INTERNATIONAL JOURNAL OF RESEARCH
In Advanced Engineering Technologies

Volume 5, Issue 2 DEC 2015

A New VLSI Architecture for Multiplication using Radix-10 Modified Booth Encoding Technique

MULUGU SANDHYA RANI[(1)],KONDRA GOPI[(2)],INALA RAGHAVA KRISHNA [(3)]
P.G. Scholar, Kshatriya College of Engg,Chepur[(1)]
Assistant Professor,Project Coordinator, Kshatriya College of Engg,Chepur [(2)]
Assistant Professor,Internal Guide, Kshatriya College of Engg,Chepur [(3)]

Abstract—We present the algorithm and architecture of a BCD parallel multiplier that exploits some properties of redundant BCD excess-3 code (XS-3), and the overloaded BCD representation(ODDS) codes to speed up its computation. In addition, new techniques are developed to reduce significantly the latency and area of previous representative high performance implementations. Partial products are generated in parallel using a signed-digit radix-10 recoding of the BCD multiplier with the digit set [-5, 5], and a set of positive multiplicand multiples (0X, 1×, 2×, 3×, 4×, 5×) coded in XS-3.A. The available redundancy allows a fast and simple generation of multiplicand multiples in a carry free way. Finally, the partial products can be recoded to the ODDS representation by just adding a constant factor into the partial product reduction tree. Since the ODDS uses a similar 4-bit binary encoding as non-redundant BCD, conventional binary VLSI circuit techniques, such as binary carry-save adders and compressor trees, can be adapted efficiently to perform decimal operations. To show the advantages of our architecture, we have synthesized a RTL model for 16×16-digit and 34×34-digit multiplications and performed a comparative survey of the previous most representative designs. We show that the proposed decimal multiplier has an area improvement roughly in the range 20-35 percent for similar target delays with respect to the fastest implementation.

Index Terms—Parallel multiplication, decimal hardware, overloaded BCD representation, redundant excess-3 code, redundant arithmetic.

1 INTRODUCTION

DECIMAL important in financial, commercial, and user-oriented fixed-point and floating-point formats are computing, where conversion and rounding errors that are inherent to floating-point binary representations cannot be tolerated [3]. The new IEEE 754-2008 Standard for Floating Point Arithmetic [15], which contains a format and specification for decimal floating-point (DFP) arithmetic [1],has encouraged a significant amount of research in decimal hardware [6], [9], [10], [28], [30].Furthermore, current IBM Power and z-System families of microprocessors [5], [8], [23], and the Fujitsu Sparc X microprocessor [26], oriented to servers and mainframes, already include fully IEEE 754-2008 compliant decimal floating-point units (DFPUs) for Decimal64 (16 precision digits) and Decimal128 (34 precision digits) formats.

Since area and power dissipation are critical design factors in state-of-the-art DFPUs, multiplication and division are performed iteratively by means of digit-by-digit algorithms [4], [5], and therefore they present low performance. Moreover, the aggressive cycle time of these processors puts an additional constraint on the use of parallel techniques [6], [19], [30] for reducing the latency of DFP multiplication in high-performance DFPUs. Thus, efficient algorithms for accelerating DFP multiplication should result in regular VLSI layouts that allow an aggressive pipelining. Hardware implementations normally use BCD instead of binary to manipulate decimal fixed-point operands and integer significant of DFP numbers for easy conversion between machine and user representations [21], [25]. BCD encodes a number X in decimal (non-redundant radix-10) format, with each decimal digit $X_i \epsilon[0,9]$ represented in a 4-bit binary number system. However, BCD is less efficient for encoding integers than binary, since codes 10 to 15 are unused. Moreover, the implementation of BCD arithmetic as more complications than binary, which lead to area an delay penalties in the resulting arithmetic units. A variety of redundant decimal formats and arithmetic's have been proposed to improve the performance of BCD multiplication. The BCD carry-save format [9]

represents a radix-10 operand using a BCD digit and a carry bit at each decimal position. It is intended for carry-free accumulation of BCD partial products using rows of BCD digit adders arranged in linear [9], [20] or tree-like configurations[19]. Decimal signed-digit (SD) representations [10], [14],[24], [27] rely on a redundant digit set {-a, … , 0, … a}, $5 \leq a \leq 9$, to allow decimal carry-free addition.BCD carry-save and signed-digit radix-10 arithmetic's offer improvements in performance with respect to non redundant BCD. However, the resultant VLSI implementations in current technologies of multi-operand adder trees may result in more irregular layouts than binary carry-save adders (CSA) and compressor trees. Some approaches rely on binary arithmetic to perform decimal multi operand addition and multiplication. In [6], a decimal multi operand adder is implemented using columns of binary compressors and subsequent binary-to-BCD conversions. Also, decimal multi operand addition can be improved using binary carry-save adders and decimal doublers if digits are not represented in BCD but in certain decimal codes, namely, 4221 and 5211. These 4-bit decimal codes satisfy that the sum of the weights of the bits is equal to 9, so that all the 16 4-bit combinations represent a decimal digit in [0,9] . These codes have been used to speed-up decimal multi operand addition and multiplication [29], [30], [31]. The additional redundancy available in the 4-bit encoding is used to speed-up BCD operations while retaining the same data path width. Furthermore, these codes are self-complementing, so that the 9's complement of a digit, required for negation, is easily obtained y bit-inversion of its 4-bit representation. A disadvantage of 4221 and 5211 codes, is the use of a non-redundant radix-10 digit set [0,9] as BCD. Thus, the redundancy is constrained to the digit bounds, so that complex decimal multiples, such as 3×, cannot be obtained in a carry-free way.

The overloaded BCD (or ODDS—overloaded decimal digit set) representation was proposed to improve decimal multi-operand addition [18], and sequential [17] and parallel[12], [13] decimal multiplications. In this code, each 4-bitbinary value represents a redundant radix-10 digit $X_i \in [0,15]$. The ODDS presents interesting properties for a fast and efficient hardware implementation of decimal arithmetic: (1) it is a redundant decimal representation so that it allows carry-free generation of both simple and complex decimal multiples (2×, 3×, 4×, 5×, 6×,...) and addition,(2) since digits are represented in the binary number system, digit operations can be performed with binary arithmetic, and (3) unlike BCD, there is no need to implement additional hardware to correct invalid 4-bit combinations. A disadvantage with respect to signed-digit and self-complementing codes, is a slightly more complex implement ation of 9's complement operation for negation of operands and subtraction.

In this work, we focus on the improvement of parallel decimal multiplication by exploiting the redundancy of two decimal representations: the ODDS and the redundant BCDexcess-3 (XS-3) representation, a self-complementing code with the digit set [3,12]. We use a minimally redundant digit set for the recoding of the BCD multiplier digits, the signed-digit radix-10 recoding [30], that is, the recoded signed digits are in the set {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}.For this digit set, the main issue is to perform the 3 multiple without long carry-propagation (note that ×2 and ×5 are easy multiples for decimal [30] and that ×4 is generated as two consecutive ×2 operations). We propose the use of a general redundant BCD arithmetic (that includes the ODDS, XS-3 and BCD representations) to accelerate parallel BCD multiplication in two ways:

- Partial product generation (PPG). By generating positive multiplicand multiples coded in XS-3 in a carry free form. An advantage of the XS-3 representation over non-redundant decimal codes (BCD and 4221/5211 [30]) is that all the interesting multiples for decimal partial product generation, including the 3× multiple, can be implemented in constant time with an equivalent delay of about three XOR gate levels. Moreover, since XS-3 is a self-complementing code, the 9's complement of a positive multiple can be obtained by just inverting its bits as in binary. Partial product reduction (PPR). By performing the reduction of partial products coded in ODDS

via binary carry-save arithmetic. Partial products can be recoded from the XS-3 representation to the ODDS representation by just adding a constant factor into the partial product reduction tree. The resultant partial product reduction tree is implemented using regular structures of binary carry-save adders or compressors. The 4-bit binary encoding of ODDS operands allows a more efficient mapping of decimal algorithms into binary techniques. By contrast, signed-digit radix-10 and BCD carry-save redundant representations require specific radix-10 digit adders .

2. REDUNDANT BCD REPRESENTATIONS

The proposed decimal multiplier uses internally a redundant BCD arithmetic to speed up and simplify the implementation. This arithmetic deals with radix-10 ten's complement integers of the form:

$$Z = -s_z \times 10^d + \sum_{i=0}^{d-1} Z_i \times 10^i, \quad (1)$$

where d is the number of digits, sz is the sign bit, and $zi \in [l-e, m-e]$ is the ith digit, with

$$0 \le l \le e,\ 9 + e \le m \le 2^4 - 1 (= 15).$$

Parameter e is the excess of the representation and usually takes values 0 (non excess), 3 or 6. The redundancy index ρ defined as $\rho = m - l + 1 - r$ [12], being $r = 10$.

The value of Z_i depends on the decimal representation parameterized by (l, m, e). We use a 4-bit encoding to represent digits Z_i. This allows us to manage decimal operands in different representations with the same arithmetic, such as BCD ($Z_i \epsilon [0,9], e = 0, l = 0, m = 9, \rho = 0$), BCD excess-3 ($Z_i \epsilon [0,9], e = 3, l = 3, m = 12, \rho = 0$).

BCD excess-6 $Z_i \epsilon [0,9], e = 6, l = 6, m = 15, \rho = 0$) and redundant representations (ρ > 0), such as the ODDS representation ($Z_i \epsilon [0,15], e = 0, l = 0, m = 15, \rho = 6$), or the XS-3 representation ($Z_i \epsilon [-3,12], e = 3, l = 0, m = 15, \rho = 6$) On the other hand, the binary value of the 4-bit vector representation of Z_i is given by

$$[Z_i] = \sum_{j=0}^{3} z_{i,j} \times 2^j, \quad (2)$$

$Z_{i,j}$ being the jth bit of the ith digit. Therefore, the value of digit Z_i can be obtained by subtracting the excess e of the representation from the binary value of its 4-bit encoding, that is,

$$Z_i = [Z_i] - e.$$

Note that bit-weighted codes such as BCD and ODDS use the 4-bit binary encoding (or BCD encoding) defined in Expression (2). Thus, $Z_i = [Z_i]$ for operands Z represented in BCD or ODDS.

This binary encoding simplifies the hardware implementation of decimal arithmetic units, since we can make use of state-of-the-art binary logic and binary arithmetic techniques to implement digit operations. In particular, the ODDS representation presents interesting properties (redundancy and binary encoding of its digit set) for a fast and efficient implementation of multi-operand addition. Moreover, conversions from BCD to the ODDS representation are straightforward, since the digit set of BCD is a subset of the ODDS representation.

In our work we use a SD radix-10 recoding of the BCD multiplier [30], which requires to compute a set of decimal multiples ({-5×,…..,0×,…..,5×}) of the BCD multiplicand. The main issue is to perform the 3 multiple without long carry propagation.

For input digits of the multiplicand in conventional BCD (i.e., in the range [0, 9], e = 0, ρ = 0), the multiplication by 3 leads to a maximum decimal carry to the next position of 2and to a maximum value of the interim digit (the result digit before adding the carry from the lower position) of 9. Therefore the resultant maximum digit (after adding the decimal carry and the interim digit) is 11. Thus, the range of the digits after the ×3 multiplication is in the range [0, 11]. Therefore the redundant BCD representations can host the resultant digits with just one decimal carry propagation. An important issue for this representation is the ten's complement

operation. Since after the recoding of the multiplier digits, negative multiplication digits may result, it is necessary to negate (ten's complement) the multiplicand to obtain the negative partial products. This operation is usually done by computing the nine's complement of the multiplicand and adding a one in the proper place on the digit array. The nine's complement of a positive decimal operand is given by

$$-10^d + \sum_{i=0}^{d-1}(9 - Z_i) \times 10^i. \quad (3)$$

The implementation of $(9\text{-}Z_i)$ leads to a complex implementation, since the Z_i digits of the multiples generated may take values higher than 9. A simple implementation is obtained by observing that the excess-3 of the nine's complement of an operand is equal to the bit-complement of the operand coded in excess-3.

TABLE 1

Nine's complement for the XS-3 Representation

Digit			Nine's Complement		
4-bit Encoding	Z_i	$[Z_i]$	4-bit Encoding	$9 - Z_i$	$[9 - Z_i]$ $(=15 - [Z_i])$
0000	-3	0	1111	12	15
0001	-2	1	1110	11	14
0010	-1	2	1101	10	13
0011	0	3	1100	9	12
0100	1	4	1011	8	11
0101	2	5	1010	7	10
0110	3	6	1001	6	9
0111	4	7	1000	5	8
1000	5	8	0111	4	7
1001	6	9	0110	3	6
1010	7	10	0101	2	5
1011	8	11	0100	1	4
1100	9	12	0011	0	3
1101	10	13	0010	-1	2
1110	11	14	0001	-2	1
1111	12	15	0000	-3	0

In Table 1 we show how the nine's complement can be performed by simply inverting the bits of a digit Z_i coded in XS-3. At the decimal digit level, this is due to the fact that:

$$(9 - Z_i) + 3 = 15 - (Z_i + 3) \quad (4)$$

For the range $Z_i \epsilon [-3,12]$ $([Z_i] \epsilon [0,15])$. The fore to have a simple negation for partial product generation we produce the decimal multiples in an excess-3 code. The negation is performed by simple bit inversion, that corresponds to the excess-3 of the nine's complement of the multiple. Moreover, to simplify the implementation we combine the multiple generation stage and the digit increment by 3(to produce the excess-3) into a single module by using theXS-3 code (more details in Section 4.1). In summary, the main reasons for using the redundantXS-3 code are: (1) to avoid long carry-propagations in the generation of decimal positive multiplicand multiples, (2)to obtain the negative multiples from the corresponding positive ones easily, (3) simple conversion of the partial products generated in XS-3 to the ODDS representation for efficient partial product reduction (more details in Section 4.3).

3 HIGH-LEVEL ARCHITECTURE

The high-level block diagram of the proposed parallel architecture for d×d-digit BCD decimal integer and fixed-point multiplication is shown in Fig. 1. This architecture accepts conventional (non-redundant) BCD inputs X, Y, generates redundant BCD partial products PP, and computes the BCD product P =X×Y. It consists of the following threestages1: (1) parallel generation of partial products coded inXS-3, including generation of multiplicand multiples and recoding of the multiplier operand, (2) recoding of partial products from XS-3 to the ODDS representation and subsequent reduction, and (3) final conversion to a non-redundant 2d-digit BCD product.

Stage 1) Decimal Partial Product Generation.

A SD radix-10 recoding of the BCD multiplier has been used. This recoding produces a reduced number of partial products that lead to a significant reduction in overall multiplier area [29].

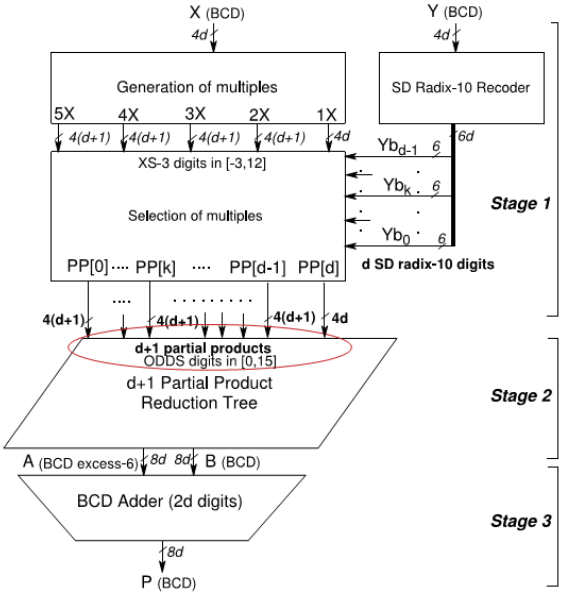


Fig. 1. Combinational SD radix-10 architecture

Therefore, the recoding of the d-digit multiplier Y into SD radix-10 digits $Yb_{d-1},\ldots,Yb_0$, produces d partial products PP[d-1],…. PP[0] , one per digit; note that each Yb_k recoded digit is represented in a 6–bit hot-one code to be used as control input of the multiplexers for selecting the proper multiplicand multiple, {-5X,…,-1X,0X,1X,….,5X}. An additional partial product PP [d] is produced by the most significant multiplier digit after the recoding , so that the total number of partial products generated is d +1.

Stage 2) Decimal partial product reduction.

In this stage, the array of $d + 1$ ODDS partial products are reduced to two 2d-digit words (A, B). Our proposal relies on a binary carry save adder tree to perform carry-free additions of the decimal partial products. The array of $d + 1$ ODDS partial products can be viewed as adjacent digit columns of height $h \leq d + 1$. Since ODDS digits are encoded in binary, the rules for binary arithmetic apply within the digit bounds, and only carries generated between radix-10 digits (4-bitcolumns) contribute to the decimal correction of the binary sum. That is, if a carry out is produced as a result of a 4-bit (modulo 16) binary addition, the binary sum must be incremented by 6 at the appropriate position to obtain the correct decimal sum (modulo 10 addition).

Two previous designs [12], [18] implement tree structures for the addition of ODDS operands. In the non speculative BCD adder [18], a combinational logic block is used to determine the sum correction after all the operands have been added in a binary CSA tree, with the maximum number of inputs limited to 19 BCD operands. By contrast, in our method the sum correction is evaluated concurrently with the binary carry-save additions using columns of binary counters. Basically we count the number of carries per decimal column and then a multiplication by 6 is performed (a correction by 6 for each carry-out from each column). The result is added as a correction term to the output of the binary carry-save reduction tree. This improves significantly the latency of the partial product reduction tree. Moreover, the proposed architecture accepts an arbitrary number of ODDS or BCD operand inputs. Some of PPR tree structures presented in [12] (the area-improved PPR tree)also exploit a similar idea, but rely on a custom designed ODDS adder to perform some of the stage reductions. Our proposal aims to provide an optimal reuse of any binary CSA tree for multi operand decimal addition, as it was done in [31] for the 4221 and 5211 decimal codings.

Stage 3) Conversion to (non-redundant) BCD.

We consider the use of a BCD carry-propagate adder [29] to perform the final conversion to a non-redundant BCD product $P = A + B$. The proposed architecture is a 2d-digit hybrid parallel prefix/carry-select adder, the BCD Quaternary Tree adder. The sum of input digits Ai, Bi at each position i has to be in the range [0,18]. so that at most one decimal carry is propagated to the next position $i + 1$ [22]. Furthermore, to generate the correct decimal carry, the BCD addition algorithm implemented requires $A_i + B_i$ to be obtained in excess-6. Several choices are possible. We opt for representing operand A in BCD excess-6 $(A_i \epsilon [0,9], [A_i] = A_i + e, e = 6)$, and B coded in BCD$(B_i \epsilon [0,9], e = 0)$.

4 DECIMAL PARTIAL PRODUCT GENERATION

The partial product generation stage comprises the recoding of the multiplier to a SD radix-10 representation, the calculation of the multiplicand multiples in XS-3 code and the generation of the ODDS partial products.

The SD radix-10 encoding produces d SD radix-10 digits $Yb_k \in [-5,5]$, with k=0,….,d-1,Y_{d-1} being the most significant digit (MSD) of the multiplier [29]. Each digit Ybk is represented with a 5-bit hot-one code ($Y1_k$, $Y2_k$, $Y3_k$, $zY4_k$, $Y5_k$) to select the appropriate multiple {1X,…5X} with a 5:1 mux and a sign bit Ys_k that controls the negation of the selected multiple. The negative multiples are obtained by ten's complementing the positive ones. This is equivalent to taking the nine's complement of the positive multiple and then adding 1. the nine's complement can be obtained simply by bit inversion. This needs the positive multiplicand multiples to be coded in XS-3,with digits in [-3,12]. The d least significant partial products PP[d-1],…,PP[0] are generated from digits Yb_k by using a set of 5:1 muxes, as shown in Fig. 2. The xor gates at the output of the mux invert the multiplicand multiple, to obtain its 9's complement, if the SD radix-10 digit is negative (Ys_k =1).

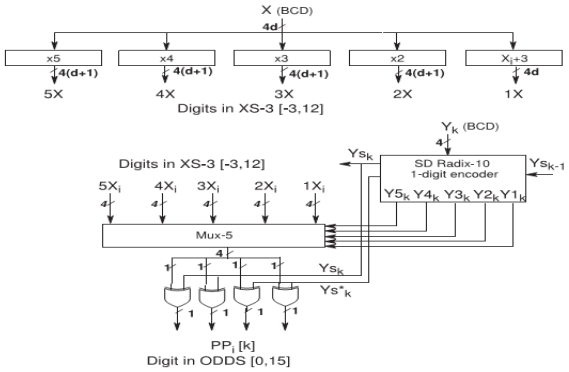


Fig.2. SD radix-10 generation of a partial product digit.

On the other hand, if the signals ($Y1_k$,$Y2_k$,$Y3_k$,$Y4_k$,$Y5_k$) are all zero then PP[k]=0, but it has to be coded in XS-3 (bit encoding 0011). Then, to set the two least significant bits to 1, the input to the XOR gate is $Ys_k^* = Ys_k \vee Ys_{k-}$ is zero denotes the boolean OR operator), where Yb_k is zero equals 1 if all the signals ($Y1_k$,$Y2_k$,$Y3_k$,$Y4_k$,$Y5_k$) are zero.

In addition, the partial product signs are encoded into their MSD. The generation of the most significant partial product PP[d] ,and only depends on Ys_{d-1}, the sign of the most significant SD radix-10 digit.

4.1 Generation of the multiplicand Multiples

We denote by NX$\in$ {1X,2X,3X,4X,5X}, the set of multiplicand multiples coded in the XS-3 representation, with digits $NX_i \in [-3,12]$, being $[NX_i]=NX_i + 3 \in [0,15]$ the corresponding value of the 4-bit binary encoding of NX_i given by Equation (2).

Fig. 3 shows the high-level block diagram of the multiples generation with just one carry propagation. This is performed in two steps: 1) digit recoding of the BCD multiplicand digits Xi into a decimal carry $0 \le Ti \le T_{max}$ and a digit $-3 \le D_i \le 12-T_{max}$, such as

$$D_i + 10 \times T_i = (N \times X_i) + 3, \qquad (5)$$

being T_{max} the maximum possible value for the decimal carry.

2) The decimal carries transferred between adjacent digits are assimilated obtaining the correct 4-bit representation of XS-3 digits NX_i , that is

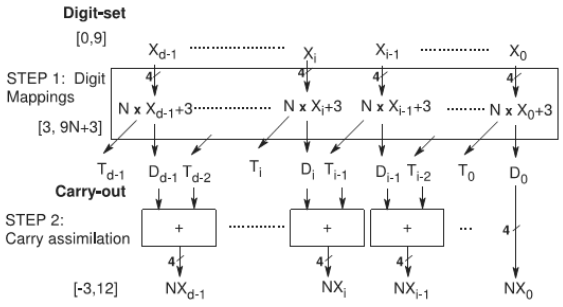


Fig.3. Generation of a decimal multiples NX.

The constraint for NX_i still allows different implementations for NX. For a specific implementation, the mappings for T_i and D_i have to be selected. Then, by inverting the bits of the representation of NX, operation defined at the ith digit by

$$\overline{NX_i} = 15 - [NX_i]$$

we obtain $\overline{NX_i}$. Replacing the relation between NX_i and $[NX_i]$ in the previous expression, it follows that

$$\overline{NX_i} = 15 - (NX_i + 3) = (9 - NX_i) + 3$$

That is, $\overline{NX}$ is the 9's complement of NX coded in XS-3,
with digits $\overline{NX_i} \epsilon [-3,12]$ and $[\overline{NX_i}] = \overline{NX_i} + 3 \epsilon [0,15]$.

4.2 Most-Significant Digit Encoding

The MSD of each PP[k], $PP_d[k]$, is directly obtained in the ODDS representation. Note that these digits store the carries generated in the computation of the multiplicand multiples and the sign bit of the partial product. For positive partial products we have

$$PP_d[k]=T_{d-1} \quad (7)$$

with $T_{d-1} \epsilon \{0,1,2,3,4\}$ (see Table 2). For negative partial products, the ten's complement operation leads to

$$PP_d[k] = -10 + (9 - T_{d-1}) = -1 - T_{d-1} \quad (8)$$

with $T_{d-1} \epsilon \{0,1,2,3,4\}$. Therefore the two cases can be expressed as

$$PP_d[k]=-Ys_k+(-1)^{Ys_k}T_{d-1}. \quad (9)$$

Since we need to encode $PP_d[k]$ in the ODDS range [0,15], we add and subtract 8 in Eq. (9), resulting in

$$PP_d[k]=-8+[PP_d[k]], \quad (10)$$

with
$$[PP[k]]=8-Ys_k+(-1)^{Ys_k}T_{d-1}.$$

Note that the term $[PP_d[k]]$ is always positive. Specifically, for positive partial products (Ys_k =0), this term results in $8+T_{d-1}$ that is within the range [8], [12] (since $0 \leq T_{d-1} \leq 4$).For negative partial products (Ys_k =1), this term results in $7-T_{d-1}$, that is within the range [3], [7]. All of the 8 terms of the different partial products are grouped together in a constant $-8 \times \sum_{k=0}^{d-1} 10^{k+d}$ that is added as a constant correction term to the results of the reduction array. Therefore, the $PP_d[k]$'s are encoded as $[PP_d[k]]$ without the -8 terms, which are added later (see Section 4.3), with only positive values of the form

$$[PP_d[k]]=\begin{cases}(8 + T_{d-1}), & if\ (Ys_k = 0),\\ (7 - T_{d-1}), & if\ (Ys_k = 1),\end{cases} \quad (11)$$

resulting in $[PP_d[k]] \epsilon [3,12]$. This encoding is implemented at bit level as an inversion of the 3 LSB's of T_{d-1} if Ys_k =1 and the concatenation of the MSB $\overline{Ys_k}$.

4.3 Correction Term

The resultant partial product sum has to be corrected off the-critical-path by adding a precomputed term, $f_c(d)$,which only depends on the format precision d. This term has to gather: (a) the -8 constants that have not been included in the MSD encoding and (b) a -3 constant for every XS-3 partial product digit (introduced to simplify the nine's complement operation).

Actually, the addition of these -3 constants is equivalent to convert the XS-3 digits of the partial products to the ODDS representation. Note that the 4-bit encoding of a XS-3 digit NX_i (or 9 $-Nx_i$) represents an ODDS digit with value

$[NX_i]=NX_i+3 \epsilon [0,15]$ or $[9-NX_i]=15[NX_i] \epsilon [0,15]$.

The pre-computed correction term is given by

$$f_c(d) = -8 \times \sum_{k=0}^{d-1} 10^{k+d} - 3 \times \left(\sum_{i=0}^{d-1} (i+1)10^{i+d} + \sum_{i=0}^{d-2} (d-1-i)10^{i+d} \right)$$

Particularizing for d=16 and d =34 digit operands, the following expressions for the correction term in 10's complement are obtained:

$$f_c(16) = -10^{32} + 074074074 \ldots 7037037.$$

$$f_c(34) = -10^{68} + 07407407 \ldots .7037037037. \quad (13)$$

The correction term is allocated into the array of $d + 1$ partial products coded in ODDS (digits in [0,15]), as we show in the next section.

4.4 Partial Product Array

As a conclusion of the considerations in the previous sections, Fig. 4 illustrates the shape of the partial product array, particularizing for d =16. Note that the maximum digit column height is d+1.

In each column several components can be observed. Digits labeled with O represent the redundant excess-3 BCD digits in the set [0,15]. Digits labeled with S_k represent the MSD of each partial product, $PP_d[k]$ (see Section 4.2).The 16 least significant digits of the correction term $f_c(16)$ are placed at the least significant position of each row after being added to Ys_k, to complete the 10's complement in case of a negative partial product; thus $H_k=Ys_k+\{0,3,7\}$ (digit wise addition, out of the critical path), so that $H_k \epsilon [0,8]$.Note that the negative bit -10^{32} is canceled with the carryout of the partial product sum in excess. The 16 leading digits of the correction term, $[f_c(16)]_d$, are added to the most significant partial product PP[d]. Thus, in parallel with the evaluation of the multiplicand multiples we compute

$XF = X + [f_c(16)]_d$

in the ODDS representation (note that this computation does not involve a carry propagation and it is out of the critical path). Digits labeled as F in Fig. 4, represent the most-significant partial product, $PP[d]$, where $PP[d] = XF$ if Ys_{d-1} =1 and $PP[d]=[f(16)]$ if $Ys_{d-1}=0$.

5 DECIMAL PARTIAL PRODUCT REDUCTION

The PPR tree consists of three parts: (1) a regular binary CSA tree to compute an estimation of the decimal partial product sum in a binary carry-save form (S, C), (2) a sum correction block to count the carries generated between the digit columns, and (3) a decimal digit 3:2 compressor which increments the carry-save sum according to the carries count to obtain the final double-word product (A, B), A being represented with excess-6 BCD digits and B being represented with BCD digits. The PPR tree can be viewed as adjacent columns of h ODDS digits each, h being the column height (see Fig. 4), and h≤d+1 .

Fig. 5 shows the high-level architecture of a column of the PPR tree (the ith column) with *h* ODDS digits in [0, 15](4 bits per digit). Each digit column of the binary CSA tree(the gray colored box in Fig. 5) reduces the h input digits and ncin input carry bits, transferred from the previous 17 partial products: 17(+1) digits wide Highest columns : 17 digits height column of the binary CSA tree, to two digits, S_i, C_i, with weight 10^i. Moreover, a group of n_{cout} carry outputs are generated and transferred to the next digit column of the PPR tree

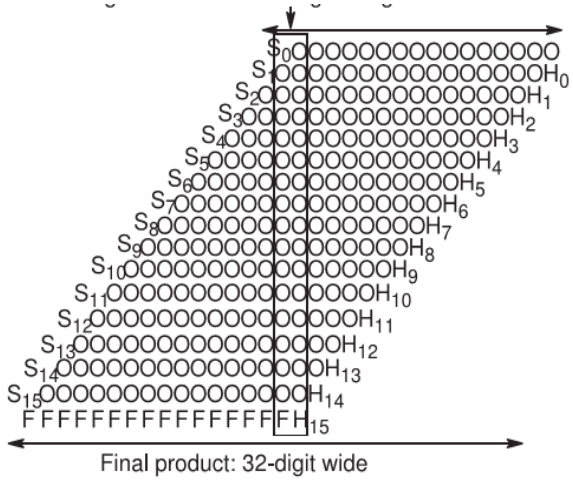


Fig.4. Decimal partial product array generated for d=16(16×16-digit multiplier).

S: Sign encoding. ODDS digit in [3,12]

H: 10's complement encoding: Ys+{0,3,7}.BCD digit in [0,8]

O: ODDS digit in [0,15]

F: Digit of operand XF or 0. ODDS digit in [0,15]

Roughly, the number of carries to the next column is $n_{cout} = h-2$.The digit columns of the binary CSA tree are implemented efficiently using 4-bit 3:2, 4:2 and higher order compressors made of full adders. These compressors take advantage of the delay difference of the inputs and of the sum and

carry outputs of the full adders, allowing significant delay reductions.

The weight of the carry-outs generated at the *i*th column, $C_{i+1}[0],\ldots,C_{i+1}[n_{cout}-1]$, is 16×10^{i} because the addition of the 4-bit digits is modulo 16. These carries are transferred to the column of the PPR tree, with weight $10^{i+1}=10\times10^{i}$. Thus, there is a difference between the value of the carry outs generated at the i-column and the value of the carries transferred to the $(i+1)$-column. This difference, T, is computed in the sum correction block of every digit column and added to the partial product sum (S, C) in the decimal CSA.

Defining

$$W_i = \sum_{k=0}^{n_{cout}-1} c_{i+1}[k] \qquad (14)$$

the contribution of the column i to the sum correction term T is given by

$$W_i \times 16 - W_{i\times}10 = W_i \times 6 \qquad (15)$$

Therefore, the sum correction is given by

$$T = \sum_{i=0}^{2d-1}(W_i \times 6 \times 10^i) = 6 \times \sum_{i=0}^{2d-1} W_i \times 10^i \qquad (16)$$

Consequently, the sum correction block evaluates $W_i\times6$. This module is composed of a m-bit binary counter and a ×6 operator. A straightforward implementation would use $m=n_{cout}$ and a decomposition of the ×6 operator into ×5 and ×1 (both without long carry propagations), and then a four to two decimal reduction to add the correction to the PPR tree result.

However, to balance paths and reduce the critical path-delay we considered some optimizations. Specifically, the optimized

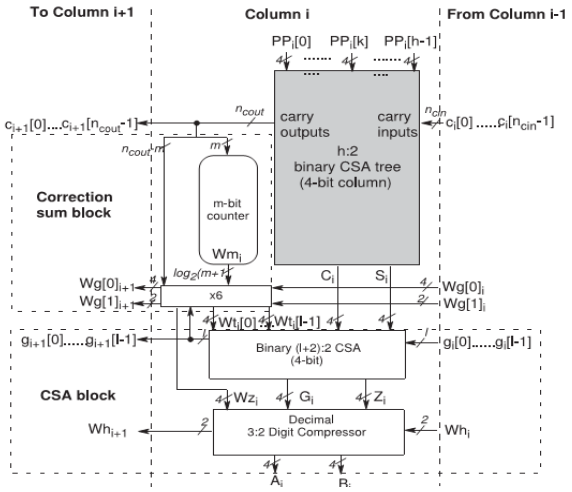


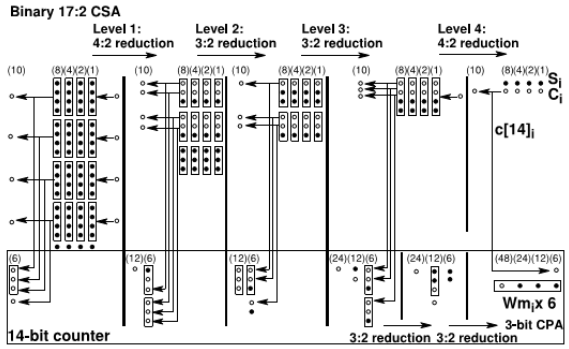
Fig. 5. High-level architecture of the proposed decimal PPR tree (h inputs, 1-digit column) implementation of this block heavily depends on the precision of the decimal representation; therefore its implementation is merely outlined here, without going into details. A detailed description of the implementation of the sum correction block is provided in Sections 5.1 and 5.2 for the Decimal64 and Decimal128 formats, respectively. To obtain W_i, the carries generated in the column are split into two parts: the m-bit counter adds the m first carries of the binary digit column and produces a binary sum Wm_i of $[\log_2(m+1)]$bits. The counter is implemented with full adders. To reduce the delay, the different arrival times of the carries have been taken into account.

Fig. 6a shows the dot-diagram representation of this reduction for a digit column with h= 17 (max. column height for Decimal64).

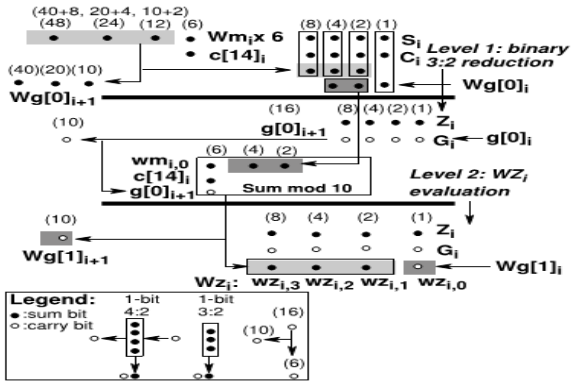
On the other hand, the remaining $n_{cout}-m$ carries are introduced directly into the ×6 block. Note that a suitable value for m minimizes the delay overhead due to the sum correction and simplifies the logic of the ×6 operation. The best value for m depends basically on h, the height of the corresponding digit column. It was first estimated using the delay evaluation model described in Section 7.1 and then validated by automated RTL synthesis of the VHDL model. The low-level implementation details of the ×6 module depend on the number of carry-outs, n_{cout} and on the size of the counter, m, and are explained in Sections 5.1 and

5.2.However, it can be advanced that the ×6 operation generates at most two carry digits $Wg[0]_{i+1}, Wg[1]_{i+1}$ to the next column. Moreover, to illustrate the stage, we show

The corresponding dot-diagram representation



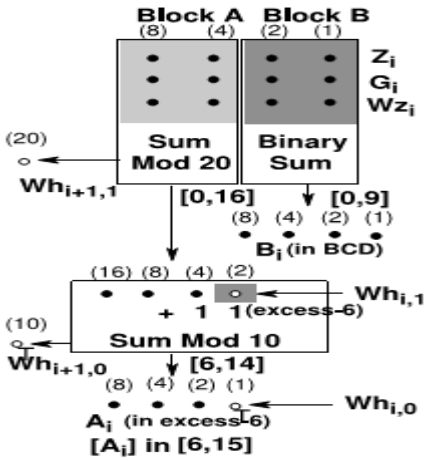
(a) Stage 1. Binary 17:2 CSA and 14-bit counter



(b) Stage 2. Decimal ×6 correction

(m=14) in Fig. 6b. An efficient implementation is obtained by representing the digit of $W_i \times 6$ with l ODDS digits, $Wt_i[0],\ldots,W_i[l-1])$, being l=1 for Decimal64, and l=2 for Decimal128.

After that, the sum correction digits $(Wt_i[0],\ldots..,Wt_i[l-1])$ and the output digits of the binary CSA tree (S_i, C_i) are reduced to two ODDS digits $G_i \in [0,15]$, and $Z_i \in [0,15]$,using a 4-bit binary $(l+2)$: 2 CSA.



This CSA generates l carry outs $g_{i+1}[0],\ldots,g_{i+1}[l-1]$with weight 16×10^i, which are transferred to the next column, and introduced into the ×6 block to produce another ODDS digit, $Wz_i \in [0,15]$.

The last step is the addition of digits *G,Z,Wz* of the column, $G_i+Z_i+Wz_i \in [0,45]$. We have designed a decimal 3:2 digit compressor that reduces digits Wz_i, G_i and Z_i to two digits A_i,B_i. The dot-diagram of the decimal 3:2 digit compressor is shown in Fig. 6c. To obtain the final BCD product by using a single BCD carry propagate addition, that is, $P=A+B$, which is the last step in the multiplication (see Fig. 1 and Section 3), it is required that $A_i+B_i \in [0,18]$. Moreover, to reduce the delay of the final BCD carry-propagate adder (see Section 6) operand A is obtained in excess-6, so that we compute $[A_i]=A_i+e$ in excess e= 6 as defined by Equation (2), being the output
digits sum $[A_i]+B_i \in [6,24]$.

The evaluation is split in two parts:

Block A computes the sum of the two MSBS of the input digits (the bits with weights 8 and 4), and a two-bit carry input $Wh_i \in \{0,1,2,3\}$. This sum is in [0,39]. The outputs of this block are a BCD digit A_i in excess-6 $[A_i] \in [6,15]$ and a two-bit decimal carry output $Wh_{i+1} \in \{0,1,2,3\}$ which is transferred to the next column (the i+1th column). Note that the LSB of the carry output Wh_{i+1} depends on the MSB of the input carry Wh_i. However, there is

no further carry propagation since the LSB of Wh_{i+1} is just the LSB of $[A_{i+1}]$,that is, $[A_{i+1,0}]$.

On the other hand, Block B implements the sum of the two LSB bits of the input digits (the bits with weights 2 and1). This sum is in [0,9], so that B_i is evaluated as a regular binary addition.

5.1 Decimal 64 Implementation

The partial product array generated in the proposed 16×16-digit BCD multiplier is shown in Fig. 4. The maximum column height in the partial product array is $h_{max}=d+1=17$. Therefore, a binary 17 : 2 CSA tree is required in this case, while other columns need CSA trees with a smaller number of inputs. Fig. 7 shows the implementation of the PPR tree for the maximum height columns. As stated previously, the maximum number of carries transferred between adjacent columns of the binary 17 : 2 CSA tree is 15. These carries are labeled $c_{i+1}[0],\ldots\ldots c_{i+1}[0]$ (output carries) and $c_i[0],\ldots c_i[14]$ (input carries) in the figure. The binary 17 : 2 CSA tree is built of a first level composed of a 9 : 2 compressor and a 8 : 2 compressors, and a second level composed of a 4 : 2 compressor. To balance the delay of the 17 : 2 CSA tree and the bit counter, m=14has been chosen.

The 14-bit counter produces the 4-bit digit Wm_i. The computation of $Wm_i \times 6$ deserves a more detailed description. The 4-bit digit $Wmi = wm_{i,3}, wm_{i,2}, wm_{i,1}, wm_{i,0}$ with $wm_{i,j}$ being the bits of the digit, is conveniently represented as

$$Wm_i = Wg[0]_{i+1} \times 2 + wm_{i,0}, \quad (17)$$

With

$$Wg[0]_{i+1} = \sum_{j=1}^{3} wm_{i,j} \times 2^{j-1}. \quad (18)$$

Note that Wm_i has been split into two parts, the three most significant bits,$Wg[0]_{i+1} \in [0,7]$, and the least-significant bit,$wm_{i,0}$. Then, $W_i = Wm_i + c_{i+1}[14]$ results in

$$W_i = Wg[0]_{i+1} \times 2 + wm_{i,0} + c_{i+1}[14],$$

and consequently,

$$W_i \times 6 = Wg[0]_{i+1} \times 12 + (wm_{i,0} + c_{i+1}[14] \times 6$$

$$= (Wg[0]_{i+1} \times 10 + Wg[0]_{i+1} \times 2) + (wm_{i,0} + c_{i+1}[14]) \times 6. \quad (20)$$

The first term in Equation (20) represents a digit transfer to the next column. On the other hand, multiplication by 2 is implemented by shifting the binary representation of $Wg[0]_{i+1}$ one bit to the left, so that $Wg[0]_{i+1} \times 2 \in [0,14]$ and its least-significant bit is 0.Then, the sum correction term (*sct*) at the *i*th column, discarding the digit ($Wg_{i+1}[0]\times10$ transferred to the *i*+1thcolumn and taking into account the digit transferred from the i 1th column, is given by This sum correction term has been encoded in two ODDS digits $Wt_i \in [0,15]$ and $Wz_i \in [0,15]$ as follows

- Digit Wis obtained by the concatenation of the most-significant bit of $Wg[0]_{i+1} \times 2$ and $Wg[0]_i$, the digit transfer from the *i-1*th column. Note that $Wg[0]_i$ is represented with only three bits ($Wg[0]_i \in [0,7]$) and the concatenation of the most-significant digit of $Wg[0]_{i+1}\times2$ results in a 4-bit digit $Wt_i \in [0,15]$.
- Digit Wt_i is added to the binary CSA tree column sum (S_i ,C_i) using a binary 4-bit 3 : 2 CSA, reducing these three digits to a double digit (G_i, Z_i).

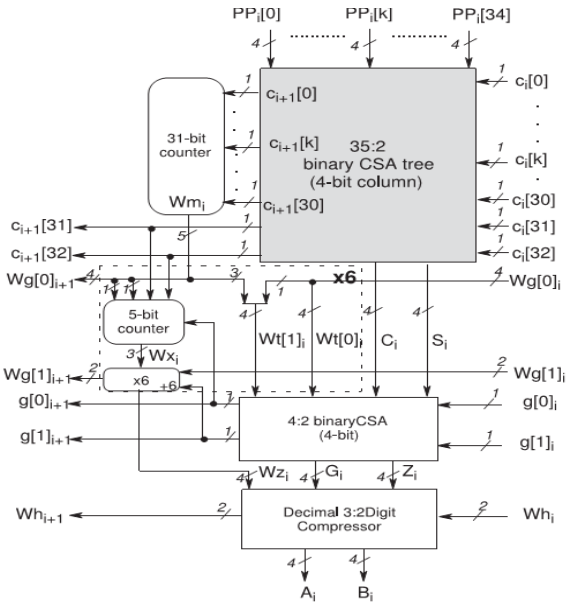


Fig.8. implementation of the PPR Tree Highest Column (h=35) for a 34×34-digit multiplication

- A carry output g_{i+1} is transferred to the i+1th column. Therefore, to obtain the sum correction term, the carry gi transferred from i-1th column has to be considered.
- Digit Wz_i is obtained by the addition of the two least significant bits of $Wg[0]_{i+1}\times 2$,$(wm_{i,0}+c_{i+1}[14])\times 6$ (see Equation (21)) and the carry $g_{i+1}\times 6$. Moreover, this addition produces a carry out bit $Wg[1]_{i+1}$ and a sum digit in [0,14]. A carry-in $Wg[1]_i$ is concatenated to this sum digit to obtain Wz_i.

Finally, a row of decimal 3 : 2 digit compressors is used to reduce the 3-operand partial product sum (*G*, *Z*, *Wz*) to two BCD operands (*A*, *B*), with *A* represented in excess-6(see Fig. 6c).

5.2 Decimal 128 Implementation.

The maximum height of the partial product array by the 34×34-digit BCD multiplier is h = 35. The proposed implementation for the maximum height columns of the PPR tree is shown in Fig. 8. The binary 35 : 2 CSA tree is built of a first level of three 9 : 2 and one 8 : 2 compressors, and a second level of one 8 : 2 compressor. The number of carries transferred to the next column of the binary CSA tree is 33.The optimal value for parameter m is m = 31. Therefore, the addition of these carries has been split into two parts. First, a 31-bit counter evaluates Wm_i, the 5-bit sum of the 31 fastest carries. Then, the two slowest carries, $c_{i+1}[31]$and $c_{i+1}[32]$, are added to Wm_i into a second 5-bit counter.

Operation $Wm_i\times 6$ is decomposed as follows:

$$Wm_i \times 6 = Wg[0]_{i+1} \times 10 + Wg[0]_{i+1} \times 2 + wm_{i,0} \times 6 \quad (22)$$

With $Wg[0]_{i+1} = \sum_{j=1}^{4} wm_{i,j} \times 2^{j-1} \epsilon [0,15]$.

As in the decimal 64 case, the first term is transferred to the $i+1$ th column and the multiplication by 2 is implemented as a one-bit binary left shift. The concatenation of digit $Wg[0]_i$,transferred from column i-1, produces two redundant BCD digits $Wt[0]_i\epsilon[0,15]$ and $Wt[1]_i\epsilon[0,15]$.

The 5-bit counter adds the remaining terms in Equation (22) with carries $c_{i+1}[31]$ and $c_{i+1}[32]$ and the multiplication ×6 produces a 4-bit digit Wz_i and two carries to the $i+1$th column, $Wg[1]_{i+1}$.

After that, digits S_i,C_i,$Wt[0]_i$,$Wt[1]_i$, are reduced to two digits G_i, Z_i $\epsilon[0,15]$ using a 4-bit binary 4 : 2 CSA. The two carries generated in the 4-bit 4 : 2 CSA are transferred to the next column i+1 and introduced into the 5-bit counter.

Finally, the three digits G_i ,Z_i ,Wz_i are reduced to two excess-6 BCD digits A_i and B_i by using the decimal digit 3 : 2 compressor.

6 .FINAL CONVERSIONS TO BCD

The selected architecture is a 2d-digit hybrid parallel prefix/carry-select adder, the BCD Quaternary Tree adder [29].The delay of this adder is slightly higher to the delay of a binary adder of 8d bits with a similar topology.

The decimal carries are computed using a carry prefix tree, while two conditional BCD digit sums are computed out of the critical path using 4-bit digit adders which implements $[A_i]+B_i+0$ and $[A_i]+B_i+1$. These conditional sums correspond to each one of the carry input values. If the conditional carry out from a digit is one, the digit adder performs a -6 subtraction. The selection of the appropriate conditional BCD digit sums is implemented with a final level of 2 : 1 multiplexers.

To design the carry prefix tree we analyzed the signal arrival profile from the PPRT tree, and considered the use of different prefix tree topologies to optimize the area for the minimum delay adder.

7. EVALUATION ANDCOMPARISON

The proposed combinational architectures for BCD 16×16-digit and 34×34-digit multipliers are evaluated and compared with other representative BCD multipliers. The area and delay figures of our architectures were obtained from an area-delay evaluation model for static CMOS gates, and validated with the synthesis of verified RTL models coded in VHDL.

7.2 Synthesis Report :

Fig 10.block diagram of top module.

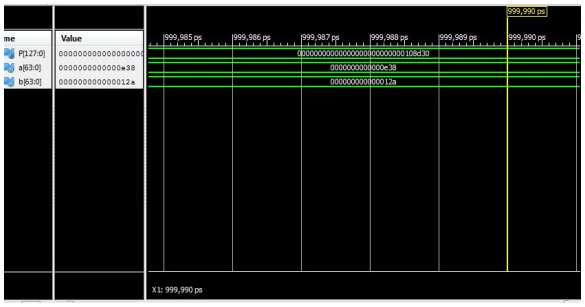


Fig 11.simuation results

CONCLUSION

In this paper we have presented the algorithm and architecture of a new BCD parallel multiplier. The improvements of the proposed architecture rely on the use of certain redundant BCD codes, the XS-3 and ODDS representations. Partial products can be generated very fast in the XS-3 representation using the SD radix-10 PPG scheme: positive multiplicand multiples (0X, 1X, 2X, 3X, 4X, 5X) are pre-computed in a carry-free way, while negative multiples are obtained by bit inversion of the positive ones. On the other hand, recoding of XS-3 partial products to the ODDS representation is straightforward. The ODDS representation uses the redundant digit-set [0, 15] and a 4-bit binary encoding (BCD encoding), which allows the use of a binary carry-save adder tree to perform partial product reduction in a very efficient way. We have presented architectures for IEEE-754 formats, Decimal64 (16 precision digits) and Decimal128 (34 precision digits). The area and delay figures estimated from both a theoretical model and synthesis show that our BCD multiplier presents 20-35 percent less area than other designs for a given target delay.

REFERENCES

[1] A. Aswal, M. G. Perumal, and G. N. S. Prasanna, "On basic financial decimal operations on binary machines," IEEE Trans. Comput., vol. 61, no. 8, pp. 1084–1096, Aug. 2012.

[2] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb, "Adecimal floating-point specification," in Proc. 15th IEEE Symp. Comput. Arithmetic, Jun. 2001, pp. 147–154.

[3] M. F. Cowlishaw, "Decimal floating-point: Algorism for computers," in Proc. 16th IEEE Symp. Comput. Arithmetic, Jul. 2003,pp. 104–111..

[4] S. Carlough and E. Schwarz, "Power6 decimal divide," in Proc. 18thIEEE Symp. Appl.-Specific Syst., Arch., Process., Jul. 2007, pp. 128-133.

[5] S. Carlough, S. Mueller, A. Collura, and M. Kroener, "The IBMzEnterprise-196 decimal floating point accelerator," in Proc. 20th

IEEE Symp. Comput. Arithmetic, Jul. 2011, pp. 139–146.

[6] L. Dadda, "Multioperand parallel decimal adder: A mixed binaryand BCD approach," IEEE Trans. Comput., vol. 56, no. 10,pp. 1320–1328, Oct. 2007.

[7] L. Dadda and A. Nannarelli, "A variant of a Radix-10 combinational multiplier," in Proc. IEEE Int. Symp. Circuits Syst., May 2008, pp. 3370–3373.

[8] L. Eisen, J. W. Ward, H.-W. Tast, N. Mading, J. Leenstra, S. M.Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, "IBM POWER6 accelerators: VMX and DFU," IBM J. Res. Dev.,vol. 51, no. 6, pp. 663–684, Nov. 2007.

[9] M. A. Erle and M. J. Schulte, "Decimal multiplication via carrysave addition," in Proc. IEEE Int. Conf Appl.-Specific Syst., Arch., Process., Jun. 2003, pp. 348–358.

[10] M. A. Erle, E. M. Schwarz, and M. J. Schulte, "Decimal multiplication with efficient partial product generation," in Proc. 17th IEEE Symp. Comput. Arithmetic, Jun. 2005, pp. 21–28.

[11] Faraday Tech. Corp. (2004). 90nm UMC L90 standard performancelow-K library (RVT). [Online]. Available: http://freelibrary.faraday-tech.com/

[12] S. Gorgin and G. Jaberipur, "A fully redundant decimal adder andits application in parallel decimal multipliers," Microelectron. J., vol. 40, no. 10, pp. 1471–1481, Oct. 2009.

[13] S. Gorgin and G. Jaberipur. (2013, May). "High speed paralleldecimal multiplication with redundant internal encodings," IEEETrans. Comput. vol. 62, no. 5, [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TC.2013.160

[14] L. Han and S. Ko, "High speed parallel decimal multiplicationwith redundant internal encodings," IEEE Trans. Comput., vol. 62, no. 5, pp. 956–968, May 2013.

[15] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754(TM)-2008IEEE Comput. Soc., Aug. 2008.

[16] G. Jaberipur and A. Kaivani, "Improving the speed of paralleldecimal multiplication," IEEE Trans. Comput., vol. 58, no. 11, pp. 1539–1552, Nov. 2009.

[17] R. D. Kenney, M. J. Schulte, and M. A. Erle, "High-frequency decimal multiplier," in Proc. IEEE Int. Conf. Comput. Des.: VLSI Comput. Process., Oct. 2004, pp. 26–29.

[18] R. D. Kenney and M. J. Schulte, "High-speed multioperand decimaladders," IEEE Trans. Comput., vol. 54, no. 8, pp. 953–963, Aug. 2005.

[19] T. Lang and A. Nannarelli, "A Radix-10 combinational multiplier," in Proc. 40th Asilomar Conf. Signals, Syst., Comput., Oct.2006, pp. 313–317.

[20] T. Ohtsuki, Y. Oshima, S. Ishikawa, H. Yabe, and M. Fukuta,"Apparatus for decimal multiplication," U.S. Patent 4 677 583, Jun.1987.

[21] R. K. Richards, Arithmetic Operations in Digital Computers. NewYork, NY, USA: Van Nostrand, 1955.