Improved Design of High-Performance Parallel Decimal Multipliers

Álvaro Vázquez, Elisardo Antelo, and Paolo Montuschi

Version: AM (Accepted Manuscript)

How to cite:

Alvaro Vazquez, Elisardo Antelo and Paolo Montuschi, “Improved Design of High-Performance Parallel Decimal Multipliers”, IEEE Transactions on Computers, vol. 59, no. 5, pp. 679-693, May 2010.

doi: 10.1109/TC.2009.167

Copyright information:

© 2010 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Improved Design of High-Performance Parallel Decimal Multipliers

Alvaro Vazquez, Elisardo Antelo, *Member, IEEE,* and Paolo Montuschi, *Senior Member, IEEE*

Abstract—The new generation of high-performance decimal floating-point units (DFUs) is demanding efficient implementations of parallel decimal multipliers. In this paper we describe the architectures of two parallel decimal multipliers. The parallel generation of partial products is performed using signed-digit radix-10 or radix-5 recodings of the multiplier and a simplified set of multiplicand multiples. The reduction of partial products is implemented in a tree structure based on a decimal multioperand carry-save addition algorithm that uses unconventional (non BCD) decimal coded number systems. We further detail these techniques and present the new improvements to reduce the latency of the previous designs which include: optimized digit recoders for the generation of 2^n-tuples (and 5-tuples), decimal CSAs (carry-save adders) combining different decimal coded operands and carry-free adders implemented by special designed bit counters. Moreover, we detail a design methodology that combines all these techniques to obtain efficient reduction trees with different area and delay tradeoffs for any number of partial products generated. Evaluation results for 16-digit operands show that the proposed architectures have interesting area-delay figures compared to conventional Booth radix-4 and radix-8 parallel binary multipliers and outperform the figures of previous alternatives for decimal multiplication.

Index Terms—Decimal multiplication, parallel multiplication, decimal carry-save addition, decimal codings.

1 Introduction

Providing hardware support for decimal floating-point arithmetic (DFP) is becoming a topic of interest. Although software DFP implementations [4], [6] satisfy the precision requirements, they are about an order of magnitude slower than hardware implementations [14], [34] and could not satisfy the high performance demands of future financial, commercial and user-oriented applications [5]. Furthermore, there have been some efforts in defining a standard for decimal arithmetic [7]. Specifically, the revision of the IEEE 754 Standard for Floating-Point Arithmetic (IEEE 754-2008) [17] incorporates specifications for DFP arithmetic that can be implemented in software, hardware or in a combination of both. For instance, the IBM z9 architecture [10] implements DFP using microcode and performs the most basic tasks in dedicated hardware. The IBM Power6 microprocessor [11], oriented to workstations and servers, and the IBM z10 mainframe processor [27], already include fully compliant IEEE 754-2008 DFUs.

The new generation of high-performance DFUs is demanding more efficient implementations of decimal multiplication. Although this is an important and frequent operation, current hardware implementations suffer from lack of performance. Parallel binary multipliers [22], [26] 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 systems based on binary signals. These issues complicate the generation and reduction of partial products. Thus, while decimal adders are implemented in a parallel fashion and are almost as efficient as binary ones, commercial implementations of decimal multipliers are sequential [1], [2], [11], [23]. These implementations are based on iterative algorithms for decimal integer multiplication [21], [23] and therefore present low performance. Recently, several techniques related to the generation of partial products [13], [31] or providing improvements of sequential decimal multipliers [12], [15], [19] have been proposed. Other recent works [8], [9], [18] have proposed new techniques to speed-up multioperand BCD addition using tree-like (parallel) structures.

The first implementation of a parallel decimal multiplier is described in [20]. Several different parallel decimal multiplier architectures are proposed in [33], which use new techniques for partial product generation and reduction. Furthermore, some of these architectures were extended to support binary multiplication. Some concepts of [33] were applied in [3] 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 [33] was presented in [16].

This work is a major extension of a previous paper [33] which presented a new family of high-performance parallel decimal multipliers. In this paper, we deal with fully combinational decimal fixed-point architectures. We describe in some detail the methods for partial product generation and reduction proposed in [33] and introduce new techniques to reduce the latency and the hardware complexity of the previous designs. The paper is organized as follows. Section

- *A. Vazquez is with the Laboratoire de l'Informatique du Parallelisme, ENS-Lyon, 69364 Lyon, France.*
  E-mail: alvaro.vazquez.alvarez@ens-lyon.fr
- *E. Antelo is with the Department of Electrical and Computer Engineering, Univ. of Santiago de Compostela, 15782 Santiago de Compostela, Spain.*
  E-mail: elisardo.antelo@usc.es
- *P. Montuschi is with Dept. of Computer Engineering, Politecnico di Torino, 10129 Torino, Italy.*
  E-mail: paolo.montuschi@polito.it

2 outlines some previous representative work on decimal multiplication. In Section 3 we present the two proposed multiplier architectures, SD (signed-digit) radix-10 and SD radix-5. 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 both SD radix-10 and SD radix-5 multiplier designs for 64-bit (16-digit) operands, using a 90nm CMOS standard cells library. We provide the resulting area-delay figures in Section 6. In Section 7 we compare the two proposed designs with some representative binary and decimal multipliers. We also perform a comparison among the different multioperand decimal tree adders. We finally summarize the main conclusions in Section 8.

2 An overview of 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

$$Z_i = \sum_{i=0}^{3} z_{i,j} \; r_j \tag{1}$$

where $Z_i \in [0,9]$ is the ith decimal digit, $z_{i,j}$ is the jth bit of the ith digit and $r_j \geq 1$ is the weight of the jth 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 $(r_3 r_2 r_1 r_0)$. The 4-bit vector that represents the decimal digit Z_i in a decimal code $(r_3 r_2 r_1 r_0)$ is denoted as $Z_i(r_3 r_2 r_1 r_0)$.

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 non redundant $2d$-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.

Proposed methods for the generation of decimal partial products follow two approaches. The first alternative performs a digit by digit multiplication of the input operands, using digit-by-digit lookup table methods [21], [31]. In a recent work [13], a magnitude range reduction of the operand digits by a signed-digit radix-10 recoding (from [0,9] to [-5,5]) is suggested. This recoding of both operands speeds-up and simplifies the generation of partial products. Then, signed-digit partial products are generated using simplified tables and combinational logic. This class of methods is only suited for serial implementations, since the high hardware demands make them impractical for parallel partial product generation (see [33]). The second approach generates and stores all the required multiplicand multiples [24]. Next, multiples are distributed to the reduction stage through multiplexers controlled by the BCD multiplier digits ($[0,9]$). This approach requires several wide decimal carry-propagate additions to generate some complex BCD multiplicand multiples (3X,6X,7X,8X,9X). In [2], only even multiples $\{2X, 4X, 6X, 8X\}$ are computed and stored. Odd multiples $\{3X, 5X, 7X, 9X\}$ are obtained on demand. A reduced set of BCD multiples $\{X, 2X, 4X, 5X\}$ is precomputed in [12] without a carry propagation. All the multiples can be obtained from the sum of two elements of this set. In [20] each multiplier digit is recoded as $Y_i = Y^U \; 5{+}Y^L$, with $Y^U \in \{0,1,2\}$ and $Y^L \in \{-2,-1,0,1,2\}$. The $2X$ and $5X$ multiples are computed in few levels of combinational logic. Negative multiples require an additional 10's complement operation.

First attempts to improve decimal multiplication performed the reduction of decimal partial products using some scheme for decimal carry propagate addition such as direct decimal addition [25]. Proposals to perform the reduction of decimal partial products using carry-free addition were suggested in [21] (carry-save) and [13], [28], [30] (signed-digit).

Decimal carry-save addition methods use two BCD words to represent sum and carry [18], [19], [21], [23] or a BCD sum word and a carry bit per digit [12], [20]. The first group implements decimal addition mixing binary CSAs with combinational logic for decimal correction. In [23] a scheme of two levels of 3:2 binary CSAs is used to add the partial products iteratively. Since it uses BCD to represent decimal digits, a digit addition of +6 or +12 (modulo 16) is required to obtain the decimal carries and to correct the sum digit. In order to reduce the contribution of the decimal corrections to the critical path, three different techniques for multioperand decimal carry-save addition were proposed in [18]. 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 [19]. It uses BCD invalid combinations (overloaded BCD representation) to simplify the sum digit logic. The other approach (non-speculative adder [18]) uses a binary CSA tree followed by a single decimal correction. Among these proposals, the non-speculative adders present the best area-delay figures and are suited for tree topologies. A recent proposal [8] 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, can be found in [9].

The second group of methods [12], [20] uses different

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.

topologies of 4-bit radix-10 carry-propagate adders [25] to implement decimal carry-save addition. In [12] a serial multiplier is implemented using an array of radix-10 CLAs (carry lookahead adders). A CSA tree using these radix-10 CLAs is implemented in the combinational decimal parallel multiplier proposed in [20]. 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. Among the different schemes, the most promising ones for fast parallel addition seem to be those using binary CSA trees or some parallel network of full adders [8], [18], due to their faster and simpler logic cells (full adders against SD adder cells or radix-10 CLAs). These methods assume that decimal digits are coded in BCD. However, BCD is highly inefficient for implementing decimal carry-save addition using binary arithmetic, because the need to correct the invalid 4-bit combinations (those not representing a decimal digit). The previous methods use different schemes to perform these BCD corrections. Moreover, the BCD carry digit must be multiplied by 2, which requires additional logic. 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.

3 Decimal Parallel Multipliers

In this Section we present a general overview of the proposed architectures for d-digit ($4d$-bit) BCD decimal fixed-point parallel multiplication. These designs are based on the techniques for partial product generation and reduction detailed in Section 4 and Section 5 respectively. The main feature of these architectures is the use of codes (4221) and (5211), instead of BCD, to represent the partial products. This improves the reduction of decimal partial products with respect to other proposals, in terms of both area and latency.

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.

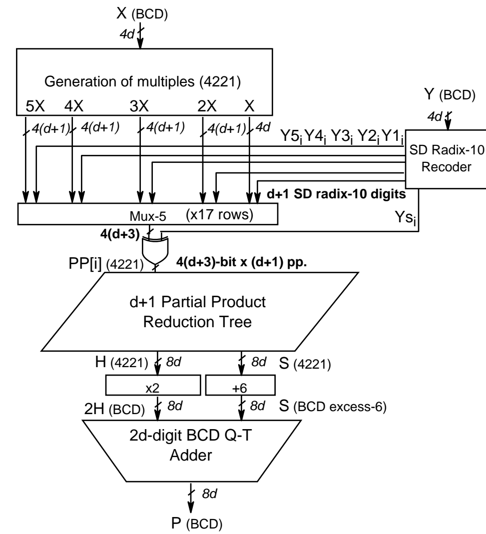


Fig. 1. Combinational SD radix-10 architecture.

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

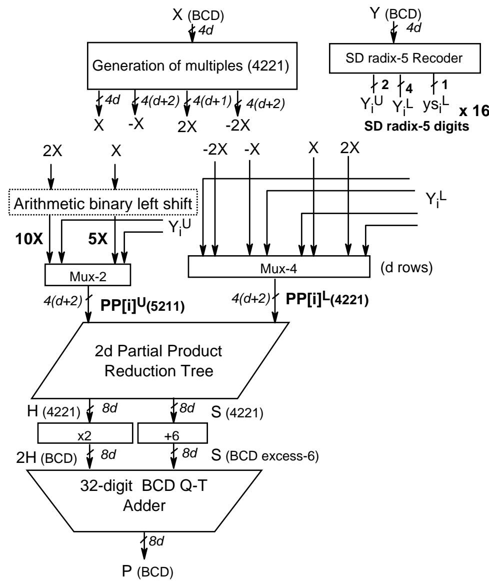


Fig. 2. Combinational SD radix-5 architecture.

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[1].

For the final BCD carry propagate addition we use a quaternary tree (Q-T) adder based on conditional speculative decimal addition [32]. It has low latency (about 10% more than the fastest binary adders) and requires less hardware than other alternatives.

3.2 SD radix-5 architecture

The dataflow of the d-digit SD radix-5 architecture is shown in Fig. 2. The multiplier consists of the following stages: generation of decimal partial products (generation of multiplicand multiples and SD radix-5 encoding of the multiplier), reduction of partial products and a final BCD carry-propagate addition. SD radix-5 recoding, described in Section 4.2, generates $2d$ decimal partial products, half coded in (4221) and the other half in (5211). This improved scheme only requires the generation of simple multiplicand multiples $\{-2X, -X, X, 2X\}$ coded in (4221), as shown in Section 4.3. The reduction of the aligned partial products is carried out using the mixed (4221/5211) decimal digit p:2 CSA trees ($2 \leq p \leq 2d$) described in Section 5.5. As in the SD radix-10 architecture, the $2d$-digit operands S and H are processed before being assimilated in the $2d$-digit BCD carry-propagate adder.

4 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. We present two 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 $\{-5, \ldots, 0, \ldots, 5\}$) produces only $d+1$ partial products but requires a carry propagate addition to generate complex multiples $3X$ and $-3X$. A second scheme, named SD radix-5 recoding, encodes each BCD digit Y_i of the multiplier into two digits $Y_i^U \in \{0, 1, 2\}$ $Y_i^L \in \{-2, -1, 0, 1, 2\}$, such that $Y_i = Y_i^U\ 5 + Y_i^L$. It generates $2d$ partial products (2 digits per radix-10 digit), but all multiplicand multiples are produced in a few levels of combinational logic. Furthermore, the (4221) and (5211) codes are self-complementing (see Section 5.1). Thus, an advantage with respect to previous schemes, 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. In addition, the previous methods based on the decomposition $Y_i = Y_i^U\ 5 + Y_i^L$ [20], [33] require combinational logic to generate the $5X$ multiple. We use mixed (4221/5211) decimal codings to remove this logic.

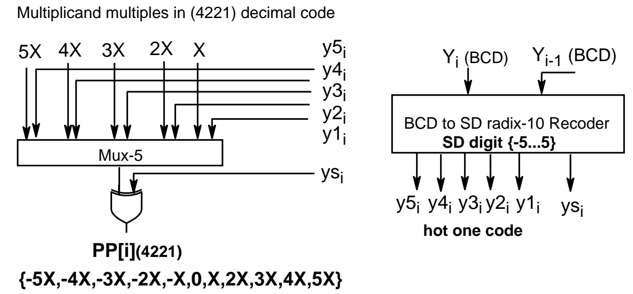


Fig. 3. Partial product generation for SD radix-10.

4.1 SD radix-10 recoding

Fig. 3 shows the block diagram of the generation of one partial product using the SD radix-10 recoding. This recoding transforms a BCD digit $Y_i \in \{0, \ldots, 9\}$ into a SD radix-10 $Yb_i \in \{-5, \ldots, 5\}$. The value of the recoded digit Yb_i depends on the decimal value of Y_i and on a signal ys_{i-1} (sign signal) that indicates if Y_{i-1} is greater or equal than 5. Thus, the d-digit BCD multiplier Y is recoded into the $d+1$-digit SD radix-10 multiplier $Yb = \sum_{i=0}^{d} Yb_i\ 10^i$ with $Yb_d = ys_{d-1} \in \{0, 1\}$.

Each digit Yb_i 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: Yb_i is represented as five 'hot one code' signals $\{y1_i, y2_i, y3_i, y4_i, y5_i\}$ and a sign bit ys_i. These signals are obtained directly from the BCD multiplier digits Y_i using the following logical expressions:

$$\begin{aligned} ys_i &= y_{i,3} \vee y_{i,2} \cdot (y_{i,1} \vee y_{i,0}) \\ y5_i &= y_{i,2} \cdot \overline{y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1}) \\ y4_i &= ys_{i-1} \cdot y_{i,0} \cdot (y_{i,2} \oplus y_{i,1}) \vee \overline{ys_{i-1}} \cdot y_{i,2} \cdot \overline{y_{i,0}} \end{aligned}$$

1. Note that this 1-bit left shift is equivalent to multiply the binary weights of (5421) by 2, such that the resulting digits are coded in BCD.

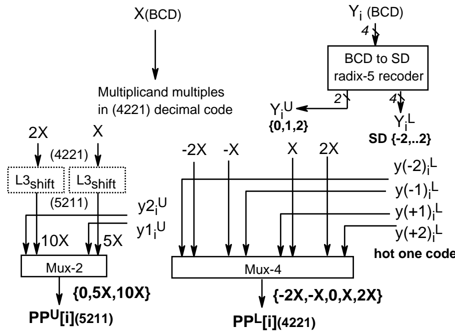


Fig. 4. Partial product generation for SD radix-5.

$$
\begin{aligned}
y3_i &= y_{i,1} \cdot (y_{i,0} \oplus ys_{i-1}) \\
y2_i &= \overline{ys_{i-1}} \cdot \overline{y_{i,0}} \cdot (y_{i,3} \vee \overline{y_{i,2}} \cdot y_{i,1}) \\
&\quad \vee ys_{i-1} \cdot \overline{y_{i,3}} \cdot y_{i,0} \cdot \overline{y_{i,2} \oplus y_{i,1}} \\
y1_i &= \overline{y_{i,2} \vee y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1})
\end{aligned}
$$

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 ys_i 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 ys_i. The addition of one ulp (unit in the last place) is performed enclosing a tail encoded bit ys_i (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 ys_i are encoded at each leading position into two digits as

$$
(PP[i]_{d+2}, PP[i]_{d+1}) = \begin{cases} (\overline{ys_0}, ys_0\ ys_0\ ys_0\ ys_0) & i=0 \\ (0, 111\overline{ys_i}) & 0 < i < d-1 \\ (0, 0000) & i = d-1 \end{cases}
$$

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

4.2 SD radix-5 recoding

Fig. 4 shows the diagram for partial product generation using the SD radix-5 recoding scheme. Each BCD digit of the multiplier is encoded into two digits $Y_i^U \in \{0, 1, 2\}$ and $Y_i^L \in \{-2, -1, 0, 1, 2\}$, so that $Y_i = Y_i^U \cdot 5 + Y_i^L$. SD radix-5 'hot one code' selection signals are obtained from the BCD input digits using the following equations:

$$
(Y_i^U) \quad \begin{cases} y2_i^U = y_{i,3} \\ y1_i^U = y_{i,2} \vee y_{i,1} \cdot y_{i,0} \end{cases}
$$

$$
(Y_i^L) \quad \begin{cases} y(+2)_i^L = y_{i,1} \cdot (y_{i,2} \cdot y_{i,0} \vee \overline{y_{i,2}} \cdot \overline{y_{i,0}}) \\ y(+1)_i^L = \overline{y_{i,3}} \cdot \overline{y_{i,2}} \cdot \overline{y_{i,1}} \cdot y_{i,0} \vee y_{i,2} \cdot y_{i,1} \cdot \overline{y_{i,0}} \\ y(-1)_i^L = y_{i,3} \cdot y_{i,0} \vee y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}} \\ y(-2)_i^L = y_{i,3} \cdot \overline{y_{i,0}} \vee \overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0} \end{cases}
$$

Each multiplier digit Y_i generates two partial products $PP[i]^U$ and $PP[i]^L$. Therefore, this scheme generates $2d$ partial products for a d-digit multiplier. The advantage of this recoding is that it uses a simple set of multiplicand multiples $\{-2X, -X, X, 2X\}$ coded in (4221). This decimal partial product generation is comparable in latency to binary Booth radix-4, due to a faster generation of multiples, as detailed in Section 4.3.

Moreover, the generation of $PP[i]^U$ (positive) only requires multiples $\{X, 2X\}$. To obtain the correct value of $PP[i]^U$, the multiples selected by Y_i^U must be first multiplied by 5. This is performed by shifting 3 bits to the left the bit vector representation of the (4221) coded multiples $\{X, 2X\}$, producing respectively the multiples $\{5X, 10X\}$ but coded in (5211)[2]. We denote by Lm_{shift} a left arithmetic binary shift of m bits, implemented with fixed wiring.

The negative multiples $\{-X, -2X\}$ are obtained by bit inverting the multiples $\{X, 2X\}$, coded in (4221), and adding an ulp as a hot one in the corresponding partial product. The sign bits ys_i^L, given by

$$
ys_i^L = y_{i,3} \vee y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}} \vee \overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0} \tag{2}
$$

are encoded to the left of $PP[i]^L$ and $PP[0]^U$ as

$$
\begin{aligned}
PP[i]_{d+1}^L &= \begin{cases} (1, 1, 1, \overline{ys_i^L}) & If (0 \le i < d-1) \\ (0, 0, 0, 0) & If (i = d-1) \end{cases} \\
PP[0]_{d+1}^U &= (0, 0, 0, ys_0^L)
\end{aligned} \tag{3}
$$

The hot ones produced by the 10's complement of the partial products, $(0, 0, 0, ys_i^L)$, are just placed in the least significant digit of $PP[i]^U$, $PP[i]_0^U$, which have a value of 0 or 5 coded in (5211). The $2d$ partial products generated are at most of $d+2$-digit length, d of them coded in (5211) ($PP[i]^U$) and the other half in (4221) ($PP[i]^L$).

4.3 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 (Lm_{shift}) in the bit-vector representation of operands. The structure of these digit recoders is discussed in Section 4.4.

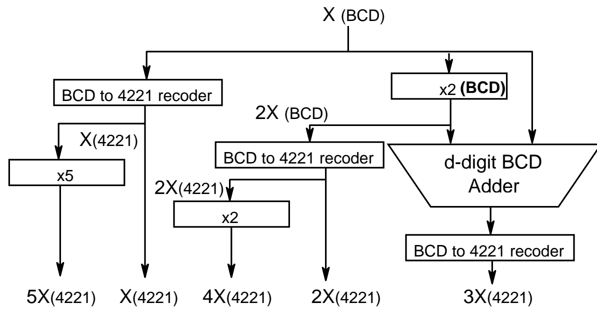
Fig. 5(a) shows the block diagram for the generation of the positive multiplicand multiples $\{X, 2X, 3X, 4X, 5X\}$ for the SD radix-10 recoding. All these multiples are coded in (4221). The X BCD multiplicand is easily recoded to (4221) using the logical expressions

$$
(w_{i,3},\ w_{i,2},\ w_{i,1},\ w_{i,0}) = (x_{i,3} \vee x_{i,2},\ x_{i,3},\ x_{i,3} \vee x_{i,1},\ x_{i,0}) \tag{4}
$$

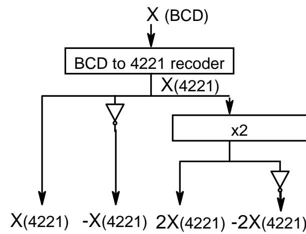
where $x_{i,j}$ and $w_{i,j}$ are, respectively, the bits of the BCD and (4221) representations of X. The generation of multiples is as follows:

Multiple $2X$: Each BCD digit is first recoded to the (5421) decimal coding shown in Table 1 (the mapping is unique). A

2. Shifting three bits to the left a (4221) decimal coded bit vector is equivalent to multiply its binary weights by 5, obtaining a (5211) coded operand.



(a) Multiples for SD radix-10.



(b) Multiples for SD radix-5.

Fig. 5. Generation of multiplicand multiples.

		x10^2	x10^1	x10^0
		4221	*4221*	*4221*
X(4221)	25	0000	0010	1001
$L3_{shift}$ ↓ x5		*5211*	*5211*	*5211*
5X(5211)	125	0001	0100	1000
Digit recoding ↓		*4221*	*4221*	*4221*
5X(4221)	125	0001	0100	1001

Multiplication by 5

Fig. 6. Calculation of $\times 5$ for decimal operands coded in (4221).

$L1_{shift}$ is performed to the recoded multiplicand, obtaining the $2X$ multiple in BCD. Then, the $2X$ BCD multiple is recoded to (4221) using Expressions (4).

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 a $L1_{shift}$. The design of the (4221) to (5211) digit recoders is described in Section 4.4. 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 $L3_{shift}$ of the (4221) recoded multiplicand, with resultant digits coded in (5211). Then, a digit recoding from (5211) to (4221) is performed (see Section 4.4). Fig. 6 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 (implemented as a quaternary-tree decimal adder [32]). The BCD sum digits are recoded to (4221) as indicated by Expression

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

The generation of (4221) decimal coded multiples $\{-2X, -X, X, 2X\}$ for the SD radix-5 recoding is shown in Fig. 5(b). The BCD multiplicand is first recoded to (4221) using Expressions (4). The $2X$ multiple is implemented as a digit recoding from (4221) to (5211) followed by a $L1_{shift}$. The negative multiples $\{-X, -2X\}$, coded in (4221), are obtained inverting the bits of the (4221) decimal coded positive multiples and encoding the sign as described in Section 4.2.

4.4 Implementation of digit recoders

The design of efficient digit recoders is a critical issue, due to their high impact on the performance and area of the whole multiplier. Digit recoders are used to compute the decimal multiplicand multiples (Section 4.3) 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 straightforward, 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 non-redundant decimal codes (4221s) and (5211s) (subsets of ten 4-bit vectors), shown in Table 2, present interesting properties. In particular, these codes verify

$$2Z(4221s) = L1_{shift}[Z(5211s)] \quad (5)$$

that is, after shifting one 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 (4) maps the BCD representation into the subset (4221s). Therefore, the subsequent $\times 2$ operations in Fig. 5(a) and Fig. 5(b) are 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% less latency.

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

$$Z_i(5211) = z_{i,3}\ 4 + z_{i,2}\ 2 + z^*_{i,1}\ 2 + z^*_{i,0} \tag{6}$$

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.

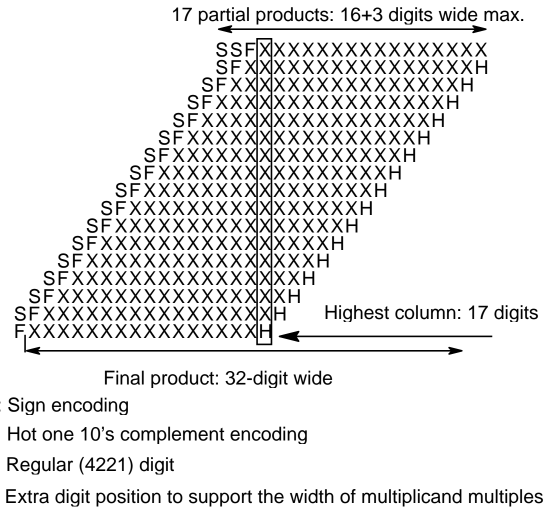
5 PARTIAL PRODUCT REDUCTION

First, we describe in Section 5.1 the partial product arrays generated by the SD radix-10 and SD radix-5 encodings. 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. To reduce the latency of the p:2 CSA trees, we make use of the decimal digit adders introduced in Section 5.3. These digit adders, implemented with bit counters, reduce up to 9 digits coded in (4221) or (5211) to 4 digits coded in (4221). Finally, we detail the design of the proposed p:2 decimal CSA trees implemented in the SD radix-10 (in Section 5.4) and SD radix-5 architecture (in Section 5.5). We present schemes optimized for area and for delay.

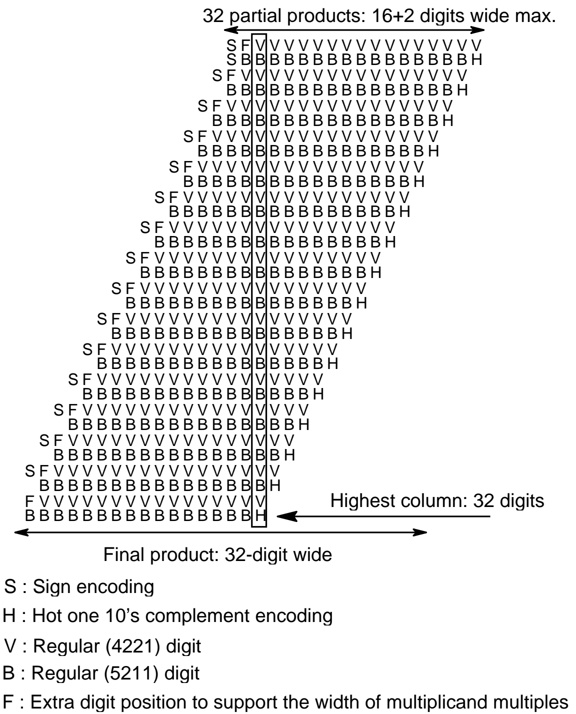
5.1 Partial product arrays

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. 7(a). 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 5.4.

For the SD radix-5 architecture, the number of partial products generated is equal to $2d$, d of them coded in (5221) and the other d coded in (4221) (see Section 4.2).



(a) SD radix-10 architecture.



(b) SD radix-5 architecture.

Fig. 7. Partial product arrays generated for 16-digit operands.

Both $PP[i]^U(5211)$ and $PP[i]^L(4221)$ have the same weight 10^i. Thus, for 16-digit input operands, the alignment of the 32 partial products results in the digit array of Fig. 7(b). The p-digit columns of the SD radix-5 partial product array are reduced using the mixed (4221/5211) decimal p:2 CSA trees presented in Section 5.5. The worst case for $d = 16$ corresponds to a column of $p = 32$ digits, reduced using a mixed (4221/5211) decimal 32:2 CSA.

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 nine, that is

$$\sum_{j=0}^{3} r_j = 9 \tag{7}$$

which includes the (4221), (5211), (4311) and (3321) codes, shown in Table 1. Some of these decimal codings are already known [35], 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. These codes have the following two properties:

- All the sixteen 4-bit vectors represent a decimal digit ($Z_i \in [0, 9]$). Therefore any boolean function (AND, OR, XOR,...) operating over the 4-bit vector representation of two or more input digits produces a 4-bit vector that represents a valid decimal digit (input and output digits represented in the same code).
- The 9's complement of a digit Z_i can be obtained by inverting their bits (as a 1's complement) since

$$9-Z_i = \sum_{j=0}^{3} r_j - \sum_{j=0}^{3} z_{i,j} r_j = \sum_{j=0}^{3} (1-z_{i,j}) r_j = \sum_{j=0}^{3} \overline{z_{i,j}} r_j \tag{8}$$

Negative operands can be obtained by inverting the bits of the positive bit vector representation and adding a 1 ulp, that is

$$-Z(r_3 r_2 r_1 r_0) = \overline{Z(r_3 r_2 r_1 r_0)} + 1 \tag{9}$$

Thus, using the first property, we perform fast decimal carry-save addition using a conventional 4-bit binary 3:2 CSA as

$$\begin{aligned} A_i + B_i + C_i &= \sum_{j=0}^{3} (a_{i,j} + b_{i,j} + c_{i,j})\ r_j \\ = \sum_{j=0}^{3} s_{i,j}\ r_j &+ 2 \times \sum_{j=0}^{3} h_{i,j}\ r_j = S_i + 2 \times H_i \end{aligned}$$

with $(r_3 r_2 r_1 r_0) \in \Big\{(4221), (5211), (4311), (3321)\Big\}$, $s_{i,j}$ and $h_{i,j}$ are the sum and carry bit of a full adder and $H_i \in [0, 9]$ and $S_i \in [0, 9]$ are the decimal carry and sum digits at position i. No decimal correction is required because the 4-bit vector expressions of H_i and S_i represent valid decimal digits in the selected coding.

However, a decimal multiplication by 2 is required before using the carry digit H_i for later computations. Here we restrict the analysis of decimal carry-save addition to only (5211) and (4221) decimal codes, since the generation of multiples of two for operands coded in (4311) and (3321) are more complex. Fig. 8 shows an example of $\times 2$ multiplications for decimal operands represented in (4221) and (5211) decimal codes. To simplify the notation, we use H for the carry vector coded in (4221) and W for the carry vector coded in (5211). Thus, we have that

$$2H = 2 \times H = L1_{shift}[W] \tag{10}$$

The resultant bit vector after shifting one bit to the left W represents the double of H. The operand $2H$ is coded in

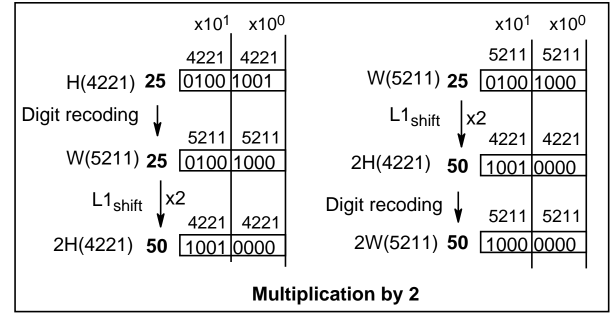
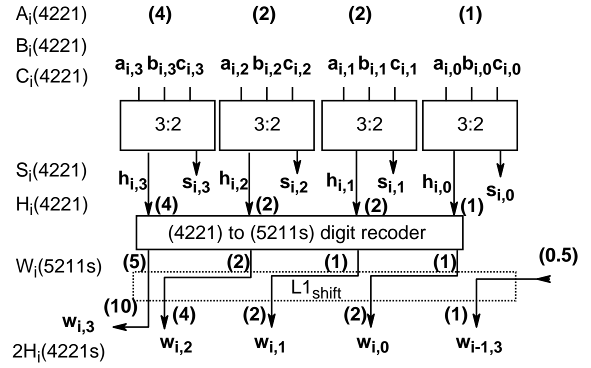
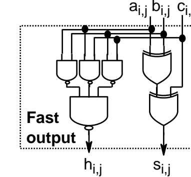


Fig. 8. Calculation of $\times 2$ for decimal operands coded in (4221) and (5211).



(a) Decimal digit (4-bit) 3:2 CSA.



(b) Full adder.

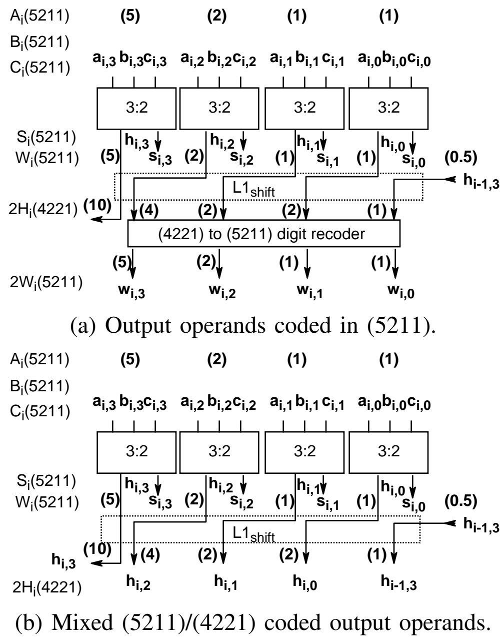
Fig. 9. Proposed decimal 3:2 CSA for operands coded in (4221).

(4221), since the weight bits of W are multiplied by 2 after the 1-bit left shift. The whole $2 \times H$ multiplication is performed by a digit recoding of H_i into W_i followed by a $L1_{shift}[W]$. The bits of W_i are denoted as $w_{i,j}$. The bit shifted out ($w_{i,3}$) represents a decimal carry out (weight 10) to the next digit position, while the bit shifted in ($w_{i-1,3}$) is a decimal carry input (weight 1). Thus, the digits of $2H$ are given by

$$(2H)_i = w_{i,2}\ 4 + w_{i,1}\ 2 + w_{i,0}\ 2 + w_{i-1,3} \tag{11}$$

Fig. 9(a) 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. 9(a) 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 produces a decimal sum digit (S_i) and a carry digit H_i coded in (4221), such that $A_i + B_i + C_i = S_i + 2 \times H_i$.

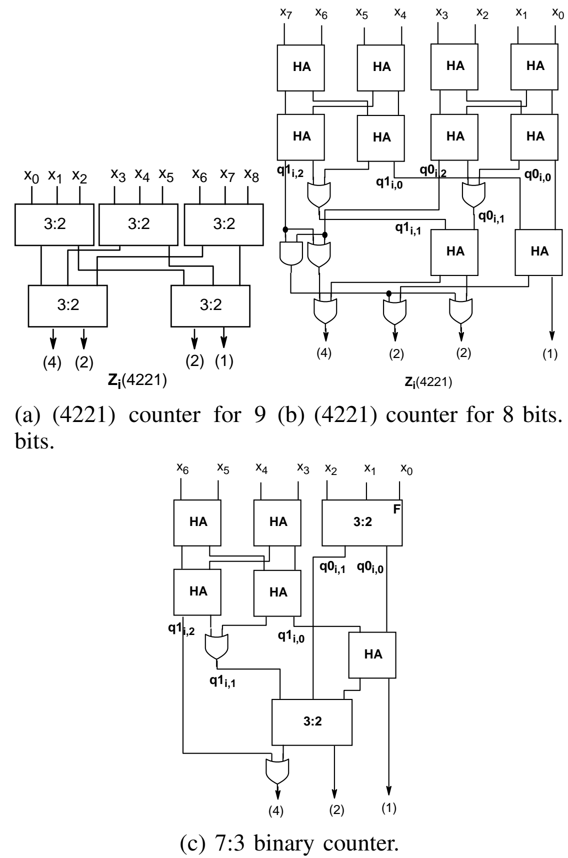
In order to obtain $(2H)_i$, H_i is first recoded to W_i using the (4221) to (5211s) digit recoder introduced in Section 4.4. The output of the digit recoder (W_i) is then left shifted by



(a) Output operands coded in (5211).

(b) Mixed (5211)/(4221) coded output operands.

Fig. 10. Proposed decimal 3:2 CSA for input operands coded in (5211).



(a) (4221) counter for 9 bits.

(b) (4221) counter for 8 bits.

(c) 7:3 binary counter.

Fig. 11. Gate level implementation of digit counters

one bit position ($L1_{shift}[W_i]$). 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. 9(b), reduces the total critical path delay.

If the input operands are represented in code (5211), $2W$ is obtained performing $2Z = L1_{shift}[W]$, followed by a digit recoding of $2Z$ from (4221) to (5211). The implementation of a decimal digit 3:2 CSA with inputs and outputs in code (5211) is shown in Fig. 10(a). Another possibility when the input digits are coded in (5211) is to use the carry vector $2Z$ coded in (4221) and the sum vector S coded in (5211). This decimal digit 3:2 CSA (Fig. 10(b)) consists only of a level of 4-bit 3:2 CSA with the carry output shifted 1-bit to the left.

5.3 Decimal digit adders using bit counters

We have designed a family of fast decimal digit adders that reduce 9, 8 or 7 digits coded in (4221) or (5211) into 4 or 3. These digit adders consists of a row of bit counters. These bit counters sum a column of up to $p = 9$ bits (same weight) and produce a q-bit vector ($q = \lceil \log_2 p \rceil \leq 4$) with weights (4221) which represents a decimal digit $Z_i \in [0, 9]$. In Fig. 11(a) we show an implementation of a bit counter that adds up to 9 bits producing a (4221) digit, which uses two levels of binary full adders. The binary weight of each output is indicated in brackets. Depending on the input, the path delay varies roughly from 2 to 4 XOR gate delays for output (1), from 2 to 3 XORs for outputs (2) and is about 2 XORs for output (4). The 8-bit counter of Fig. 11(b) only sums up to 8 bits but has a similar critical path delay as a binary 4:2 CSA (3 XOR gate delays for output (1)). Basically, the first two levels of half adders (HA) and the two OR gates perform the computation

$$Q0_i = q0_{i,2}\ 4 + q0_{i,1}\ 2 + q0_{i,0}\ 1 = \sum_{k=0}^{3} x_k$$
$$Q1_i = q1_{i,2}\ 4 + q1_{i,1}\ 2 + q1_{i,0}\ 1 = \sum_{k=4}^{7} x_k \qquad (12)$$

Since $Q0_i, Q1_i \in [0, 4]$, the total sum $Z_i(4221) = Q1_i + Q0_i \in [0, 8]$ is implemented in a simple way in the final logic level of Fig. 11(b) as

$$Z_i(4221) = \begin{cases} z_{i,3} = & q1_{i,2} \cdot q0_{i,2} \vee q1_{i,1} \cdot q0_{i,1} \\ z_{i,2} = & q1_{i,2} \cdot q0_{i,2} \vee q1_{i,0} \cdot q0_{i,0} \\ z_{i,1} = & q1_{i,2} \cdot q0_{i,2} \vee (q1_{i,1} \oplus q0_{i,1}) \\ z_{i,0} = & q1_{i,0} \oplus q0_{i,0} \end{cases}$$

In addition, a conventional 7:3 binary counter (shown in Fig. 11(c)), reduces a column of 7 bits into a 3-bit vector with weights (421).

These counters can be used to reduce 9 or 8 decimal digits (coded in (4221) or (5211)) into 4, or 7 digits into 3. An example of this procedure is described in Fig. 12 for nine input operands coded in (5211). The procedure is similar for (4221) input operands. A row of four (4221) decimal counters of Fig. 11(a) sums the values of each bit column producing a (4221) digit per bit column. The four bits of each (4221) digit are placed in a column from the most significant (top) to the least significant (bottom) and aligned in four rows according to

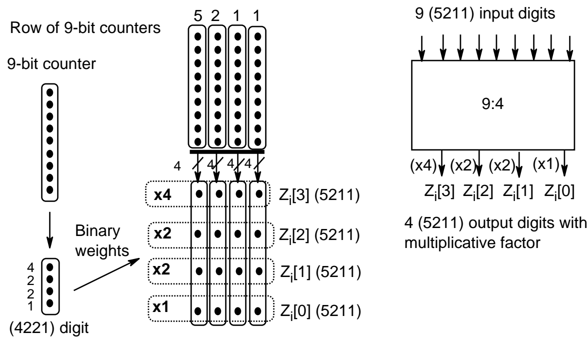
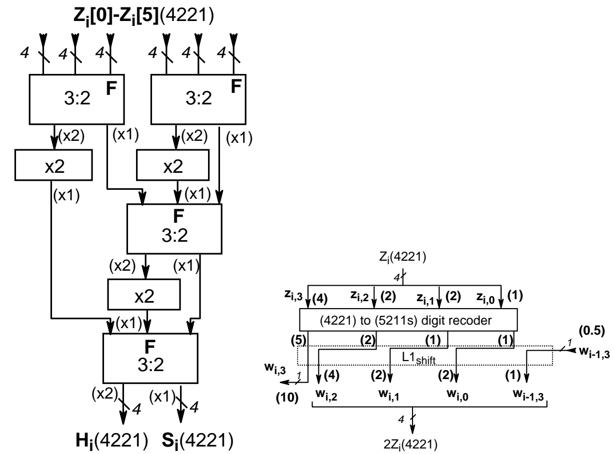


Fig. 12. $9:4$ reduction of (5211) decimal coded operands.



(a) Digit column. (b) Implementation of a $\times 2$ block.

Fig. 13. Decimal 6:2 CSA for (4221) decimal coded operands.

the weight bit of their column $(5\times10^i, 2\times10^i, 1\times10^i, 1\times10^i)$. This generates 4 decimal digits coded in (5211) that must be multiplied by a different factor (given by the binary weights of (4221)) before being added together. This organization causes all the bits of an output operand to have the same latency. The $\times2$ and $\times4$ blocks are implemented as described in Section 5.4. The 9:4 decimal digit adder is represented by the labeled box of Fig. 12 where the multiplicative factor of each output is indicated in brackets. The 8:4 and 7:3 decimal digit adders are implemented using a row of four counters of Fig. 11(b) and Fig. 11(c) respectively. Alternatively, the 4-bit decimal 3:2 CSA presented in Section 5.1 is a 3:2 decimal digit adder implemented using a row of 3-bit counters (a level of full adders) with the carry output multiplied by a factor $\times2$.

5.4 Decimal p:2 CSA trees for digits coded in (4221)

A decimal digit p:2 CSA tree reduces p ($p \geq 3$) input digits $Z_i[l]$ (with weight 10^i) 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^{i+1}) and a certain number of decimal carry inputs comes from the previous position (10^{i-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 $\times2$ 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). Fig. 13(a) shows an implementation of a decimal 6:2 CSA (the multiplicative factor associated with each signal is in brackets). We use the full adder configuration of Fig. 9(b) to minimize the critical path delay of the CSA tree. The digit blocks labeled $\times2$ consists of a (4221) to (5211s) digit recoder with the outputs (for 4221 coded operands) 1-bit left shifted, as shown in Fig. 13(b). 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.

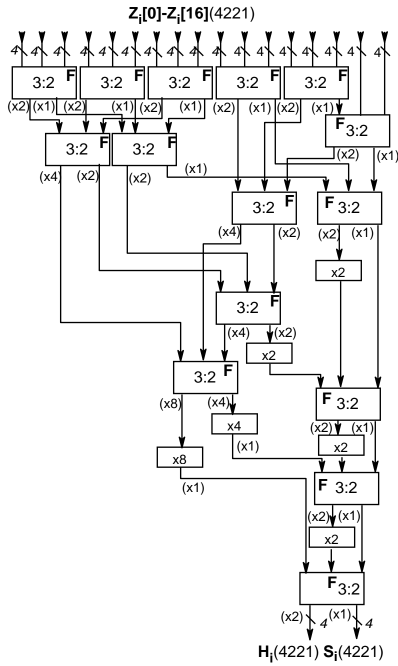
- For $p \geq 7$ we follow different strategies to obtain area-optimized or delay-optimized implementations. For area-optimized implementations, the input digits $Z_i[l]$ are reduced in a first level of binary 3:2 CSAs. Each intermediate operand is associated with a multiplicative factor power of two.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

$$2^n\,A+2^n\,B+2^n\,C = 2^n\,(A+B+C) = 2^n\,S+2^{n+1}\,H \quad (13)$$

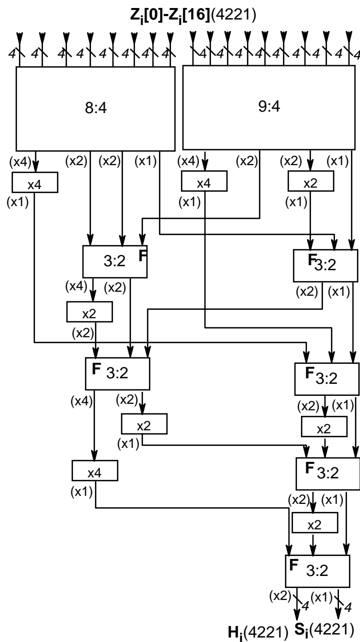
This reduces the hardware complexity since the overall number of $\times2$ operations is reduced. An area-optimized decimal 17:2 CSA tree for operands coded in (4221) is shown in Fig. 14(a). The $\times4$ and $\times8$ digit blocks produce two and three decimal carry-outs to the next significant digit column of the partial product array. They are implemented using a cascade configuration of $\times2$ blocks as shown in Fig. 15. We represent two adjacent digit columns to show how decimal carries are passed (lateral connections). For delay optimized implementations, the input digits $Z_i[l]$ are reduced in a first level of the decimal digit adders described in Section 5.3. These adders reduces 9 or 8 digits coded in (4221) or (5211) to 4 digits, and 7 digits to 3. The output digits, coded in (4221), may have multiplicative factors of ($\times4$), ($\times2$) or ($\times1$). 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 of Fig. 14(b) has more hardware complexity (equivalent to two $\times2$ 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 labeled 9:4 and 8:4 represent the decimal digit adders.

5.5 Decimal p:2 CSA trees for mixed (4221/5211) operands

These decimal p:2 CSA trees, with p even, reduce the columns of mixed (4221/5211) digits of the partial product array



(a) Area-optimized tree.



(b) Delay-optimized tree.

Fig. 14. Proposed decimal 17:2 CSAs.

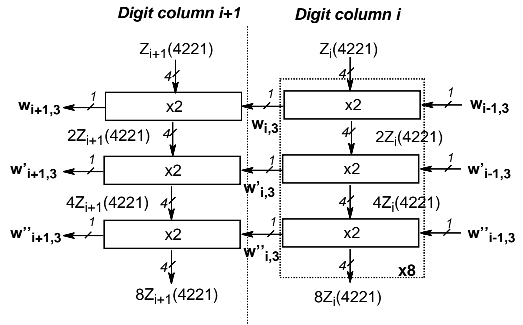


Fig. 15. Implementation of $\times 8$ multiplication for two adjacent columns.

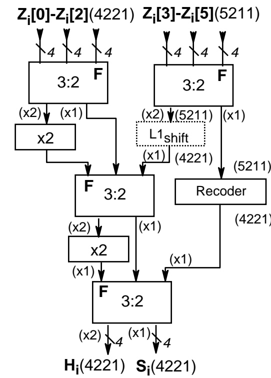


Fig. 16. Decimal 6:2 CSAs for mixed (4221/5211) decimal coded operands.

generated for the SD radix-5 architecture. Fig. 16 shows a decimal 6:2 CSA with mixed coded operands: half of the input digits are coded in (5211) and the other half in (4221). For (5211) coded operands, $\times 2$ is implemented as a 1-bit wired left shift, and the result is coded in (4221). A (5211) to (4221s) digit recoder is required to reduce a (5211) coded operand with two (4221) coded operands using the same 4-bit binary 3:2 CSA.

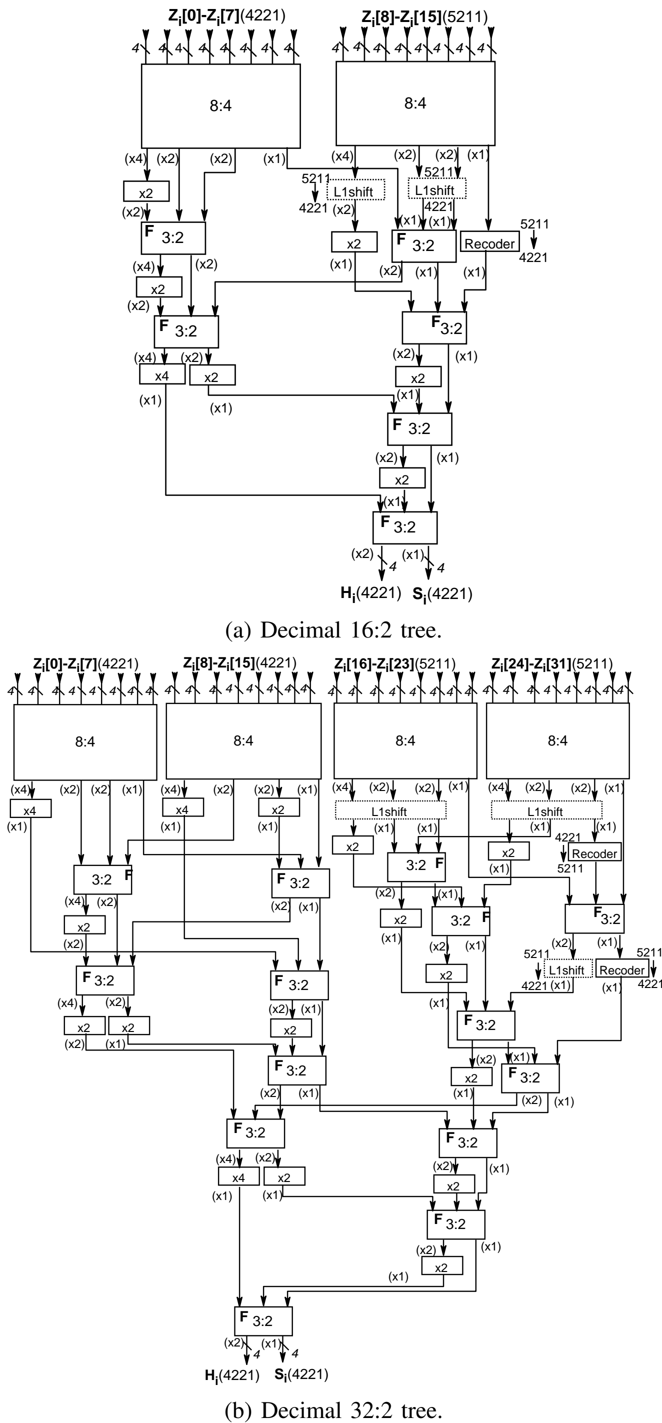
The decimal 16:2 and 32:2 CSA trees of Fig. 17 are delay-optimized implementations for mixed (4221)/(5211) digits. In particular, these designs take into account that the (5211) decimal coded operands are generated faster than the (4221) operands. Moreover, they use the fact that an $L1_{shift}$ performed over an operand coded in (5211) is equivalent to multiplying it by $\times 2$ and recoding to (4221).

6 Evaluation Results

We have estimated the area and delay of the SD radix-10 and SD radix-5 decimal multipliers for 16-digit (64-bit) BCD input operands. The area and delay figures were obtained from an area-delay evaluation model for static CMOS gates detailed in Section 6.1, and from the synthesis of verified RTL models coded in VHDL as detailed in Section 6.2.

6.1 Area-delay model based on logical effort

We have used a rough evaluation model for static CMOS gates to estimate the area and delay figures of the proposed architectures. The delays are given in FO4 units (delay of an 1x inverter with a fanout of 4 inverters) using a model based on logical effort [29]. The total stage delay is obtained as the sum of the delays of the gates on the critical path assuming equal input and output capacitances for the stage. It considers the different input and output gate loads, but neither interconnections nor gate sizing optimizations. Instead, we assume gates with the drive strength of the minimum sized (1x) inverter using buffers for high loads. The hardware complexity is given as the number of equivalent minimum size NAND2



(a) Decimal 16:2 tree.

(b) Decimal 32:2 tree.

Fig. 17. Proposed decimal CSAs for mixed coded operands.

Component	Delay ($FO4$)	L_{in} (# inv)	Area (NAND2)
Nand2	0.4+ 0.2 L_{out}	4/3	1
Xor2	0.9+ 0.2 L_{out}	2	2.5
FA (sum, carry)	(2.2, 0.8)+ 0.2 L_{out}	5	10
Bin. 4:2 CSA	4.0+ 0.2 L_{out}	5	20
(4221/5211s) rec	2.5+ 0.2 L_{out}	3	27
(4221s/5211s) rec	1.6+ 0.2 L_{out}	4	19
4-bit bin. CLA	4.5+ 0.2 L_{out}	5	45
BCD (4-bit) CLA	5.4+ 0.2 L_{out}	5	60

TABLE 3
Area and delay values for components.

	SD Radix-10		SD radix-5	
Component	Delay #FO4	Area #NAND2	Delay #FO4	Area #NAND2
SD rec.+buff	4.8+3.3	450*	3.6+3.3*	550*
Gen. of mult.	17.2*	2700*	3.5	350*
Sel. of mult.	3.3*	12850*	1.8*	11500*
PPG stage	20.5	16000	8.7	12400
PPR tree	21.5	15000	26.5	27500
Adder setup	3.2	1050	3.2	1050
128-bit adder	13.1	3700	13.1	3700
Total stage	58.3	35750	51.5	41300

*These terms contribute to the area or the delay of PPG.

TABLE 4
Area and delay for the proposed architectures.

gates. The cost related to the area of a gate is a function of the number of transistors and its size (width). We do not expect this rough model to give absolute area-delay figures, due to the high wiring complexity of parallel multipliers. However we consider that it is good enough for making design decisions at gate level and that it provides quite accurate area and delay ratios to compare different designs.

In Table 3 we detail the latency, input capacitance [3] (L_{in}) and area of some common components used in decimal multipliers. The L_{out} parameter represents the normalized output load connected to the gate. To obtain delay figures closer to the values given by synthesis, we use a XOR2 gate implemented with CMOS transmission gates, which is 30% faster and has 50% less area than the XOR2 used in our previous paper [33] (implemented with NAND2 gates). Therefore, these estimations may differ from the figures provided in [33], specially the area and delay figures for the binary and decimal CSA trees.

The area and delay figures for the 16-digit SD radix-10 and SD radix-5 architectures described in Sections 3.1 and 3.2 are shown in Table 4. The partial product generation (PPG) includes the recoding of the multiplier and the generation and the selection of multiples. For the SD radix-10 architecture, we provide the area and delay figures of an area-optimized implementation. Thus, the delay of the partial product reduction (PPR) is the critical path delay of the decimal 17:2 CSA of Fig. 7(b) (the area-optimized design). For the SD radix-5 architecture, we opt for an area-delay tradeoff implementation. In this case, the delay of PPR corresponds to the critical path delay of the mixed (4221/5211) decimal 32:2 CSA of Fig. 8(b). The BCD adder setup includes the $2 \times H$ multiplication and the conversion of $S(4221)$ to BCD excess 6.

6.2 Synthesis results

The 16-digit SD radix-10 (Fig. 1) and SD radix-5 (Fig. 2) combinational multipliers have been synthesized using Synopsys Design Compiler B-2008.09-SP3 and the Faraday UMC

3. Normalized to the input capacitance of the 1x inverter.

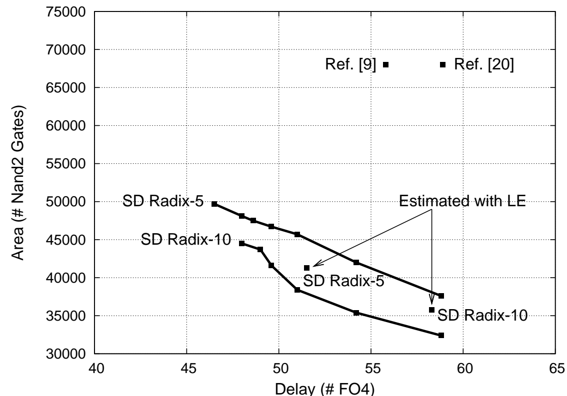


Fig. 18. Area-delay space obtained from synthesis.

90nm SP-RVT (Low k) CMOS standard cells library. The FO4 delay for this library is $49ps$ under typical conditions (1V, 25°C). The area-delay curves of Fig. 18 were obtained with the constraints $C_{out} = C_{in} = 4C_{inv}$, where C_{inv} is the input capacitance of an 1x inverter from the library. 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. 14(a). On the other hand, the SD radix-5 multiplier implements the decimal mixed (4221/5211) CSA tree of Fig. 17, optimized for delay. The area-delay values measured for the SD radix-10 multiplier range from $2.30ns$ (48 FO4) and 44500 NAND2 gates to $2.88ns$ (58.8 FO4) and 32400 NAND2 gates. For the SD radix-5 multiplier we have obtained values ranging from $2.28ns$ (46.5 FO4) and 49700 NAND2 to $2.88ns$ (58.8 FO4) and 37600 NAND2 gates. These figures agree with the values of Table 4 obtained from our area-delay evaluation model (also included in Fig. 18).

7 Comparison

We have analyzed several decimal carry-free tree adders for sixteen 64-bit operands based on different methods described in Section 2. We have evaluated and compared these implementations with our proposals using the area-delay evaluation model previously described.

We have also compared the area and delay figures obtained from synthesis of representative proposals of decimal multipliers (sequential and parallel) and a binary radix-4 parallel multiplier. In this Section we present the results of these comparisons.

7.1 Multioperand adders

The methods analyzed are grouped in decimal signed-digit (SD) trees [13], [28], BCD CLA trees [12], [20] and based on binary CSA trees [8], [18], [23]. We have also considered a binary 16:2 CSA using a 3-level tree of binary 4:2 compressors. Table 5 shows the area and delay estimations for these different decimal tree adders and sixteen 64-bit operands. The estimations for the proposed decimal CSA trees include a final stage to convert the (4221) decimal coded operands $2\times H$ and S into two BCD operands. Area and delay ratios are given with respect to our area-optimized decimal 16:2 CSA. Thus, a binary CSA tree of 4:2 compressors is 40% faster (delay ratio 0.60) and requires 30% less area (area ratio 0.70) than the proposed area-optimized decimal CSA. The proposed delay-optimized decimal CSA is slightly faster than the area-optimized decimal CSA (roughly 5% faster) but requires 10% more area. In the case of the (4221/5211) decimal mixed implementation, the figures are close to the area-optimized decimal CSA. The complexity of the signed-digit decimal adder [30] leads to decimal signed-digit tree adders [13], [28] with high area and delay figures, inappropriate for high speed multioperand addition. The decimal CSA tree proposed in [23] also presents high area and delay figures, due to the multiple and complex corrections and digit additions performed in the critical path. BCD CLA trees [12], [20] present good area/delay tradeoffs, but for high speed multioperand decimal addition the schemes based on binary trees [8], [18] may be a better choice. Compared with [8] our decimal CSA is at least 45% faster and requires about 10% less hardware and is, therefore, a good choice for high performance multioperand addition with moderate area.

7.2 Multipliers

Table 6 shows the area and delay estimations obtained from synthesis for some representative sequential and combinational multipliers. The comparison ratios are given with respect to the area and delay figures of a 53-bit binary Booth radix-4 multiplier extracted from [20]. The figures of the SD radix-10 and radix-5 multipliers corresponds to the points of minimum latency. The other two 16-digit decimal fixed-point parallel multipliers analyzed are detailed in [9], [20]. The recoding and the generation of partial products in [20] is similar to our SD radix-5 recoding scheme except that the 10's complement operation to obtain the negative BCD multiples, $-2X$ and $-X$, is more complex than a simple bit inversion. Also, they require combinational logic to generate the $5X$ multiple. For 16 decimal digits, 32 partial products are generated. The partial product reduction tree uses seven levels of decimal CSAs (implemented by arrays of 4-bit decimal CLAs) in parallel with two levels of decimal digit counters. The final assimilation consists of a simplified direct decimal carry-propagate adder. A variant of this multiplier proposed in [9] uses a configuration of full adders [8] to perform the partial product reduction. Both multipliers were synthesized using the STM 90nm CMOS standard cells library with a FO4 of $45ps$ for typical conditions. The latency values given by the authors for the combinational designs are $2.51ns$ (55.8 FO4) [9] and $2.65ns$ (58.9 FO4) [20]. Both designs present a hardware complexity of about 68000 NAND2 gates and are optimized for minimum delay. For comparison purposes, we also show these points in the area-delay design space of Fig. 18. We observe that the SD radix-5 decimal multiplier has a maximum speedup of 1.20 with respect to [9] using at most 0.70 times its area. On the other hand, the SD radix-10 multiplier is 1.15 times faster and 0.65 times smaller than the design proposed in [9].

To extract fair conclusions from the comparison between sequential and parallel implementations we have included

Architecture	Delay (# FO4)	Area (NAND2)	Delay Ratio	Area Ratio
Binary CSA	15.0	9000	0.60	0.70
SD tree adder [13], [28]	43.0	31300	2.00	2.55
BCD CLA trees [12], [20]	32.0	16200	1.30	1.30
Based on binary CSAs				
Ref. [23]	39.0	24000	1.60	1.90
Ref. [18]	30.0	17600	1.20	1.45
Ref. [8]	36.0	13500	1.45	1.10
Proposed decimal CSAs				
Delay-optimized (Section 5.4)	23.0	13800	0.95	1.10
Area-optimized (Section 5.4)	24.5	12500	1	1
Mixed (4221/5211) (Section 5.5)	24.0	12900	1.00	1.05

TABLE 5
Area-delay figures for multioperand tree adders (Sixteen operands of 16-BCD digits).

Architecture	Delay	Latency			Throughput	Area	
	(# FO4)	# Cycles	(# FO4)	Ratio	Mult./Cycle	(NAND2)	Ratio
Combinational multipliers:							
Bin. Booth radix-4 [20]	31.1	1	31	1.00	1	34000	1.00
Dec. Ref. [20]	58.9	1	58.9	1.90	1	68000	2.00
Dec. Ref. [9]	55.8	1	55.8	1.80	1	68000	2.00
Dec. SD Radix-10 (Fig. 1)	48	1	48	1.55	1	44500	1.30
Dec. SD Radix-5 (Fig. 2)	46.5	1	46.5	1.50	1	49700	1.45
BCD sequential multipliers:							
Ref. [13]	16	20	320	10.20	1/17	16000	0.50
Ref. [12]	14.7	20	294	9.45	1/17	18550	0.55
Ref. [19]	12.7	24	305	9.80	1/17	31500	0.90

TABLE 6
Area-delay figures for 53-bit binary/16-BCD digit decimal fixed-point multipliers.

the throughput of each multiplier. Sequential multipliers are more than two times smaller than parallel multipliers, but have higher latency and reduced throughput. For instance, the proposed SD radix-5 parallel multiplier is about 7 times faster than the best sequential implementation proposed in [13], but requires 3 times more area. In addition, it can issue a 16-digit BCD multiplication every cycle instead of one every 17 cycles.

8 Conclusions

In this paper we have presented several techniques to implement decimal parallel multiplication in hardware. We propose two different SD encodings for the multiplier that lead to fast parallel and simple generation of partial products. We have developed a decimal carry-save algorithm based on unconventional (4221) and (5211) decimal encodings for partial product reduction. It makes possible the construction of p:2 decimal CSA trees that outperform the area and delay figures of existing proposals. We have proposed architectures for decimal SD radix-10 and SD radix-5 parallel multiplication. The area and delay figures from a comparative study including conventional binary parallel multipliers and other representative decimal proposals show that our decimal SD radix-10 multiplier is an interesting option for high performance with moderate area. For higher performance the choice is the SD radix-5 architecture, although both designs have very close figures.

Acknowledgments

This work was done while A. Vazquez was with the Department of Electronic and Computer Engineering, University of Santiago de Compostela, Spain. It was supported in part by the Ministry of Education and Science of Spain under contract TIN 2007-67537-C03. The authors would also like to thank IBM for their support.

References

[1] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz and S. R. Carlough. “The IBM z900 Decimal Arithmetic Unit”, Conference Record of the Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1335–1339, Nov. 2001.

[2] F. Y. Busaba, T. Slegel, S. Carlough, C. Krygowski, J. G. Rell. “The design of the Fixed Point Unit for the z990 Microprocessor”, Proc. 14th ACM Great Lakes Symposium on VLSI 2004, pp. 364–367, Apr. 2004.

[3] I. D. Castellanos and J. E. Stine, “Compressor Trees for Decimal Partial Product Reduction”, 18th ACM Great Lakes Symposium on VLSI”, pp. 107–110, May 2008.

[4] M. Cornea, C. Anderson, J. Harrison, P.T.P. Tang, E. Schneider and C. Tsen, “A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format”, Proc. 18th IEEE Symposium on Computer Arithmetic, pp. 29–37, June 2007.

[5] M. F. Cowlishaw, “Decimal Floating-Point: Algorism for Computers”, 16th IEEE Symposium on Computer Arithmetic, pp. 104–111, July 2003.

[6] M. F. Cowlishaw, “The decNumber ANSI C library”, IBM Corp., 2003.

[7] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith and C. F. Webb, “A Decimal Floating–Point Specification”, 15th IEEE Symposium on Computer Arithmetic, pp. 147–154, June 2001.

[8] L. Dadda, “Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach”, IEEE Transactions on Computers, vol. 56, no. 10, pp. 1320–1328, Oct. 2007

[9] L. Dadda and A. Nannarelli “A Variant of a Radix-10 Combinational Multiplier”, IEEE Int. Symposium in Circuits and Systems, ISCAS 2008, pp. 3370–3373, May 2008
[10] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni and T. J. Bohizic, “Decimal Floating-Point in Z9: An Implementation and Testing Perspective”, IBM Journal Research and Development, pp. 217–227, vol. 51, No. 1/2, Jan. 2007.
[11] L. Eisen et al. “ IBM POWER6 accelerators: VMX and DFU”, IBM Journal Research and Development, pp. 663–684, vol. 51, No. 6, Nov. 2007.
[12] M. A. Erle and M. J. Schulte, “Decimal Multiplication Via Carry-Save Addition”, IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors, the Hague, Netherlands, pp. 348–358, June 2003.
[13] M. A. Erle, E. M. Schwarz, and M. J. Schulte, “Decimal Multiplication With Efficient Partial Product Generation”, Proc. 17th IEEE Symposium on Computer Arithmetic, pp. 21–28, June 2005.
[14] M. A. Erle, J. M. Linebarger, and M. J. Schulte, “Potential Speedup Using Decimal Floating-Point Hardware”, 36th Asilomar Conference on Signals, Systems and Computers, pp. 1073–1077, Nov. 2002.
[15] M. A. Erle, M. J. Schulte, and B. J. Hickman “Decimal Floating-Point Multiplication Via Carry-Save Addition”, Proc. 18th IEEE Symposium on Computer Arithmetic, pp. 46–55, June 2007.
[16] B. J. Hickman, A. Krioukov, M. A. Erle and M. J. Schulte, “A Parallel IEEE P754 Decimal Floating-Point Multiplier”, Proc. 25th IEEE Conference on Computer Design, pp. 296–303, Oct. 2007.
[17] IEEE Std 754(TM)-2008, “IEEE Standard for Floating-Point Arithmetic”, ISBN 978-0-7381-5753-5, IEEE Computer Society, Aug. 2008.
[18] R. D. Kenney and M. J. Schulte, “High-speed multioperand decimal adders”, IEEE Trans. on Computers, Vol. 54, No. 8, pp. 953-963, Aug. 2005.
[19] R. D. Kenney, M. J. Schulte, and M. A. Erle, “High-Frequency Decimal Multiplier”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 26–29, Oct. 2004.
[20] T. Lang and A. Nannarelli, “A Radix-10 Combinational Multiplier”, Proc. of 40th Asilomar Conference on Signals, Systems, and Computers, pp. 313–317, Oct. 2006.
[21] R. H. Larson, “High-Speed Multiply Using Four Input Carry-Save Adder”, IBM Tech. Disclosure Bulletin, vol. 16, No. 7, pp. 2053–2054, Dec. 1973.
[22] N. Ohkubo, M. Suzuki et al. “A 4.4 ns CMOS 54x54–bit multiplier using pass-transistor multiplexer”, IEEE Journal of Solid State Circuits, Vol. 30, No. 3, pp. 251–256, Mar. 1995.
[23] T. Ohtsuki et al.,“Apparatus for Decimal Multiplication”, U.S. Patent No. 4,677,583 , June 1987
[24] R. K. Richards, “Arithmetic Operations in Digital Computers”, New Jersey: D. Van Nostrand Company, Inc., 1955.
[25] M. Schmookler and A. Weinberger. “High Speed Decimal Addition”, IEEE Trans. on Computers, vol. c-20, no. 8, pp. 862-866, Aug. 1971.
[26] E. M. Schwarz, R. M. Averill, III, L. J. Sigal, “A Radix-8 CMOS S/390 Multiplier”, 13th IEEE Symposium on Computer Arithmetic (ARITH-13 ’97), pp. 2-9, July 1997.
[27] E. M. Schwarz, J. S. Kapernick and M. F. Cowlishaw, “Decimal Floating-Point Support on the IBM System z10 processor”, IBM Journal of Research and Development, vol. 51 , no. 1, Jan/Feb 2009.
[28] B. Shirazi, D. Y. Y. Yun, and C. N. Zhang, “RBCD: Redundant Binary Coded Decimal Adder”, IEE Proceedings - Computers and Digital Techniques, vol. 136, pp. 156–160, March 1989.
[29] I.E. Sutherland, R.F. Sproull and D. Harris, “Logical Effort: Designing Fast CMOS Circuits”, Morgan Kaufmann, 1999.
[30] A. Svoboda, “Decimal Adder with Signed-Digit Arithmetic”, IEEE Trans. on Computers, vol. C, pp. 212-215, Mar. 1969.
[31] T. Ueda, “Decimal Multiplying Assembly and Multiply Module”, US Patent No. 5379245, Jan. 1995.
[32] A. Vázquez and E. Antelo, “Conditional Speculative Decimal Addition”, Proc. of the 7th Conference on Real Numbers and Computers (RNC 7), pp. 47–57, July 2006.
[33] A. Vázquez, E. Antelo and P. Montuschi, “A New Family of High-Performance Parallel Decimal Multipliers”, Proc. 18th IEEE Symposium on Computer Arithmetic, pp. 195–204, June 2007.
[34] L. K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani. “Benchmarks and Performance Analysis of Decimal Floating-Point Applications”, Proc. of XXV IEEE International Conference on Computer Design, pp. 164–170, Lake Tahoe, CA, Oct., 2007.
[35] G. S. White, “Coded Decimal Number Systems for Digital Computers”, Proceedings of the IRE, vol. 41, No. 10, pp. 1450–1452, Oct., 1953.

Alvaro Vazquez graduated in Physics in 1997 and received the PhD. degree in Electronic and Computer Engineering in 2009 from the University of Santiago de Compostela, Spain. In 1998 he joined the Departamento de Electronica e Computacion at the University of Santiago de Compostela, where he worked on different research projects in the Computer Architecture Group. Since October 2009 he is a post-doctoral researcher at the Laboratoire de l’Informatique du Parallelisme, ENS Lyon, France. His research interests include different aspects of computer arithmetic and computer architecture, such as decimal floating-point arithmetic, design of high-speed and low-power numerical processors and algorithms and architectures for computing elementary functions.

Elisardo Antelo Elisardo Antelo graduated with a degree in Physics in 1991 and received the Ph. D. in computer engineering in 1995 from the University of Santiago de Compostela, Spain. In 1992, he joined the Departamento de Electronica e Computacion at the University of Santiago de Compostela. From 1992 to 1998, he was an assistant professor and, since 1998 he has been a tenured associate professor in this Department. He was a research visitor at the University of California at Irvine several times between 1996 and 2000. Dr. Antelo is a member of the Computer Architecture group at the University of Santiago de Compostela. Since 2001, he has been involved in the Program Committee of the IEEE Symposium on Computer Arithmetic. He also has been involved with the Program Committee of the Real Numbers and Computers Conference since 2006. Since 2007 he is an Associate Editor of IEEE Transactions on Computers. His primary research and teaching interest are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors, application-specific modules, computer arithmetic and design issues related to multi-core processors. Dr. Antelo is member of the IEEE and the IEEE Computer Society.

Paolo Montuschi graduated in electronic engineering in 1984 and received the PhD degree in computer engineering in 1989 from Politecnico di Torino, Italy. Since 2000, he has been a full tenured professor with Politecnico di Torino and, since 2003, he has been the chair of the Department of Computer Engineering at Politecnico di Torino. Currently, he is involved in several management and directive committees at Politecnico di Torino. From 1995 to 1997 and in 2006, he was deputy chair of the Center for Computing Facilities and Services of Politecnico di Torino. He worked as a visiting scientist with the University of California at Los Angeles and the Universitat Politecnica de Catalunya in Barcelona, Spain. Dr. Montuschi served on the program committees for the 13th through 19th IEEE Symposia on Computer Arithmetic and was program cochair of the 17th IEEE Symposium on Computer Arithmetic. From 2000 to 2004, he served as an associate editor of the IEEE Transactions on Computers. In 2009, he served as a co-guest editor for a special section on computer arithmetic in the IEEE Transactions on Computers. His current research interests cover several aspects of both computer arithmetic, with a special emphasis on algorithms and architectures for fast elementary function evaluations, and computer graphics. Dr. Montuschi is a member of the IEEE Computer Society and a senior member of the IEEE. From 2006 to 2008, he has been a member of the CPOC and from 2007 to 2009 of the DLOC (Digital Library Operating Committee) of the Computer Society. Since 2008, he has been a member-at-large of the Publication Board of the IEEE Computer Society. Since 2009 he is serving as Associate Editor of the IEEE Transactions on Computers.