July 2012

RADIX-10 PARALLEL DECIMAL MULTIPLIER

MRUNALINI E. INGLE

Electronics Engineering, Yeshwantrao Chavan College of Engineering, Nagpur, India,
mrunalingle@gmail.com

TEJASWINI PANSE

Electronics Engineering, Yeshwantrao Chavan College of Engineering, Nagpur, India,
tejaswini.deshmukh@gmail.com

Follow this and additional works at: <https://www.interscience.in/ijess>

Part of the [Electrical and Electronics Commons](#)

Recommended Citation

INGLE, MRUNALINI E. and PANSE, TEJASWINI (2012) "RADIX-10 PARALLEL DECIMAL MULTIPLIER,"
International Journal of Electronics Signals and Systems: Vol. 2 : Iss. 1 , Article 4.

DOI: 10.47893/IJESS.2012.1059

Available at: <https://www.interscience.in/ijess/vol2/iss1/4>

This Article is brought to you for free and open access by the Interscience Journals at Interscience Research Network. It has been accepted for inclusion in International Journal of Electronics Signals and Systems by an authorized editor of Interscience Research Network. For more information, please contact sritampatnaik@gmail.com.

RADIX-10 PARALLEL DECIMAL MULTIPLIER

¹MRUNALINI E. INGLE & ²TEJASWINI PANSE

^{1&2}Electronics Engineering, Yeshwantrao Chavan College of Engineering, Nagpur, India
E-mail : mrunalingle@gmail.com, tejaswini.deshmukh@gmail.com

Abstract - This paper introduces novel architecture for Radix-10 decimal multiplier. The new generation of high-performance decimal floating-point units (DFUs) is demanding efficient implementations of parallel decimal multiplier. The parallel generation of partial products is performed using signed-digit radix-10 recoding of the multiplier and a simplified set of multiplicand multiples. The reduction of partial products is implemented in a tree structure based on a new algorithm decimal multioperand carry-save addition that uses a unconventional decimal-coded number systems. We further detail these techniques and it significantly improves the area and latency of the previous design, which include: optimized digit recoders, decimal carry-save adders (CSA's) combining different decimal-coded operands, and carry free adders implemented by special designed bit counters.

Keywords— *Decimal computer arithmetic, parallel decimal multiplication, partial product generation and reduction, Decimal carry-save addition.*

I. INTRODUCTION

Hardware implementations of decimal arithmetic units have recently gained importance because they provide higher accuracy in financial, commercial, scientific, and internet based applications. One reason is the need for precise floating-point representation of many decimal values (e.g. 0.5) that do not have an exact binary representation[1]. The revision of the IEEE 754 Standard for Floating-Point Arithmetic (IEEE 754-2008) incorporates specifications for DFP arithmetic that can be implemented in software, hardware, or in a combination of both.

The computer arithmetic literature includes articles on decimal arithmetic hardware such as BCD adders and multioperand BCD adders, sequential BCD multipliers and dividers. However, some single arithmetic operations (e.g., division or square root), and also function evaluation circuits (e.g., radix-10 exponentiation or logarithm), are often implemented by a sequence of simpler operations including several multiplications. Therefore, hardware implementation of such operations and functions would call for high-speed parallel decimal multiplication. Such multiplication schemes are generally implemented as a sequence of three steps: partial product generation (PPG), partial product reduction (PPR), and the final carry- propagating addition. Parallel binary multipliers are used extensively in most of the binary floating point units for high performance. However, decimal multiplication is more difficult to implement due to the complexity in the generation of multiplicand multiples and the inefficiency of representing decimal values in system based on binary signals. These issues complicate the generation and reduction of partial products. The first implementation of a parallel decimal multiplier is described in [5]. Several different parallel decimal multiplier architectures are proposed in [3], which use new techniques for partial

product generation and reduction. Furthermore, some of these architectures were extended to support binary multiplication. Some concepts of [3] were applied in [6] to design decimal 4:2 compressor trees. All of the previous designs are combinational fixed point architectures. A pipelined IEEE 754-2008 compliant DFP multiplier based on an architecture from [3] was presented in [7].

The work is the major extension of the previous paper [3], which presented a new family of high-performance parallel decimal multipliers. In this paper, we deal with fully combinational decimal fixed-point architecture. We describe in some detail the methods for partial product generation and reduction proposed in [3] and introduce new techniques to reduce the latency and the hardware complexity of the previous designs. The paper is organized as follows: Section 2 outlines some previous representative work on decimal multiplication. In Section 3, we present the proposed multiplier architecture, signed-digit (SD) radix-10. The parallel generation of decimal partial products is detailed in Section 4. In Section 5, we describe the method for fast multioperand decimal carry-save addition and propose several tree architectures for an efficient reduction of partial products. Design decisions are supported by the area-delay model for static CMOS gates described in Section 6. In addition, we have synthesized SD radix-10 multiplier designs for 64-bit(16-digit) operands, using a 90 nm CMOS standard cells library. The expected result of area-delay figures shown in Section 6. Finally we summarize the main conclusions in Section 7.

Z _i	Z _i (BCD)	Z _i (5421)	Z _i (4221)	Z _i (5211)	Z _i (4311)	Z _i (3321)
0	0000	0000	0000	0000	0000	0000
1	0001	0001	0001	0001 0010	0001 0010	0001
2	0010	0001	0100 0010	0100 0011	0011	0010
3	0011	0011	0101 0011	0101 0110	0100	0100 1000 0011
4	0100	0100	0110 1000	0111	1000 0110 0101	1001 0101
5	0101	1000	0111 1001	1000	1001 0111 1010	1010 0110
6	0110	1001	1010 1100	1010 1001	1011	1100 1011 0111
7	0111	1010	1011 1101	1011 1100	1100	1101
8	1000	1011	1110	1110 1101	1110 1101	1110
9	1001	1100	1111	1111	1111	1111

TABLE-1 Decimal Codings

II. FIXED-POINT DECIMAL MULTIPLICATION

A digit Z_i of a decimal integer operand $Z = \sum_{i=0}^{d-1} Z_i 10^i$ is coded as a positive weighted 4-bit vector as

$$Z = \sum_{i=0}^3 z_{i,j} r_i \quad (1)$$

Where $Z_i \in [0,9]$ is the i th decimal digit, $z_{i,j}$ is the j th bit of the i th digit, and $r_j, j \geq 1$ is the weight of the j th bit. The previous expression represents a set of coded decimal number systems that includes BCD (with $r_j = 2^j$), shown in Table 1. The other decimal codes shown in Table 1 are used for representing different decimal operands, as required by the methods presented in this paper, and are referenced later. We refer to these codes by their weight bits as (r3r2r1r0) is denoted by Z_i (r3r2r1r0).

The multiplicand $X = \sum_{i=0}^{d-1} X_i 10^i$ and multiplier $Y = \sum_{i=0}^{d-1} Y_i 10^i$ are unsigned decimal integer d -digit BCD words. Fixed-point multiplication (both binary and decimal) consists of three stages; generation of partial products, reduction (addition) of partial products to two operands, and a final conversion (usually a carry propagate addition) to a nonredundant $2d$ -digit BCD representation $P = \sum_{i=0}^{2d-1} P_i 10^i$. Extension to decimal floating-point multiplication involves exponent addition, rounding of $P = X \times Y$ to fit the required precision, sign calculations, and exception detection and handling.

Decimal fixed-point multiplication is more complex than binary multiplication mainly for two reasons: the larger range of decimal digits ($[0,9]$), which increments the number of multiplicand multiples and the inefficiency of directly representing decimal values in systems based on binary logic using BCD (since only 9 out of the 16 possible 4-bit combinations represent a valid decimal digit). These issues complicate the generation and reduction of partial products. To improve the decimal multiplication performed the reduction of decimal partial product using some scheme for decimal carry-propagate addition such as direct decimal addition.

To reduce the contribution of the decimal corrections to the critical path, three different techniques for multioperand decimal carry-save addition were proposed in [4]. Two of them perform BCD corrections (+6 digit additions) using combinational logic and an array of binary carry-save adders (speculative adders), although a final correction is also required. A sequential decimal multiplier based on these techniques is presented in [8]. It uses BCD invalid combinations (overloaded BCD representation) to simplify the sum digit logic. The other approach (non-speculative adder [4]) uses a binary CSA tree followed by a single decimal correction. A recent proposal uses a binary carry-free tree adder and a subsequent binary to BCD conversion to add up to N d -digit BCD operands. An example of this architecture, implemented in a decimal parallel multiplier.

The another group of methods [1], [5] uses different topologies of 4-bit radix-10 carry-propagate adders to implement decimal carry-save addition. In [1], a serial multiplier is implemented using an array of radix-10 carry look-ahead adders (CLAs). A CSA tree using these radix-10 CLAs is implemented in the combinational decimal parallel multiplier proposed in [5]. To optimize the partial product reduction, they also use an array of decimal digit counters. The reduction of all decimal partial products in parallel requires the use of efficient multioperand decimal tree adders. We also implement multioperand decimal tree adders using a binary CSA tree, but with operands coded in decimal codings that are more efficient than BCD, namely (4221) or (5211). These multioperand decimal CSA trees are detailed in Section 5.

II. DECIMAL PARALLEL MULTIPLIER

In this section, we present a general overview of the Radix-10 architecture for d -digit (4d-bit) BCD decimal fixed-point parallel multiplication. This design is based on the techniques for partial product generation and reduction detailed in Sections 4 and 5, respectively. The code (4221) and (5211) is used instead of BCD to represent the partial product is the main feature of this architecture. This improves the reduction of decimal partial product with respect to other proposals, in terms of latency and area is expected.

3.1 SD Radix-10 Architecture

The architecture of the d -digit SD radix -10 multiplier is shown in Fig.1. The multiplier consists of the following stages : Generation of decimal partial products coded in (4221) (generation of multiplicand multiples and SD radix-10 encoding of the multiplier), reduction of partial products, and a final BCD carry-propagate addition.

The generation of the $d+1$ partial products is performed by an encoding of the multiplier into d SD radix-10 digits and an additional leading bit as described in Section 4.1. Each SD radix-10 digit controls a level of 5:1 muxes, which selects a positive multiplicand multiple (0, X, 2X, 3X, 4X, 5X) coded in (4221). The generation of these multiples is detailed in Section 4.3. To obtain each partial product, a level of XOR gates inverts the output bits of the 5:1 muxes when the sign of the corresponding SD radix-10 digit is negative.

Before being reduced, the $d+1$ partial products, coded in (4221), are aligned according to their decimal weights. Each p -digit column of the partial product array is reduced to two (4221) decimal digits using one of the decimal digit $p:2$ CSA trees described in Section 5.4. The number of digits to be reduced for each column varies from $p=d+1$ to $p=2$. Thus, the $d+1$ partial products are reduced to two 2d-digit operands S and H coded in (4221).

The final product is a 2d-digit BCD word given by $P=2H+S$. Before being added, S and H need to be processed. S is recoded from (4221) to BCD excess-6 (BCD value plus 6, which requires practically the same logical complexity as a recoding to BCD). The $H \times 2$ multiplication is performed in parallel with the recoding of S. This $\times 2$ block uses a (4221) to (5421) digit recoder (see Section 4.4) and a 1-bit wired left shift to obtain the operand 2H coded in BCD.

For the final BCD carry-propagate addition, we use a quaternary tree (Q-T) adder based on conditional speculative decimal addition. It has low latency and requires less hardware than other alternatives.

IV. DECIMAL PARTIAL PRODUCT GENERATION

We aim for a parallel generation of a reduced number of partial products coded in (4221) or (5211). This is achieved with the recoding of the d -digit BCD multiplier and the generation of a reduced and simple set of multiplicand multiples.

Fig 1. Combinational SD radix-10 architecture

We present a different schemes with good trade-offs between fast generation of partial products and the number of partial products generated. A minimally redundant SD radix-10 recoding of the multiplier (with digits in $\{-5, \dots, 0, \dots, 5\}$) produces only $d+1$ partial products but requires a carry-propagate addition to generate complex multiples 3X and -3X. Furthermore, the (4221) and (5211) codes are self complementing (see Section 5.1). Thus, an advantage of using this scheme, which use BCD multiples, is that the 9's complement of each digit can be obtained by inverting its bits. This simplifies the generation of the negative multiplicand multiples from the positive ones.

Fig 2. Partial product generation for SD radix-10.

4.1 SD Radix-10 Recoding

Fig. 2 shows the block diagram of the generation of one partial product using the SD radix-10 recoding. This recoding transforms a BCD digit $Y_i \in \{0, \dots, 9\}$ into an SD radix-10 $Y_{bi} \in \{-5, \dots, 5\}$. The value of the recoded digit Y_{bi} depends on the decimal value of Y_i and on a signal $ysi-1$ (sign signal) that indicates if Y_{i-1} is greater than or equal to 5. Thus, the d -digit BCD multiplier Y is recoded into the $d+1$ -digit SD radix-10 multiplier Y_b

$$= \sum_{i=0}^d Y_{bi} 10^i \text{ with } Y_{bd} = y_{sd-1} \in \{0, 1\}.$$

Each digit Y_{bi} generates a partial product $PP[i]$ selecting the proper multiplicand multiple coded in (4221). This is performed in a similar way to a modified Booth recoding: Y_{bi} is represented as five "hot one code" signals $\{y_{1i}, y_{2i}, y_{3i}, y_{4i}, y_{5i}\}$ and a sign bit y_{si} . These signals are obtained directly from the BCD multiplier digits Y_i using the following logical expression

$$y_{si} = y_i, 3 \vee y_i, 2 \cdot (y_i, 1 \vee y_i, 0),$$

$$y_{5i} = y_i, 2 \cdot y_i, 1 \cdot (y_i, 0 \oplus y_{si} - 1),$$

$$y_{4i} = y_{si} - 1 \cdot y_i, 0 \cdot (y_i, 2 \oplus y_i, 1) \vee y_{si} - 1 \cdot y_i, 2 \cdot y_i, 0,$$

$$y_{3i} = y_i, 1 \cdot (y_i, 0 \oplus y_{si} - 1),$$

$$y_{2i} = y_{si} - 1 \cdot y_i, 0 \cdot (y_i, 3 \vee y_i, 2 \cdot y_i, 1) \vee y_{si} - 1 \cdot y_i, 3 \cdot y_i, 0 \cdot y_i, 2 \oplus y_i, 1,$$

$$y_{1i} = y_i, 2 \vee y_i, 1 \cdot (y_i, 0 \oplus y_{si} - 1).$$

Symbols $\vee$, $\cdot$, and $\oplus$ indicate Boolean operators OR, AND, and XOR, respectively. The five “hot one code” signals are used as selection control signals for the 5:1 muxes to select the positive $d+1$ -digit multiples $\{0, X, 2X, 3X, 4X, 5X\}$. The generation of the positive multiples ($X, 2X, 3X, 4X, 5X$) coded in (4221) from the BCD multiplicand is detailed in Section 4.3. To obtain the correct partial product, the selected positive multiple is 10's complemented if ysi is one. This is performed simply by a bit inversion of the positive (4221) decimal-coded multiple using a row of XOR gates controlled by ysi . The addition of one *ulp* (unit in the last place) is performed enclosing a tail-encoded bit ysi (hot one) to the next significant partial product $PP[i+1]$, since it is shifted a decimal position to the left from $PP[i]$. To avoid a sign extension, and thus, to reduce the complexity of the partial product reduction tree, the partial product sign bits ysi are encoded at each leading position into two digits as

$$\begin{aligned} (PP[i]_{d+2}, PP[i]_{d+1}) &= \{(\overline{ys0}, ys0\ ys0\ ys0\ ys0)\}, i = 0 \\ &= \{(0, 111\ ysi)\}, 0 < i < d-1, \\ &= \{(0, 0000)\}, i = d-1. \end{aligned}$$

Therefore, each partial product $PP[i]$ is at most of $(d+3)$ -digit length.

4.2 Generation of Multiplicand Multiples

All the required decimal multiplicand multiples, except the $3X$ multiple, are obtained in a few levels of combinational logic using different digit recoders and performing different fixed m -bit left shifts ($Lmshift$) in the bit-vector representation of operands. The structure of these digit recoders is discussed in Section 4.3. Fig. 3 shows the block diagram for the generation of the positive multiplicand multiples $\{X, 2X, 3X, 4X, 5X\}$ for the SD radix-10 recoding.

Fig 3 (a).Generation of multiplicand multiples for SD radix-10

Fig 3(b) Waveforms of Multiplicand multiples

All these multiples are coded in (4221). The X BCD multiplicand is easily recoded to (4221) using the logical expressions.

$(w_{i,3}, w_{i,2}, w_{i,1}, w_{i,0}) = (x_{i,3} \vee x_{i,2}, x_{i,3}, x_{i,3} \vee x_{i,1}, x_{i,0}), (2)$ where $x_{i,j}$ and $w_{i,j}$ are input and output, respectively, the bits of the BCD and (4221) representations of X . The generation of multiples is as follows:

Multiple 2X. Each BCD digit is first recoded to the (5421) decimal coding shown in Table 1 (the mapping is unique). An L1shift is performed to the recoded multiplicand, obtaining the $2X$ multiple in BCD. Then, the $2X$ BCD multiple is recoded to (4221) using Expressions (2).

Multiple 4X. It is obtained as $2X \times 2$, where the $2X$ multiple is coded in (4221). The second $\times 2$ operation is implemented as a digit recoding from (4221) to code (5211), followed by an L1shift. The design of the (4221) to (5211) digit recoders is described in Section 4.3. The $\times 2$ operation, with input operands coded in (4221) or (5211), is also implemented in the decimal CSA trees used for partial product reduction, and therefore, it is more detailed in Section 5.1.

Multiple 5X. It is obtained by a simple L3shift of the (4221) recoded multiplicand, with resultant digits coded in (5211). Then, a digit recoding from (5211) to (4221) is performed (see Section 4.3). Fig. 4 shows an example of this operation.

Multiple 3X. It is evaluated by a carry-propagate addition of BCD multiples X and $2X$ in a d -digit BCD adder. The BCD sum digits are recoded to (4221) as indicated by Expression (2). The latency of

the partial product generation for the SD radix-10 scheme is constrained by the generation of $3X$.

		$\times 10^2$	$\times 10^1$	$\times 10^0$
$X(4221)$	25	4221	4221	4221
	$L3_{\text{shift}}$	0000	0010	1001
$5X(5211)$	125	5211	5211	5211
	$\downarrow \times 5$	0001	0100	1000
Digit recoding	$\downarrow$			
$5X(4221)$	125	4221	4221	4221
		0001	0100	1001
Multiplication by 5				

Fig 4 : Calculation of $\times 5$ for decimal operands coded in (4221)

Z_i	0	1	2	3	4
$Z_i(4221s)$	0000	0001	0010	0011	1000
$Z_i(5211s)$	0000	0001	0100	0101	0111
Z_i	5	6	7	8	9
$Z_i(4221s)$	1001	1010	1011	1110	1111
$Z_i(5211s)$	1000	1001	1100	1101	1111

Table 2. Selected Decimal Codes for the Recoded Digits

4.3 Implementation of Digit Recoders

Digit recoders are used to compute the decimal multiplicand multiples (Section 4.2) and in the reduction of partial products (Section 5) to compute $\times 2^n$ ($n > 0$) operations. The logical implementation of digits recoders for BCD, BCD excess-6, and (5421) decimal codes is straight forward, since there is only a mapping of decimal digits to these codes (each decimal digit has a single 4-bit representation). However, due to the redundancy of (4221) and (5211) decimal codes, there are several choices for the digit recoding to (4221) or (5211). The sixteen 4-bit vectors of a coding can be mapped (recoded) into different subsets of 4-bit vectors of the other decimal coding representing the same decimal digit. These subsets of the (4221) and (5211) codes are also decimal codings.

Among all the subsets analyzed, the nonredundant decimal codes (4221s) and (5211s) (subsets of ten 4-bit vectors), shown in Table 2, present interesting properties. In particular, these codes verify.

$$2Z(4221s) = L1_{\text{shift}}[Z(5211s)], \quad (3)$$

that is, after shifting 1 bit to the left an operand Z represented in (5211s), the resultant bit-vector represents the decimal value of $2Z$ coded in (4221s). This fact simplifies the implementation of $\times 2^n$ operations for $n > 1$. Specifically, for a decimal operand $Z(4221)$, $Z \times 2^n$ is implemented by a first level of $Z_i(4221)$ to $Z_i(5211s)$ digit recoders followed by $n-1$ levels of $Z_i(4221s)$ to $Z_i(5211s)$ digit recoders. The output of each level of digit recoders is shifted 1 bit to the left such that the most significant

bit of each (5211s) digit (weight 5) is shifted out to the next decimal position (weight 10).

Moreover, in some cases, the $\times 2$ may be simplified. In particular, the recoding given by Expression (2) maps the BCD representation into the subset (4221s). Therefore, the subsequent $\times 2$ operations in Fig.3 is implemented using a level of simpler (4221s) to (5211s) digit recoders. A (4221) to (5211s) digit recoder has a hardware complexity of about 27 NAND2 gates, and its critical path has (roughly) the delay of a full adder. The (4221s) to (5211s) digit recoder has a simpler hardware complexity (about 19 NAND2 gates) with 25 percent less latency.

$$Z_i(5211) = z_{i,3}4 + z_{i,2}2 + z_{i,1}2 + z_{i,0}, \quad (4)$$

Additionally, the inverse digit recoding (from (5211) to (4221)) is easily implemented using a single full adder, since

with $z_{i,1}^*2 + z_{i,0}^* = (z_{i,3} + z_{i,1} + z_{i,0}) \leq 3$. This recoder is used to generate the $\times 5$ multiple for the (4221) coding and in mixed (4221/5211) multioperand CSAs (Section 5.5) to convert a (5211) decimal-coded operand into the equivalent (4221) coded one.

V. PARTIAL PRODUCT REDUCTION

Section 5.1 describes the partial product arrays generated by the SD radix-10 encoding. Each column of p digits is reduced to two digits by means of a decimal digit $p:2$ CSA tree. Also, decimal carries are passed between adjacent digit columns. In Section 5.2, we present the set of preferred decimal codings and the method for decimal carry-save addition. We propose the use of the (4221) and (5211) decimal codings instead of BCD for an efficient implementation of decimal carry-save addition with binary CSAs or full adders. The use of these codes avoids the need for decimal corrections, so we only need to focus on the $\times 2$ decimal multiplications. The implementation of decimal $3:2$ CSAs for the proposed codings is also described in Section 5.2. We present the Decimal $p:2$ CSA Trees for Digits Coded in (4221) Operands in Section 5.3

5.1 Partial Product Arrays

As we detailed in Section 4.1, the SD radix-10 architecture produces $d+1$ partial products coded in (4221) of $d+3$ -digit length. Before being reduced, the $d+1$ partial products $PP[i]$ are aligned according to their decimal weights by $4i$ -bit wired left shifts ($PP[i] \times 10^i$). The resultant partial product array for 16-digit input operands is shown in Fig. 5. In this case, the number of digits to be reduced varies from $p=17$ to $p=2$. In particular, the highest columns can be

reduced with the area-optimized or delay-optimized decimal 17:2 CSA trees presented in Section 5.3.

5.2 Method for Decimal Carry-Save Addition

Among all the possible decimal codes defined by Expression (1) in Section 2, there is a family of codes suitable for simple decimal carry-save addition. This family of decimal codings verifies that the sum of their weight bits is 9, that is,

$$\sum_{j=0}^3 r_j = 9, \quad (5)$$

which includes the (4221), (5211), (4311), and (3321) codes, shown in Table 1. Some of these decimal codings are already known, but we use them in a different context, to design components for decimal carry-save arithmetic. Moreover, they are redundant codes, since two or more different 4-bit vectors may represent the same decimal digit.

Fig. 6a shows the implementation of a decimal 3:2 CSA for digits coded in (4221) using a 4-bit binary 3:2 CSA. The weight bits in Fig. 9a are placed in brackets above each bit column. The 4-bit binary 3:2 CSA adds three decimal digits (A_i , B_i , C_i), coded in (4221), and

Fig 5 Partial product arrays generated for 16-digit operands. SD radix-10 architecture.

Fig.6 Proposed decimal 3:2 CSA for operands coded in (4221). (a) Decimal digit (4-bit) 3:2 CSA. (b) Full Adder.

produces a decimal sum digit (S_i) and a carry digit H_i coded in (4221), such that

$$A_i + B_i + C_i = S_i + 2 \times H_i.$$

In order to obtain $(2H)_i$, H_i is first recoded to W_i using the (4221) to (5211s) digit recoder introduced in Section 4.3. The output of the digit recoder (W_i) is then left shifted by 1 bit position ($L1shift[W_i]$). A decimal carry output $w_{i,3}$ is passed to the next significant digit position, while a decimal carry in $w_{i-1,3}$ comes from the previous. Since the recoder is placed in the carry path, a full adder implementation with a fast carry output, such as the one shown in Fig. 9b, reduces the total critical path delay.

Fig 7. Implementation of $\times 2$ block.

5.3 Decimal p:2 CSA Trees for Decimal Coded in (4221)

Operands.

A decimal digit p:2 CSA tree reduces p ($p \geq 3$) input digits $Z_i[l]$ (with weight 10^l) coded in (4221) into two decimal digits H_i and S_i . In addition, several decimal carry outputs are generated to the next significant decimal position (10^{l+1}) and a certain number of decimal carry inputs come from the previous position (10^{l-1}).

These decimal p:2 CSA trees are designed as follows:

- For $p < 7$, the input digits $Z_i[l]$ are reduced in a first level of binary 3:2 CSAs. Each carry output digit is multiplied by 2 before being reduced in the next level of the binary 3:2 CSA tree. Each $\times 2$ operation produces a decimal carry output to the next significant digit column of the partial product array. The slowest outputs are connected to fast inputs of the next binary 3:2 CSA level to balance the total delay of the different paths (an F indicates the fast input). We use the full adder configuration of Fig. 6b to minimize the critical path delay of the CSA tree. The digit blocks labelled $\times 2$ consist of a (4221) to (5211s) digit recoder with the outputs (for 4221 coded operands) 1-bit left shifted, as shown in Fig. 7.

The most significant output bit ($w_{i,3}$) represents a decimal carry to the next digit column. To simplify the diagrams of the different decimal $p:2$ CSA trees, the carries passed between adjacent digit columns ($w_{i,3}$, $w_{i-1,3}$) are not represented. The carry output H_i must be multiplied by 2 before being assimilated with the sum output S_i .

- For $p \geq 7$, we follow different strategies to obtain area-optimized or delay-optimized implementations. For area-optimized implementations, the input digits $Z_i[l]$ are reduced in a first level of binary 3:2 CSAs. Each intermediate operand is associated with a multiplicative factor power of 2. Operands with the same factor are reduced in a binary 3:2 CSA before being multiplied by this factor, that is,

$$2^n A + 2^n B + 2^n C = 2^n (A + B + C) = 2^n S + 2^{n+1} H. \quad (6)$$

This reduces the hardware complexity since the overall number of $\times 2$ operations is reduced. An area-optimized decimal 17:2 CSA tree for operands coded in (4221) is shown in Fig. 8a. The $\times 4$ and $\times 8$ digit blocks produce two and three decimal carry outs to the next significant digit column of the partial product array. They are implemented using a cascade configuration of $\times 2$ blocks as shown in Fig. 9. The critical path delay is reduced by balancing the delay of the different paths. For this purpose, the intermediate operands with higher multiplicative factors are multiplied in parallel with the reduction of the other intermediate operands using binary 3:2 CSAs. The delay-optimized 17:2 CSA tree in Fig. 8b has more hardware complexity (equivalent to two $\times 2$ blocks more) but the critical path is slightly faster (about 1 XOR delay faster). Its delay is of about six levels of binary 3:2 CSAs and three levels of digit recoders. The blocks labelled 9:4 and 8:4 represent the decimal digit adders.

VI. SYNTHESIS RESULT

The 16-digit SD radix-10 (Fig 1) combinational multiplier have been synthesized using Modelsim SE6.5c. For partial product reduction, the SD radix-10 multiplier implements area-optimized decimal $p:2$ CSA trees, similar to the decimal 17:2 CSA tree of Fig 8a.

VII. CONCLUSION

In this paper, we have presented one technique to implement decimal parallel multiplication in hardware. We propose the SD encoding for the multiplier that lead to fast parallel and simple generation of partial products. We have developed a decimal carry-save algorithm based on (4221) and (5211) decimal encoding for partial product

reduction. We have proposed architecture for decimal SD radix-10 multiplication.

Fig 8. Proposed decimal 17:2 CSAs. (a) Area Optimized tree. (b) Delay-Optimized tree.

Fig 9. Implementation of $\times 8$ Multiplication for two adjacent columns.

REFERENCES

- [1]. M. A. Erle and M. J. Schulte. Decimal multiplication via carry-save addition. In Proc. IEEE Int'l Conference on Application-Specific Systems, Architectures, and Processors, pages 348–358, June 2003.
- [2]. M. A. Erle, E. M. Schwarz, and M. J. Schulte. Decimal multiplication with efficient partial product generation. In Proc. IEEE 17th Symposium on Computer Arithmetic, pages 21–28, June 2005.
- [3]. Va'zquez, E. Antelo, and P. Montuschi, "A New Family of High Performance Parallel Decimal Multipliers," Proc. 18th IEEE Symp. Computer Arithmetic, pp. 195-204, June 2007.
- [4]. R. D. Kenney and M. J. Schulte. High-speed multioperand decimal adders. IEEE Trans. on Computers, 54(8):953–963, Aug. 2005.
- [5]. T. Lang and A. Nannarelli. A radix-10 combinational multiplier. In Proc. 40th Asilomar Conference on Signals, Systems, and Computers, pages 313–317, Oct. 2006.
- [6]. I.D. Castellanos and J.E. Stine, "Compressor Trees for Decimal Partial Product Reduction," Proc. 18th ACM Great Lakes Symp. VLSI, pp. 107-110, Mar. 2008.
- [7]. B.J. Hickman, A. Krioukov, M.A. Erle, and M.J. Schulte, "A Parallel IEEE P754 Decimal Floating-Point Multiplier," Proc. 25th IEEE Conf. Computer Design, pp. 296-303, Oct. 2007.
- [8]. R.D. Kenney, M.J. Schulte, and M.A. Erle, "High-Frequency Decimal Multiplier," Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, pp. 26-29, Oct. 2004.