed would produce further insight into its architecture. Another area that could be researched is the decimal operation Fused Multiply Add (FMA) or $f = A \times B + C$ utilizing the principles developed for partial product generation and reduction. Finally, the implementation of an architecture study to combine both decimal and binary arithmetic in one unit and assess the speedup and penalties from having such a capability.

BIBLIOGRAPHY

[1] Standards Committee of the IEEE Computer Society, IEEE Standard 754 for Binary Floating Point Arithmetic. IEEE Press, August 1985.

[2] M.F. Cowlishaw, “Decimal floating-point: algorism for computers.” Proceedings of the 16th IEEE Symposium on Computer Arithmetic, 2003.

[3] A. Tsang, M. Olschanowsky, “A study of dataBase 2 customer queries.” IBM Technical Report TR 03.413, IBM Santa Teresa Laboratory, San Jose, CA, April 1991.

[4] IBM Corporation, General Decimal Arithmetic, 2004. Available at: http://www2.hursley.ibm.com/decimal/

[5] European Commission, “The Introduction of the Euro and the Rounding of Currency Amounts,” Brussels, 1998.

[6] European Commission, “Council Regulation (EC) No 1103/97 of 17 June 1997 on certain provisions relating to the introduction of the euro,” Official Journal L 162, pp. 1-3, 1997.

[7] IBM Press Release: IBM Unleashes World's Fastest Chip in Powerful New Computer, May 21, 2007. Available at: http://www-03.ibm.com/press/us/en/pressrelease/21580.wss

[8] Standards Committee of the IEEE Computer Society, Draft Standard for Floating Point Arithmetic. IEEE, 2005. Available from: http://754r.ucbtest.org/drafts/754r.pdf.

[9] M.F. Cowlishaw, “Decimal Arithmetic Encoding,” Strawman 4d. Draft version 0.96. IBM UK laboratories, 2003.

[10] M.F. Cowlishaw, “Densely packed decimal encoding,” IEE Proceedings in Computers and Digital Techniques, pp.102-104, 2002.

[11] Semiconductor Industry Association, “International technology roadmap for semiconductors,” 2003.

[12] M.A. Erle, M.J. Schulte, “Decimal multiplication via carry-save addition,” Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors ASAP, 2003.

[13] R.D. Kenney, M.J. Schulte and M.A. Erle, “A high-frequency decimal multiplier,” Proceedings of the IEEE International Conference on Computer Design, 2004.

[14] M.A. Erle, E.M. Schwarz and M.J. Schulte, “Decimal multiplication with efficient partial product generation,” Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH), 2005.

[15] M.S. Schmookler, A.W. Weinberger, “High speed decimal addition,” IEEE Transactions on Computers, vol. C-20, pp. 862-867, Aug 1971.

[16] J. Thompson, N. Karra, M.J. Schulte, “A 64-bit decimal floating-point adder.” Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI System Design, pp. 297-298, 2004.

[17] A. Svoboda, “Decimal adder with signed digit arithmetic,” IEEE Transactions on Computers, March 1969.

[18] R.D. Kenney, M.J. Schulte, “High-speed multioperand decimal adders,” IEEE Transactions on Computers, August 2005.

[19] A. Vasquez, E. Antelo and P. Montuschi, “A new family of high-performance parallel decimal multipliers,” Proceedings of the 18th IEEE Symposium on Computer Arithmetic, 2006.

[20] T. Lang, A. Nannarelli, “A radix-10 combinational multiplier,” Proceedings of the 40th Asilomar Conference on Signals, Systems and computers, pp. 313-317, 2006.

[21] A.W. Burks, H.H. Goldstine, J. von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” Inst. For Advanced Study, Princeton, N. J., pp. 42, 1946.

[22] J. L. Hennessy, D.A. Patterson. Computer Architecture: a Quantitative Approach, Morgan-Kaufmann Publishers, 2002.

[23] K.W. Glass, “Digital comparator circuit,” U.S. Patent 5,260,680, February 13, 1992. D. Norris, “Comparator circuit,” U.S. Patent 5,534,844, April 3, 1995.

[24] F. Murabayashi, T. Hotta, S. Tanaka, T. Yamauchi, H. Yamada, T. Nakano, Y. Kobayashi, and T. Bandoh, “3.3 V BiCMOS techniques for a 120-MHz RISC microprocessor,” IEEE Journal of Solid-State Circuits, vol. 29, no. 3, pp. 298–302, 1994.

[25] E.S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, “A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor,” IEEE Journal of Solid-State Circuits, vol. 37, no. 11, pp. 1433–1440, 2002.

[26] J.E. Stine, M.J. Schulte, “A combined two’s complement and floating point comparator.” Proceedings of the IEEE International Symposium on Circuits and Systems, pp. 89- 92, May 2005.

[27] D.L. Weaver, T. Germond, The Sparc Architecture Manual, Version 9. Prentice Hall, inc., 2000.

[28] E. E. Swartzlander, Jr., Computer Arithmetic I. IEEE Press, 1990.

[29] M.D. Ercegovac, T. Lang. Digital Arithmetic, Morgan Kaufman, San Francisco, 2004, Chapter 8.4.

[30] A. Weinberger, "4:2 Carry-Save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, 1981.

[31] R.K. Richards. Arithmetic Operations in Digital Computers, D. Van Nostrand Company, Princeton, NJ, 1955.

[32] N.H. Weste, D. Harris. CMOS VLSI Design: A circuits and systems perspective, 3rd Edition. Addison Wesley, Boston, 2005, Chapter 10.9.

[33] C.S. Wallace, "A Suggestion for a Fast Multiplier." IEEE Transactions on Electronic Computers, February 1964.

[34] G.W. Bewick, Fast Multiplication: Algorithms and Implementation, PhD. Dissertation Stanford University, 1994.

[35] F. Y. Busaba, T. Slegel, S. Carlough, C. Krygowski, and J. G. Rell. The design of the fixed point unit for the z990 microprocessor. In Proc. ACM Great Lakes 14th Symposium on VLSI, pages 364–367, Apr. 2004.

[36] R. H. Larson. High-speed multiply using four input carrysave adder. IBM Tech. Disclosure Bulletin, 16(7):2053–2054, Dec. 1973.

[37] T. Ohtsuki. Apparatus for decimal multiplication. U.S. Patent No. 4,677,583, June 1987.

[38] T. Ueda. Decimal multiplying assembly and multiply module. US Patent No. 5379245, Jan. 1995.

[39] K.P. Acken, M.J. Irwin, R.M. Owens, "Power comparisons for barrel shifters." Proceedings of the International Symposium on Low Power Electronics and Design, pp. 209--212, 1996.

[40] P. Kornerup, "Revisiting SRT Quotient Digit Selection," Proceedings of the 16th IEEE Symposium on Computer Arithmetic, 2003.

[41] M.D. Ercegovac, T. Lang. Division and Square Root, Kluwer Academic Publishers, Massachusetts, 1994.

[42] N. Ohkubo and M. Suzuki. "A 4.4 ns CMOS 54x54–bit multiplier using pass-transistor multiplexer," IEEE Journal of Solid State Circuits, 30(3):251–256, Mar. 1995.

[43] E. Schwarz, R. M. Averill, and L. J. Sigal. “A radix-8 CMOS S/390 multiplier,” Proceedings of the IEEE 13th Symposium on Computer Arithmetic, pages 2–9, July 1997.

[44] A.D. Booth, “A signed binary multiplication technique,” Quarterly Journal of Mechanics and Applied Mathematics, 4(2):236–240, 1951.

[45] O. L. MacSorley. “High-Speed Arithmetic in Binary Computers.” Proceedings of the IRE, 49(1):67–91, Jan 1961.

[46] L. Dadda, “Some schemes for Parallel Multipliers,” Alta Frequenza, vol.34, pp. 349-356, 1965.

[47] J.E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Illiev, N. Jachimiec, “A Framework for High-Level Synthesis of System-on-Chip Designs,” Proceedings of 2005 IEEE International Conference on Microelectronic Systems Education, pp. 67-68, 2005.

[48] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, R. Jenkal, “FreePDK: An Open-Source Variation-Aware Design Kit,” Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education.

[49] E. L. Braun, Digital Computer Design, Logic Circuitry, Synthesis. New York, N.Y.: Academic Press, 1963.

[50] DesignWare Building Block IP Quick Reference Guide. Synopsys, Inc., June 2004.

[51] I. Sutherland, R. F. Sproull, D. Harris. Logical Effort: Designing Fast CMOS Circuits, Morgan Kaufmann, 1999.

[52] E. Schwarz, S. R. Carlough, “Power6 decimal divide,” Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors ASAP, 2007.

[53] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, E. Fluhr, “Design and Implementation of the POWER6 Microprocessor,” IEEE Journal of Solid-State Circuits, pg. 21-28, 2008.

[54] T.C. Chen, T. Ho, “Storage-efficient representation of decimal data,” CACM, 1975. 18, (1), pp. 49–52 (summarized at http://www2.hursley.ibm.com/decimal/chen-ho.html).

9. APPENDICES

9.1 APPENDIX A – DENSELY PACKED DECIMAL ENCODING

Densely Packed Decimal encoding or DPD is a lossless compression technique based on the Chen-Ho binary encoding of decimal numbers [54]. It provides an efficient method of encoding three BCD digits into 10-bits and back.

DPD encoding or compression of the three BCD digits depends on the size of each digit. Numbers less than or equal to 7 (3-bits) are considered small and large if greater than 7 (4-bits). In this manner, a specific mapping is used in each situation: when all digits are small, when left digit is small, when middle digit is large and so on. The following table illustrates DPD compression where the three BCD digits are ***abcd***, ***efgh*** and ***ijkm*** respectively and the encoded 10-bits DPD ***pqr stu v wx y***. The letter ***l*** is not used to avoid confusion with the letter ***i*** and 1.

Table 19. DPD Encoding / Compression, taken from [10].

aei	*pqr*	*stu*	*v*	*wx*	*y*	Comments
000	bcd	fgh	0	jk	m	All digits are small
001	bcd	fgh	1	00	m	Right digit is large (0-9 are unchanged)
010	bcd	jkh	1	01	m	Middle digit is large
100	jkd	fgh	1	10	m	Left digit is large
110	jkd	00h	1	11	m	Right digit is small (left and middle are large)
101	fgd	01h	1	11	m	Middle digit is small (left and right are large)
011	bcd	10h	1	11	m	Left digit is small (middle and right are large)
111	00d	11h	1	11	m	All digits are large; two bits are unused

DPD decoding or expansion on the other hand converts the DPD 10-bits back to three BCD digits. The digits ***vwxst*** are used as indicators and used with the mappings shown in the next table:

Table 20. DPD Decoding / Expansion, taken from [10].

vwxst	*abcd*	*efgh*	*ijkm*
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

For example, if the encoded bits *pqr stu v wx y* are *100 011 1 10 1* the indicator bits *vwxst* are *11001* and these are matched in the fourth row of the table. Consequently the three BCDs represented are given by the input bits mapped to output bits as: *1000 0011 0101*, or 835.

As another example, if the encoded bits *pqr stu v wx y* are *011 101 1 11 1* then *vwxst* is *11110* which matches row 7 in the table. The BCD digits represented are then: *0011 1001 1001*, or 399.

Boolean expressions can also be used instead of the direct mappings shown in the previous tables for encoding and decoding. For encoding/compression these results in:

$$
\begin{aligned}
p &= (b \cdot \bar{a}) + (j \cdot a \cdot \bar{i}) + (f \cdot a \cdot \bar{e} \cdot i) \\
q &= (c \cdot \bar{a}) + (k \cdot a \cdot \bar{i}) + (g \cdot a \cdot \bar{e} \cdot i) \\
r &= d \\
s &= (f \cdot \bar{e} \cdot (\overline{a \cdot i})) + (j \cdot (\bar{a} \cdot e \cdot \bar{i})) + (e \cdot i) \\
t &= (g \cdot \bar{e} \cdot (\overline{a \cdot i})) + (k \cdot (\bar{a} \cdot e \cdot \bar{i})) + (a \cdot i) \\
u &= h \\
v &= a + e + i \\
w &= a + (e \cdot \mathrm{i}) + (j \cdot \bar{e} \cdot \bar{i})
\end{aligned}
$$

$$x = e + (a \cdot \mathrm{i}) + (k \cdot \bar{a} \cdot \bar{i})$$
$$y = m$$

For decoding/expansion the Boolean expressions are:

$$a = (v \cdot w) \cdot (\bar{x} + \bar{s} + (s \cdot t))$$
$$b = p \cdot (\bar{v} + \bar{w} + (x \cdot s \cdot \bar{t}))$$
$$c = q \cdot (\bar{v} + \bar{w} + (x \cdot s \cdot \bar{t}))$$
$$d = r$$
$$e = v \cdot ((\bar{w} \cdot x) + (w \cdot x \cdot (s + \bar{t})))$$
$$f = (s \cdot (\bar{v} + (v \cdot \bar{x}))) + (p \cdot v \cdot w \cdot x \cdot \bar{s} \cdot t)$$
$$g = (t \cdot (\bar{v} + (v \cdot \bar{x}))) + (q \cdot v \cdot w \cdot x \cdot \bar{s} \cdot t)$$
$$h = u$$
$$i = v \cdot ((\bar{w} \cdot \bar{x}) + (w \cdot x \cdot (s + t)))$$
$$j = (w \cdot \bar{v}) + (s \cdot v \cdot \bar{w} \cdot x) + (p \cdot v \cdot w \cdot (\bar{x} + (\bar{s} \cdot \bar{t})))$$
$$k = (x \cdot \bar{v}) + (t \cdot v \cdot \bar{w} \cdot x) + (q \cdot v \cdot w \cdot (\bar{x} + (\bar{s} \cdot \bar{t})))$$
$$m = y$$

VITA

Ivan Dario Castellanos

Candidate for the Degree of

Doctor of Philosophy

Dissertation: ANALYSIS AND IMPLEMENTATION OF DECIMAL ARITHMETIC HARDWARE IN NANOMETER CMOS TECHNOLOGY

Major Field: Electrical Engineering

Biographical:

Education: Received the Bachelor of Science degree in Electrical Engineering from the University of Los Andes, Bogotá, Colombia in September 2001. Received the Master of Science degree in Electrical Engineering from the Illinois Institute of Technology, Chicago, Illinois in December 2004. Completed the requirements for the Doctor of Philosophy in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in July 2008.

Experience: Employed by Oklahoma State University as a Research Assistant at the VLSI Computer Architecture Research Group, 2005 to present. Internship with Cadence Design Systems, San Jose, CA during the year 2005. Employed by the Illinois Institute of Technology as a Research Assistant at the VLSI Computer Architecture and CAD Laboratory, 2003 to 2004.

Name: Ivan Dario Castellanos Date of Degree: July, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: ANALYSIS AND IMPLEMENTATION OF DECIMAL ARITHMETIC HARDWARE IN NANOMETER CMOS TECHNOLOGY

Pages in Study: 115 Candidate for the Degree of Doctor of Philosophy
Major Field: Electrical Engineering

Scope and Method of Study: In today's society, decimal arithmetic is growing considerably in importance given its relevance in financial and commercial applications. Decimal calculations on binary hardware significantly impact performance mainly because most systems utilize software to emulate decimal calculations. The introduction of dedicated decimal hardware on the other hand promises the ability to improve performance by two or three orders of magnitude. The founding blocks of binary arithmetic are studied and applied to the development of decimal arithmetic hardware. New findings are contrasted with existent implementations and validated through extensive simulation.

Findings and Conclusions: New architectures and a significant study of decimal arithmetic was developed and implemented. The architectures proposed include an IEEE-754 current revision draft compliant floating-point comparator, a study on decimal division, partial product reduction schemes using decimal compressor trees and a final implementation of a decimal multiplier using advanced techniques for partial product generation. The results of each hardware implementation in nanometer technologies are weighed against existent propositions and show improvements upon area, delay, and power.

ADVISER'S APPROVAL: JAMES E. STINE