A NEW VLSI ARCHITECTURE FOR MULTIPLICATION USING RADIX-10 MODIFIED BOOTH ENCODING TECHNIQUE

Gone Ragini¹, V.Ramya²
¹PG Scholar in VLSI, ²Asst. Professor, ECE Department, Vaagdevi College of Engineering, Bollikunta, Warangal.

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, 1x, 2x, 3x, 4x, 5x) 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 16x16-digit and 34x34-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 implementations.

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

I. 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 has encouraged a significant amount of research in decimal 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 has encouraged a significant amount of research in decimal 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 has encouraged a significant amount of research in decimal 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 has encouraged a significant amount of research in decimal 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 has encouraged a significant amount of research in decimal 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 has encouraged a significant amount of research in decimal computing, where conversion and rounding errors that are inherent to floating-point binary representations cannot be tolerated [3].
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 by 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 3x, cannot be obtained in acary-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-bit binary value represents a redundant radix-10 digit Xic [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 (2x, 3x, 4x, 5x, 6x,...) 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 implementation 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 bere coded 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.

II. 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:

\[ Z = -s_2 \times 10^d + \sum_{i=1}^{d} Z_i \times 10^i, \]

where \( d \) is the number of digits, \( s \) is the sign bit, and \( Z_i \in [0 \leq l \leq e, 9 + e \leq m \leq 2^4 - 1 (= 15) \). Parameter \( e \) is the excess of the representation and usually takes values 0 (non excess), 3 or 6. The redundancy index \( \rho \) 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\((Z_e \in [0,9], e = 0, l = 0, m = 9, \rho = 0)\), BCD excess-3\((Z_e \in [0,9], e = 3, l = 3, m = 12, \rho = 0)\), BCD excess-6\((Z_e \in [0,9], e = 6, l = 6, m = 15, \rho = 0)\) and redundant representations \((\rho > 0)\), such as the ODDS representation\((Z_e \in [0,15], e = 0, l = 0, m = 15, \rho = 6)\), or the XS-3 representation \((Z_e \in [-3,12], e = 3, l = 0, m = 15, \rho = 6)\). On the other hand, the binary value of the 4-bit vector representation of \( Z_i \) is given by

\[ Z_i = \sum_{j=0}^{3} Z_{ij} \times 2^j, \]

where \( j \) being the \( j \)th bit of the \( i \)th 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 \((-5x, -4x, ... , 0x, ... , 5x)\) 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 \([0,9]\), \( e = 0, \rho = 0 \)), 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 \( x3 \) 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

\[
-10^d + \sum_{i=0}^{d-1} (9 - Z_i) \times 10^i.
\]

The implementation of \((9 - Z_i)\) 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

<table>
<thead>
<tr>
<th>Digit</th>
<th>Nine’s Complement for the XS-3 Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4-bit Encoding</td>
</tr>
<tr>
<td>0000</td>
<td>-3</td>
</tr>
<tr>
<td>0001</td>
<td>-2</td>
</tr>
<tr>
<td>0010</td>
<td>-1</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>2</td>
</tr>
<tr>
<td>0110</td>
<td>3</td>
</tr>
<tr>
<td>0111</td>
<td>4</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
</tr>
<tr>
<td>1001</td>
<td>6</td>
</tr>
<tr>
<td>1010</td>
<td>7</td>
</tr>
<tr>
<td>1011</td>
<td>8</td>
</tr>
<tr>
<td>1100</td>
<td>9</td>
</tr>
<tr>
<td>1101</td>
<td>10</td>
</tr>
<tr>
<td>1110</td>
<td>11</td>
</tr>
<tr>
<td>1111</td>
<td>12</td>
</tr>
</tbody>
</table>

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:

\[(9 - Z_i) + 3 = 15 - (Z_i + 3)\]  \(\text{(4)}\)

For the range \( Z_i \in \{-3,12\}\) ([\(Z_i\] \in \([0,15]\)]. Therefore 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 the XS-3 code (more details in Section 4.1). In summary, the main reasons for using the redundant XS-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).

### III. HIGH-LEVEL ARCHITECTURE

The high-level block diagram of the proposed parallel architecture for \(d\times d\)-digit BCD decimal integer and fixed-point multiplication is shown in Fig. 1. This architecture accepts conventional (non-redundant) BCD inputs \(X, Y\) and generates redundant BCD partial products PP, and computes the BCD product \(P = X \times Y\). It consists of the following three stages: (1) parallel generation of partial products coded in \(X\S3\), including generation of multiplicand multiples and recoding of the multiplier operand, (2) recoding of partial products from \(X\S3\) 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].

Stage 2) Decimal partial product reduction. In this stage, the array of \(d+1\) ODDS partial products are reduced to twodigit 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 \(\leq d + 1\). 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 offer 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 Ai + Bi to be obtained in excess-6. Several choices are possible. We opt for representing operand A in BCDexcess-6 (Ai, e[0,9], [Ai] = Ai + e, e=6) and B coded inBCD6-e[0,9], e=0.

IV. 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 multipicand 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, with k=0,...,d-1, Yb, being the most significant digit (MSD) of the multiplier [29]. Each digit Yb is represented with a 5-bit hot-one code (Yb, Yb, Yb, zYb, Yb) to select the appropriate multiple {1X,...,5X} with a 5:1 mux and a sign bit Yb 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 negative complement of the positive multiple and then adding 1, the nine’s complement can be obtained simply by bit inversion. This needs the positive multipicand 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 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 (YSk =1).

![Fig.2. SD radix-10 generation of a partial product digit.](image)

On the other hand, if the signals (Y1, Y2, Y3, Y4, Y5) 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 = YSk, YSk is zero (boolean OR operator), where Yb is zero if all the signals (Y1, Y2, Y3, Y4, Y5) 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 YSk, the sign of the most significant SD radix-10 digit.

A. Generation of the multipicand multiples
We denote by NXi {1X,2X,3X,4X,5X}, the set of multipicand multiples coded in the XS-3 representation, with digits NXi ∈ [-3,12], being [NXi]=NXi + 3c [0,15] the corresponding value of the 4-bit binary encoding of NXigiven 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≤Ti ≤Tmax and a digit -3≤Di≤12-Tmax, such as Di + 10 × Ti = (N × Xi) + 3, (5) 2) the decimal carries transferred between adjacent digits are assimilated obtaining the correct 4-bit representation of XS-3 digits NXi, that is

![Fig.3. Generation of a decimal multiples NX](image)
representation of NX.operation defined at the ith digit by
\[ \frac{NX}{} = 15 - \lfloor NX \rfloor \]
we obtain \( NX \). Replacing the relation between \( NX \) and \( \lfloor NX \rfloor \) in the previous expression, it follows that
\[ \frac{NX}{} = 15 - (NX + 3) = (9 - NX) + 3 \]
That is, \( NX \) is the 9's complement of \( NX \) coded in XS-3, with digits \( \lfloor NX \rfloor \in [-3,12] \) and \( \lfloor NX \rfloor = NX + 3e[0,15] \).

B. Most-Significant Digit Encoding

The MSD of each PP\([k]\), PP\([d]\), 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
\[ PP[d][k]=T_{d,1} \]
with \( T_{d,1} \in \{0,1,2,3,4\} \) (see Table 2). For negative partial products, the ten’s complement operation leads to
\[ PP[d][k] = -10 + (9 - T_{d,1}) = -1 - T_{d,1} \]
with \( T_{d,1} \in \{0,1,2,3,4\} \). Therefore the two cases can be expressed as
\[ PP[d][k]=Ys_{k}+(-1)^{k}T_{d,1} \]
Since we need to encode PP\([k]\) in the ODDS range \([0,15]\), we add and subtract 8 in Eq. (9), resulting in
\[ PP[d][k]=8+PP[d][k] \]
With \([PP[d][k]]=Ys_{k}+(-1)^{k}T_{d,1} \).

Note that the term \([PP[d][k]]=\) always positive. Specifically, for positive partial products (\( Ys_{k} = 0 \), this term results in \( 8+PP[d][k] \), 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 the positive values of the form
\[ PP[d][k]=\begin{cases} \frac{8+PP[d][k]}{2}, & \text{if } (Ys_{k} = 0), \\ \frac{7-P[d][k]}{2}, & \text{if } (Ys_{k} = 1), \end{cases} \]
resulting in \([PP[d][k]]=3[3,12] \). This encoding is imple-mented 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 \( V_{SK} \).

C. Correction Term

The resultant partial product sum has to be corrected off-the-critical-path by adding a pre-computed 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\(_o\) (or 9 - NX\(_o\) ) represents an ODDS digit with value \([NX_o]=NX_o+3e[0,15] \) or \([9-NX_o]=15-NX_o \in \{0,15\} \). The pre-computed correction term is given by
\[
f_{c}(d) = -8 \times \sum_{k=0}^{d-1} 10^{k+d} - 3 \]
\[
\times \left( \sum_{i=0}^{d-1} (i+1)10^{i+d} \right) \]
\[
+ \sum_{i=0}^{d-1} (d - 1 - i)10^{i+d} \]
Particularizing for \( d=16 \) and \( d=34 \) digit operands, the following expressions for the correction term in 10's complement are obtained:
\[
f_{c}(16) = -10^{16} + 074074074 \ldots 7037037037 \]
\[
f_{c}(34) = -10^{34} + 07407407407 \ldots 7037037037 \]
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.

D. 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 represent the MSD of each partial product, PP\([k]\) (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 \( H_{k} \in [0,8] \). Note that the negative bit \(-10^{16} \) is canceled with the carryout of the partial product sum in excess. The 16 leading digits of the correction term, \([f_{c}(16)]_a\), are added to the most significant partial product PP\([d]\). Thus, in parallel with the evaluation of the multiplicand multiples we compute \( XF = X + [f_{c}(16)]_a \) 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 \( PP[d] = [f_{c}(16)]_a \) if \( Ys_{d,1}=0 \).

V. 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 hODDS 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 nc in input carry bits, transferred from the previous 17 partial products: 17(1+1) digits wide. Highest columns: 17 digits height column of the binary CSA tree, to two digits, $S_1$, $C_1$, with weight 10. Moreover, a group of $n_{\text{cout}}$ carry outputs are generated and transferred to the next digit column of the PPR tree.

Consequently, the sum correction block evaluates $W_1$. This module is composed of a m-bit binary counter and a $\times 6$ operator. A straightforward implementation would use $m=n_{\text{cout}}$ and a decomposition of the $\times 6$ operator into $\times 5$ and $\times 1$ (both without long carry propagations), and then a four to two decimal reduction to add the correction to the PPR tree. However, to balance paths and reduce the critical path-delay we considered some optimizations. Specifically, the optimized

$$W_1 = \sum_{k=0}^{n_{\text{cout}}-1} c_i + 1 \left[ k \right]$$

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

$$W_i \times 16 - W_{i+1} = W_i \times 6$$

Therefore, the sum correction is given by

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

Consequently, the sum correction block evaluates $W_i \times 6$. This module is composed of a m-bit binary counter and a $\times 6$ operator. A straightforward implementation would use $m=n_{\text{cout}}$ and a decomposition of the $\times 6$ operator into $\times 5$ and $\times 1$ (both without long carry propagations), and then a four to two decimal reduction to add the correction to the PPR tree. However, to balance paths and reduce the critical path-delay we considered some optimizations. Specifically, the optimized
Stage 1. Binary 17:2 CSA and 14-bit counter

Stage 2. Decimal ×6 correction

This CSA generates 1 carry outs gi+1[0],….,gi+1[14] with weight 16×10i, which are transferred to the next column, and introduced into the ×6 block to produce another ODDS digit, Wzi ∈[0,15]. The last step is the addition of digits G, Z, Wz of the column, Gzi+Zi+Wzi ∈[0,45]. We have designed a decimal 3:2 digit compressor that reduces digits Wzi, Gi and Zi to two digits Ai, Bi. 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 Ai+Bi ∈[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 [Ai]=Ai+6 in excess e=6 as defined by Equation (2), being the output digits sum [Ai]+6[14]. 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 Whi ∈[0,1,2,3]. This sum is in [0,39]. The outputs of this block are a BCD digit Ai in excess-6 [Ai]e[6,15] and a two-bit decimal carry output Whi+1 ∈[0,1,2,3] which is transferred to the next column (the i+1th column). Note that the LSB of the carry output Whi+1 depends on the MSB of the input carry Whi. However, there is no further carry propagation since the LSB of Whi+1 is just the LSB of [Ai+1], that is, [Ai+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 and 1). This sum is in [0,9], so that Bi is evaluated as a regular binary addition.

A. 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 hmax=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 ci[0],….,ci[14] (output carries) and ci[0],….,ci[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 compressor, 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=14 has been chosen. The 14-bit counter produces the 4-bit digit Wmi. The computation of Wm1 is similar to the previous case. The 4-bit digit Wmi is decomposed into its MSBS and LSBs. The 4-bit digit Wmi = wmi,1 wmi,2 wmi,3 wmi,0 with wmi,j being the bits of the digit, is conveniently represented as

\[ W_{m1} = W g[0]_{i+1} × 2 + w_{m1,0} \]  

With

\[ W g[0]_{i+1} = \sum_{j=1}^{4} w_{m1,j} × 2^{j-1}. \]  

Note that Wm1 has been split into two parts, the three most significant bits, Wg[0]_{i+1} ∈[0,7], and the least-significant bit, w_{m1,0}. Then, Wf = W_{m1} + c_{i+1}[14] results in

\[ W_f = W g[0]_{i+1} × 2 + w_{m1,0} + c_{i+1}[14]. \]  

and consequently,

\[ W_f × 6 = W g[0]_{i+1} × 12 + (w_{m1,0} + c_{i+1}[14]) × 6 = (W g[0]_{i+1} × 10 + W g[0]_{i+1} × 2) + (w_{m1,0} + c_{i+1}[14]) × 6. \]  

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} ×2[0,14] and its least-significant bit is 0. Then, the sum correction term (sct) at the ith column, discarding the digit (Wg_{si}[0]×10
transferred to the i+1th column 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 Wtϵ[0,15] and Wzϵ[0,15] as follows

- Digit Wt obtained by the concatenation of the most-significant bit of Wg[0]i×2 and Wg[0], the digit transfer from the i-1th column. Note that Wg[0], is represented with only three bits (Wg[0], ϵ[0,7]) and the concatenation of the most-significant digit of Wg[0]i×2 results in a 4-bit digit Wt, ϵ[0,15].

- Digit Wt, is added to the binary CSA tree column sum (Si, Ci) using a binary 4-bit 3 : 2 CSA, reducing these three digits to a double digit (G, Z).

\[
W_t = W_g[0]i \times 2 + W_g[0, i+1] \times 2 + w_m,0 \times 6
\]

\[
W_t = \sum_{i=1}^{j} w_m, i \times 2^{i-1} \epsilon[0,15],
\]

As in the decimal 64 case, the first term is transferred to the i + 1th column and the multiplicity by 2 is implemented as a one-bit binary left shift. The concatenation of digit Wg[0], transferred from column i-1, produces two redundant BCD digits Wt[i]ϵ[0,15] and Wt[1],ϵ[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, and two carries to the i + 1th column, Wg[1],i+1. After that, digits S, Ci, Wt[i], Wt[1], are reduced to two digits G, Z, ϵ[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, Z, Wz are reduced to two excess-6 BCD digits A, and B, by using the decimal digit3 : 2 compressor.

VI. 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]+B₄+0 and [A]+B₅+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 PPR tree, and considered the use of different prefix tree topologies to optimize the area for the minimum delay adder.

VII. 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.

A. Synthesis Report:

Fig 10. block diagram of top module.
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 in version 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


