A New Family of High–Performance Parallel Decimal Multipliers*

Alvaro Vázquez, Elisardo Antelo
University of Santiago de Compostela
Dept. of Electronic and Computer Science
15782 Santiago de Compostela, Spain
alvaro@dec.usc.es, elisardo@dec.usc.es

Paolo Montuschi
Politecnico di Torino
Dept. of Computer Engineering
10129 Torino, Italy
montuschi@polito.it

Abstract

This paper introduces two novel architectures for parallel decimal multipliers. Our multipliers are based on a new algorithm for decimal carry–save multioperand addition that uses a novel BCD–4221 recoding for decimal digits. It significantly improves the area and latency of the partial product reduction tree with respect to previous proposals. We also present three schemes for fast and efficient generation of partial products in parallel. The recoding of the BCD–8421 multiplier operand into minimally redundant signed–digit radix–10, radix–4 and radix–5 representations using new recoders reduces the complexity of partial product generation. In addition, SD radix–4 and radix–5 recodings allow the reuse of a conventional parallel binary radix–4 multiplier to perform combined binary/decimal multiplications. Evaluation results show that the proposed architectures have interesting area–delay figures compared to conventional Booth radix–4 and radix–8 parallel binary multipliers and other representative alternatives for decimal multiplication.

1. Introduction

Providing hardware support for decimal arithmetic is becoming a topic of interest. Specifically, the revision of the IEEE–754 Standard for Floating–Point Arithmetic (IEEE–754r) [1] already incorporates specifications for decimal arithmetic. Thus, it is expected that microprocessor manufacturers include decimal floating–point units in their products oriented to mainframe servers to satisfy the high performance demands of current financial, commercial and user–oriented applications [3].

An important and frequent operation in decimal computations is multiplication. However, due to the inherent inefficiency of decimal arithmetic implementations in binary logic, practically all the proposed decimal multipliers are sequential units [2, 4, 7, 9, 11, 16]. Recently, the first implementation of a parallel decimal multiplier was presented in [8]. Parallel multipliers are used extensively in most of the binary floating–point units [10, 13] and are of interest for decimal applications to scale performance.

In this paper, we introduce new methods for the efficient implementation of decimal parallel multiplication by a parallel generation of partial products and the reduction of these partial products using a novel decimal carry–save addition tree. We present the architectures of two different high–performance parallel multipliers that implement these methods. The second architecture also allows an effective implementation of a combined binary/decimal multiplier. These high–performance implementations have similar hardware complexity or a moderate increment in area with respect to the equivalent binary parallel multipliers. The paper is organized as follows. Section 2 outlines the previous (most representative) work on decimal multiplication. In Section 3 we introduce our proposals for an efficient implementation of decimal parallel multiplication. The proposed techniques for the generation of partial products are more detailed in Section 4, while the reduction of partial products is fully discussed in Section 5. We describe the two resulting architectures and some variants in Section 6. In Section 7 we provide rough area–delay evaluation results for 64–bit (16 decimal digits) decimal and binary parallel multipliers. We compare these results with some other representative works. Finally we summarize the main conclusions in Section 8.

2. An overview of decimal multiplication

Multiplication consists of three stages: generation of partial products, fast reduction (addition) of partial products to a two operand and a final carry propagate addition. Decimal multiplication is more complex than binary multiplication mainly for two reasons: the higher range of decimal digits

*A. Vázquez and E. Antelo supported in part by the Ministry of Science and Technology of Spain under contract TIN2004-07797-C02 and Xunta de Galicia under contract PGIDT03TIC10502PR.

($[0, 9]$), which increments the number of multiplicand multiples and the inefficiency of representing decimal values in systems based on binary logic using BCD–8421 (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 [2, 4] generates and stores all the required multiplicand multiples. Next, multiples are distributed to the reduction stage through multiplexers controlled by the multiplier digits. This approach requires more than a cycle to generate some complex BCD-8421 multiplicand multiples (3X,6X,7X,8X,9X). To avoid complicated multiples the multiplier can be recoded. In [8] each multiplier digit is recoded as $Y_i = Y_H\ 5 + Y_L$, with $Y_H \in \{0, 1\}$ and $Y_L \in \{-2, -1, 0, 1, 2\}$. Multiples 2X and 5X can be computed without a carry propagation over the whole number. Negative multiples requires an additional 9's complement addition. The second approach generates only the partial product as needed using digit–by–digit lookup table methods [9, 16]. In a recent work [5], a magnitude range reduction of the operand digits by a radix–10 signed–digit recoding (from [0,9] to [-5,5]) is suggested. This recoding of both operands speeds–up and simplifies the generation of partial products. Then, overlapped signed–digit partial products [1] are generated using simplified tables and a set of multiplexers and xor gates.

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 [12]. Proposals to perform the reduction of decimal partial products using multioperand carry–free addition were suggested in [9] (carry–save) and [15] (signed–digit). Recently several techniques have been proposed that improve these previous works. In [5] a signed–digit decimal adder based on [15] is used. Redundant binary coded decimal (RBCD) adders [14] can also perform decimal carry–free additions using a signed–digit representation of decimal digits ($\in [-7, 7]$). In [11] a scheme of two levels of 3–2 binary carry–save adders (CSA) is used to add the partial products iteratively. Since it uses BCD–8421 to represent decimal digits, a digit addition of +6 or +12 (modulo 16) is required to obtain the decimal carry and to correct the sum digit. Logic for detection of decimal carries and sum digit is in the critical path (sum path). In order to eliminate decimal corrections from the critical path of the binary CSA, three different techniques were proposed in [6]. Among these proposals, non–speculative adders present the best area–delay figures and are the most suitable for multioperand addition using a CSA tree. Non–speculative adders reduce the BCD–8421 input operands using a binary CSA tree. Preliminary sum digits are then obtained using a level of 4–bit carry propagate adders. Finally, decimal carry and sum digit corrections are determined from the preliminary sum digit and the carries passed to the next more significant digit position in the binary CSA tree[2]. Decimal correction is performed using combinational logic (its complexity depends on the number of input operands added) and a 3–bit carry propagate adder per digit.

Another representative technique [4] uses an array of 4–bit decimal carry–propagate adders based on direct decimal addition. This adder takes two BCD–8421 digits and a 1–bit input carry and generates a 1–bit decimal carry and the BCD–8421 sum digit. An iterative decimal multiplier based on a refinement of [4] is presented in [7]. It uses BCD–8421 invalid combinations to simplify the sum digit logic. A combinational radix–10 CSA tree is implemented in [8] using these 4–bit decimal carry–propagate adders. To optimize the partial product reduction they also use an array of decimal digit counters. Each counter adds 8 decimal carries of the same weight and produces a BCD–8421 digit.

3. Proposed techniques for decimal parallel multiplication

We assume that multiplicand X and multiplier Y are unsigned decimal integer words. Extension to decimal floating–point multiplication involves exponent addition, rounding of $X \cdot Y$ to fit the required precision and sign calculations. We represent the decimal digits of any d–digit decimal integer operand $Z = \sum_{i=0}^{d-1} Z_i \cdot 10^i$ as

$$Z_i = \sum_{i=0}^{3} z_{i,j} \cdot r_j$$

where $Z_i \in [0, 9]$ is the i^{th} decimal digit, $z_{i,j}$ is the j^{th} bit of the BCD i^{th} digit and r_j is the weight of the j^{th} bit. In Table 1 diverse BCD codings are represented. For BCD–8421, $r_j = 2^j$. BCD–4221 and BCD–5211 are new codings introduced in this paper characterized by the use of redundancy in decimal digit representation. As we have mentioned, the use of BCD–8421 to represent decimal digits means introducing costly decimal corrections in the partial product reduction binary CSA tree to obtain the correct decimal carry and sum. To avoid these corrections we use the BCD–4221 coding of Table 1 to represent partial product digits. Thus, we can perform fast decimal carry–save addition using an ordinary 4–bit binary 3:2 CSA as

$$A_i + B_i + C_i \quad = \quad \sum_{j=0}^{3} (s_{i,j} + 2h_{i,j})\ r_j$$

[1] Two overlapped digits in the range of $[-5, 5]$ and $[-2, 2]$ are generated for each partial product digit position.

[2] A +6 must be added each time a carry is passed to the next more significant digit position.

	BCD-8421	BCD-5421	BCD-4221	BCD-5211
0	0000	0000	0000	0000
1	0001	0001	0001	0001 | 0010
2	0010	0010	0010 | 0100	0100 | 0011
3	0011	0011	0011 | 0101	0101 | 0110
4	0100	0100	1000 | 0110	0111
5	0101	1000	1001 | 0111	1000
6	0110	1001	1010 | 1100	1001 | 1010
7	0111	1010	1011 | 1101	1100 | 1011
8	1000	1011	1110	1110|1101
9	1001	1100	1111	1111

Table 1. BCD codings

$$= \sum_{j=0}^{3} s_{i,j}\ r_j \quad + \quad 2 \sum_{j=0}^{3} h_{i,j}\ r_j = S_i + 2H_i$$

with $(r_3, r_2, r_1, r_0) = (4, 2, 2, 1)$ and

$$\begin{aligned} s_{i,j} &= a_{i,j} \oplus b_{i,j} \oplus c_{i,j} \\ h_{i,j} &= a_{i,j} \cdot b_{i,j} \vee (a_{i,j} \vee b_{i,j}) \cdot c_{i,j} \end{aligned}$$

$H_i \in [0, 9]$, $S_i \in [0, 9]$ are the decimal carry and sum digits at position i while symbols $\vee$, $\cdot$, and $\oplus$ indicate binary operators OR, AND and XOR respectively. No decimal correction is required because H_i and S_i are valid decimal digits in BCD–4221 code. However a decimal multiplication by 2 is required before using the carry digit for the computations. This can be performed in a simple way by a digit recoding to BCD–5211 (shown in Table 1) followed by a 1–bit wired left shift:

$$2H_i = l1shift(W_i) = w_{i,3}\ 10 + w_{i,2}\ 4 + w_{i,1}\ 2 + w_{i,0}\ 2$$

where

$$W_i = w_{i,3}\ 5 + w_{i,2}\ 2 + w_{i,1} + w_{i,0}$$

is the BCD–5211 recoded decimal carry digit. Moreover, this operation is in the fast path (carry path of a full–adder). Note that the 1–bit left shift of W_i produces a carry output ($w_{i,3}$) to the next decimal digit ($i + 1$), while the less significant bit position is occupied by the carry input ($w_{i-1,3}$) of the previous digit W_{i-1}. Logical expressions for BCD–4221 to BCD–5211 recoding are given by

$$\begin{aligned} w_{i,3} &= h_{i,3} \cdot (h_{i,2} \vee h_{i,1} \vee h_{i,0}) \vee h_{i,2} \cdot h_{i,1} \cdot h_{i,0} \\ w_{i,2} &= h_{i,2} \cdot h_{i,1} \cdot \overline{h_{i,3} \oplus h_{i,0}} \vee (h_{i,3} \cdot \overline{h_{i,0}}) \oplus h_{i,2} \oplus h_{i,1} \\ w_{i,1} &= h_{i,2} \cdot h_{i,1} \cdot \overline{h_{i,3} \oplus h_{i,0}} \vee h_{i,3} \cdot \overline{h_{i,0}} \cdot \overline{h_{i,2} \oplus h_{i,1}} \\ w_{i,0} &= (h_{i,2} \cdot h_{i,1}) \oplus h_{i,3} \oplus h_{i,0} \end{aligned}$$

Nevertheless, due to the redundancy of BCD–4221 and BCD–5211 codings, there are several choices with different area–delay trade–offs for the logical implementation of this digit recoding. This decimal carry–save algorithm leads to fast and area optimized decimal carry–save tree adders detailed in Section 5. Furthermore, conversions between BCD–8421 and BCD–4221 codings can be performed using a simple gate level.

To generate all the partial products in parallel, we obtain all the required multiples. We aim for a fast generation of a reduced number of partial products. This is achieved with the recoding of the multiplier. We have developed three different recodings for the multiplier with good trade–offs between fast generation of partial products and the number of partial products generated. A minimally redundant signed–digit (SD) radix–10 recoding (digits in $[-5, 5]$) produces only $d+1$ partial products but requires a carry propagate addition to generate complex multiples $3X$ and $-3X$. Minimally redundant signed–digit (SD) radix–4 and radix–5 recodings (with digits in $[-2, 2]$) produce $2d$ partial products (2 digits per radix–10 digit) but multiplicand multiples are produced in a few levels of combinational logic. Furthermore, another advantage of using BCD–4221 to represent partial product digits is that the 9's complement of each digit can be obtained by bit inverting each digit. This simplifies the generation of the negative multiplicand multiples. The proposed BCD–8421 to SD recoders and the generation and selection of multiples are detailed in Section 4.

For the final decimal carry propagate addition we use a binary quaternary tree (Q-T) adder modified to perform decimal additions [17]. Decimal quaternary tree adders based on conditional speculative decimal addition present low latency (about 10% more than the fastest binary adders) and require less hardware than other alternatives.

4. Generation of partial products

4.1. Multiplier recoding

A. Signed–Digit Radix–10 Recoding.

This recoding transforms the digit set $\{0, \ldots, 9\}$ into the signed–digit (SD) set $\{-5, \ldots, 5\}$ to perform the selection of multiples in a similar way as modified Booth recoding. Fig. 1 shows a block diagram of the recoding and the multiplicand multiple selection units.

We denote $Y_i^* = y_{i,3}^*\ 5 + \sum_{j=0}^{2} y_{i,j}^*\ 2^j$ the digits of the multiplier coded in BCD–5421 (see Table 1). The recoded SD radix–10 multiplier can be expressed in terms of Y_i^* as

$$\begin{aligned} Y &= \sum_{i=0}^{d-1} \left(y_{i,3}^*\ 10 - y_{i,3}^*\ 5 + \sum_{j=0}^{2} y_{i,j}^*\ 2^j \right) 10^i \\ &= y_{d-1,3}^*\ 10^d - \sum_{i=0}^{d-1} Yb_i\ 10^i \end{aligned}$$

where the value of each SD radix–10 digit $Yb_i \in [-5, 5]$ is

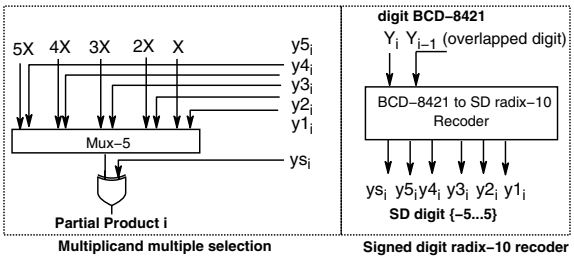


Figure 1. Partial product generation for SD radix–10.

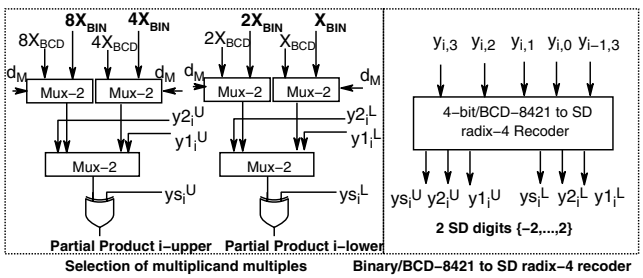


Figure 2. Partial product generation for SD radix–4.

given by

$$Yb_i = -y^*_{i,3}\, 5 + \sum_{j=0}^{2} y^*_{i,j}\, 2^j + y^*_{i-1,3}$$

with $y^*_{-1,3} = 0$. Control signals (in "hot–one" code) can be obtained directly from input BCD–8421 multiplier digits using the following logical expressions:

$$\begin{aligned}
ys_i &= y_{i,3} \vee y_{i,2} \cdot (y_{i,1} \vee y_{i,0}) \\
y5_i &= y_{i,2} \cdot \overline{y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1}) \\
y4_i &= ys_{i-1} \cdot y_{i,0} \cdot (y_{i,2} \oplus y_{i,1}) \\
y3_i &= y_{i,1} \cdot (y_{i,0} \oplus ys_{i-1}) \\
y2_i &= \overline{y_{i,0} \vee ys_{i-1}} \cdot (y_{i,3} \vee \overline{y_{i,2}} \cdot y_{i,1}) \\
y1_i &= \overline{y_{i,2} \vee y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1})
\end{aligned}$$

Since multiplicand multiples are recoded to BCD–4221, negative multiples can be generated by the XOR of ys_i with the corresponding positive multiple as shown in the multiplicand multiple selector of Fig. 1.

B. Signed–Digit Radix–4 Recoding.

Two SD radix–4 digits $Y_i^U \in \{0,1,2\}$ (upper), $Y_i^L \in \{-2,-1,0,1,2\}$ (lower) are generated per each BCD–8421 digit ($Y_i = Y_i^U \cdot 4 + Y_i^L$). We obtain the SD radix–4 selection signals directly from the BCD–8421 digits as

$$(Y_i^U) \begin{cases} ys_i^U = y_{i,3} \\ y2_i^U = \overline{y_{i,3}} \cdot y_{i,2} \cdot y_{i,1} \\ y1_i^U = \overline{y_{i,3}} \cdot y_{i,2} \oplus y_{i,1} \end{cases}$$

$$(Y_i^L) \begin{cases} ys_i^L = y_{i,3} \vee y_{i,1} \\ y2_i^L = ys_i^L \cdot \overline{y_{i,0}} \cdot \overline{y_{i-1,3}} \vee \overline{ys_i^L} \cdot y_{i,0} \cdot y_{i-1,3} \\ y1_i^L = y_{i,0} \oplus y_{i-1,3} \end{cases}$$

The block diagram of a 4–bit combined binary/decimal recoder and the corresponding multiplicand multiple selector are shown in Fig. 2 where control signal d_M is true for decimal multiplication. The combined SD radix–4 recoder implements the decimal selection signals and the conventional Booth radix–4 selection signals. Upper signals select multiples $\pm 8X$ and $\pm 4X$ while lower signals select multiples $\{-2X, -X, X, 2X\}$. Although the resulting combined SD radix–4 recoders and multiple selectors are simple, obtaining decimal multiples $4X$ and $8X$ requires double and triple latency with respect to obtaining the decimal $2X$ multiple.

C. Signed–Digit Radix–5 Recoding.

This recoding uses a different set of multiplicand multiples ($5X$,$10X$ instead of $4X$,$8X$) for decimal partial product generation that have a similar latency to $2X$ and X. Each BCD–8421 digit of the multiplier is encoded into two radix–5 digits ($Y_i = Y_i^U \cdot 5 + Y_i^L$) with $Y_i^U \in \{0,1\}$ and $Y_i^L \in \{-2,-1,0,1,2\}$.

SD radix–5 selection signals are obtained from the BCD–8421 input digits using:

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

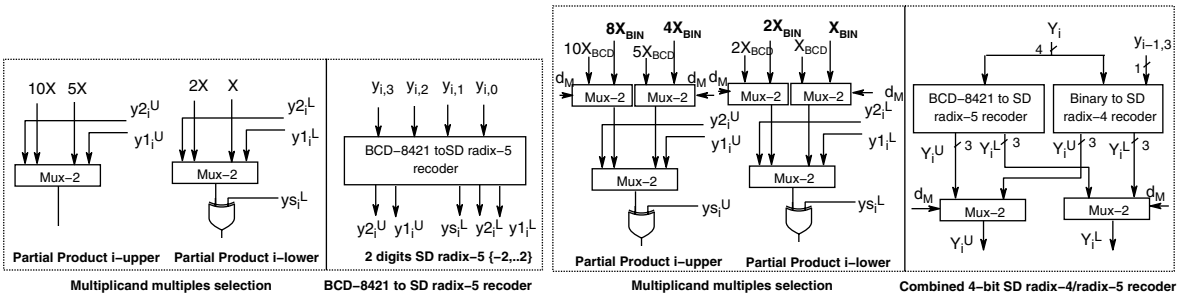
$$(Y_i^L) \begin{cases} ys_i^L = y_{i,3} \vee y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}} \vee \overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0} \\ y2_i^L = \overline{y_{i,0}} \cdot (y_{i,3} \vee y_{i,1}) \vee \overline{y_{i,2}} \cdot y_{i,1} \\ y1_i^L = y_{i,2} \cdot \overline{y_{i,0}} \vee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

The block diagram of the digit recoder and multiples selector is shown in Fig. 3(a). A combined binary radix–4/decimal radix–5 block diagram for the partial product generation is proposed in Fig. 3(b). Multiplexers controlled by d_M select the operands required by binary or decimal multiplications. Although BCD to SD radix–4 encoding is slightly simpler than radix–5, partial product generation for decimal SD radix–5 is faster and comparable in latency with binary SD radix–4, due to a faster generation of multiplicand multiples as we show in the following subsection.

4.2. Generation of multiplicand multiples

Decimal multiplicand multiples $2X$ and $5X$ are obtained in a few levels of logic using recoding and wired left shifts. Any other multiple is generated using these multiples or from multiplicand X. The generation sequence of $2X$ is as follows. Each BCD–8421 digit is first recoded to BCD–5211 using

$$w_{i,3} \quad = \quad h_{i,3} \vee h_{i,2} \cdot (h_{i,1} \vee h_{i,0})$$



(a) Decimal SD radix–5 recoding. (b) Combined binary/decimal to SD radix–4/radix–5 recoding.

Figure 3. Partial product generation for SD radix–5.

$$
\begin{aligned}
w_{i,2} &= h_{i,3} \vee (h_{i,1} \oplus (h_{i,2} \cdot \overline{h_{i,0}})) \\
w_{i,1} &= h_{i,3} \cdot h_{i,0} \vee h_{i,2} \cdot \overline{h_{i,1} \vee h_{i,0}} \\
w_{i,0} &= h_{i,3} \vee (h_{i,2} \oplus h_{i,0})
\end{aligned}
$$

Then a wired 1–bit left shift is performed over the recoded multiplicand, obtaining the $2X$ multiple in BCD–4221.

The $5X$ multiple is obtained by a simple 3–bit left shift of the multiplicand, but with resultant digits coded in BCD–5421. Thus a digit recoding from BCD–5421 to BCD-4221 is performed using expressions

$$
\begin{aligned}
w_{i,3} &= h_{i,3} \vee h_{i,2} \\
w_{i,2} &= h_{i,3} \cdot (h_{i,2} \vee (h_{i,1} \cdot h_{i,0})) \\
w_{i,1} &= h_{i,1} \cdot h_{i,3} \cdot (h_{i,2} \vee h_{i,0}) \\
w_{i,0} &= h_{i,3} \oplus h_{i,0}
\end{aligned}
$$

The generation of negative multiples is performed by evaluating the 10's complement of positive multiples as

$$-X = \sum_{i=0}^{d-1} (9 - X_i) \cdot 10^i + 1$$

For BCD–8421 this is performed by a digit addition of +6 followed by a bit–complement operation since $9 - X_i = \overline{X_i + 6}$. For BCD–4221, a 10's complement is performed simply by bit–complementing the positive multiple, since $9 - X_i = \overline{X_i}$. Addition of the 10's complement +1 is performed in the partial product reduction tree by a tail encoding bit, since each partial product is 4–bit (or at least 1–bit) left shifted from the previous one. To avoid sign extension and thus to reduce the complexity, the partial product signs sg_i are encoded in each leading digit position as

$$
\begin{aligned}
&-\sum_{i=0}^{d-1} sg_i 10^{i+d} = -10^{2d} + \sum_{i=0}^{d-1} (9 - sg_i) 10^{i+d} + 1 = \\
&= -10^{2d} + \sum_{i=1}^{d-1} (8 + \overline{sg_i}) 10^{i+d} + (\overline{sg_0} 10 + sg_0 9) 10^d
\end{aligned}
$$

Each partial product is at most of $d + 3$–digit length, due to the three extra digit positions required for the encoded sign, the tail encoding bit and the left shifting.

Fig. 4(a) shows the block diagram for the generation of multiplicand multiples for SD radix–10 encoding. Multiple $4X$ is obtained as $2 \times 2X$. Multiple $3X$ is evaluated by a carry propagate addition of multiples X and $2X$ in a decimal quaternary tree [17]. The latency of the partial product generation is constrained by the generation of $3X$. The SD radix–10 multiple selector of Fig. 1 uses the xor operation to select positive or negative multiples as a function of the SD radix–10 control signal ys_i.

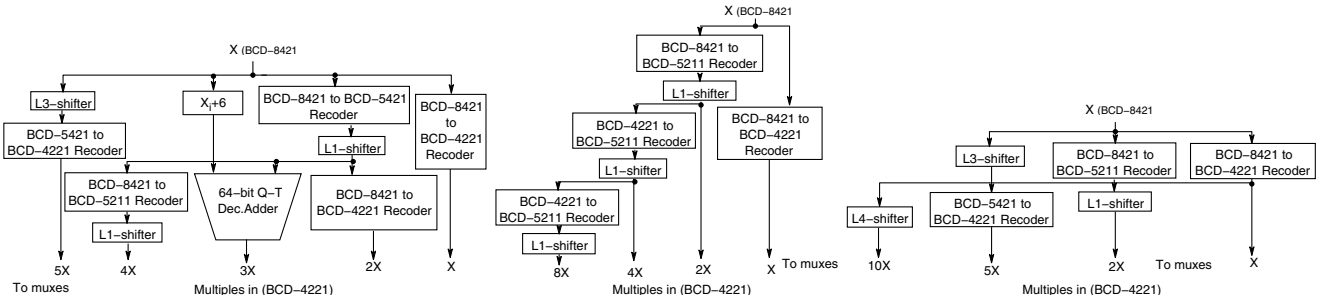
Fig. 4(b) shows the generation of multiples for the case of decimal SD radix–4 recoding. Multiple $8X$ is obtained as $2 \times 2 \times 2X$, so the latency of multiplicand multiples generation is about three times the latency of $2X$ operation. On the other hand, generation of radix–5 multiples is faster (approx. the latency of $2X$) as it is shown in Fig. 4(c).

5. Reduction of partial products

To implement the algorithm for carry–save addition formulated in Section 3 we propose a decimal 3:2 CSA that reduces 3 BCD–4221 digits to a carry and a sum BCD–4221 digits. This module consists of a 4–bit binary 3:2 CSA plus a BCD–4221 to BCD–5211 digit recoder. From this module we construct *p:2* ($p \geq 3$) decimal CSAs, optimizing the critical path delay using fast inputs and outputs.

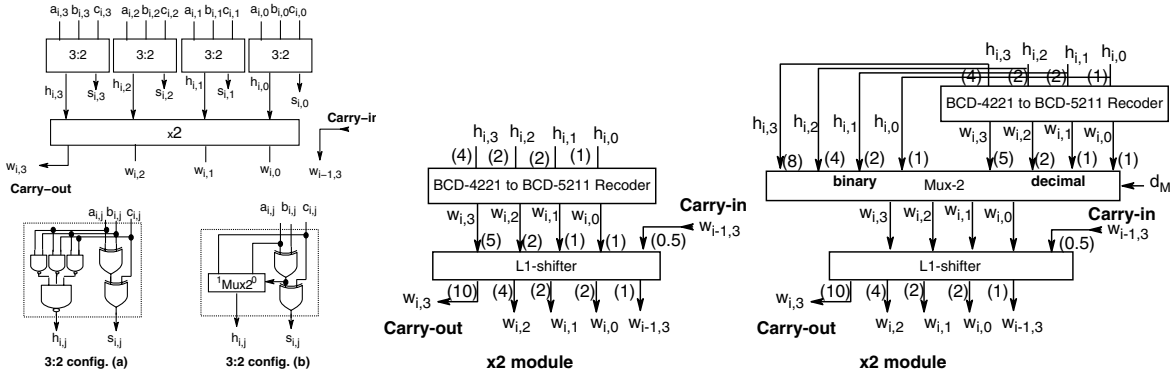
5.1. Decimal 3:2 carry-save adder

The block diagram of the proposed 4–bit 3:2 CSA is shown in Fig. 5(a). The block labeled $\times 2$ performs the multiplication of the carry digit by 2. For decimal multiplication the $\times 2$ module is detailed in Fig. 5(b). It consist of a BCD–4221 to BCD–5211 digit recoder and a 1–bit wired left shift. A combined binary/decimal 3:2 CSA is shown in Fig. 5(c). A 4–bit 2:1 multiplexer controlled by d_M selects



(a) Multiples for SD radix–10 encoding. (b) Multiples for SD radix-4. (c) Multiples for SD radix-5.

Figure 4. Generation of multiplicand multiples.



(a) Generic scheme of the 3:2 CSA. (b) Scheme of $\times 2$ for BCD-4221. (c) Scheme of $\times 2$ for both binary/decimal.

Figure 5. Proposed 3:2 CSA for decimal and combined binary/decimal.

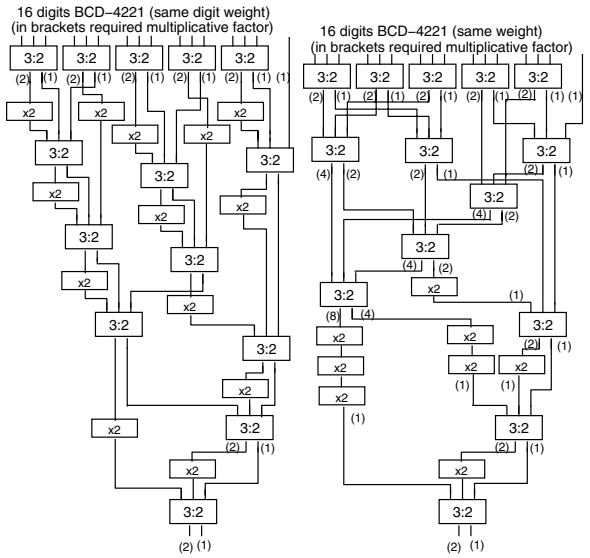
the appropriate output. Different implementations for the binary 3–2 CSA have been proposed. We consider the two alternatives shown in Fig. 5(a). Since their inputs present different delays we use the most suitable in every particular case to minimize the final path delay of the CSA tree.

5.2. Decimal *p:2* CSA trees

To reduce the partial products generated by the SD radix–10 encoder we require a *(d+1):2* decimal CSA while, for the other two SD recodings, the reduction is accomplished by a *2d:2* decimal CSA. Fig. 6 shows two examples of a 16:2 decimal CSA trees that reduce 16 rows of decimal digits to 2. The blocks labeled as 3:2 represent a 4–bit binary 3:2 CSA. The modules labeled as $\times 2$ can be either the decimal module of Fig. 5(b) or the combined binary/decimal module of Fig. 5(c). Thus these CSA trees can perform both binary and decimal additions. In the first implementation (Fig. 6(a)), every carry output is multiplied by 2 before connecting to any other input. Since the carry path is slightly more complex than the sum path, outputs of block $\times 2$ are connected to fast inputs of the 3:2 CSA. The second implementation (Fig. 6(b)) reduces the hardware complexity by adding the carry outputs of the previous tree level before being multiplied by 2. Therefore it is necessary to perform several $\times 2$ operations in a row for some paths. Both implementations present similar critical path delays so, aside from wiring issues, the second implementation (Fig. 6(b)) is preferable, because of reduced hardware complexity. A 32:2 decimal CSA tree, shown in Fig. 7(a), uses the 16:2 CSA of Fig. 6(b) and a first level of decimal 4:2 CSAs shown in Fig. 7(b). This decimal 4:2 CSA was designed for optimal carry and sum path delays, assuming similar input delays. The $\times 2$ block was described in Fig. 5.

6. Proposed Architectures

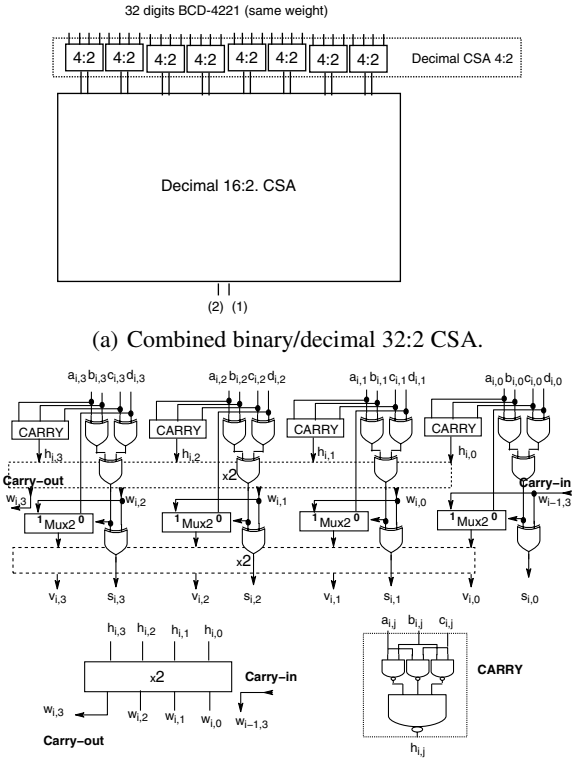
Based on the techniques described in the previous Sections we have designed a decimal radix–10 parallel multiplier and combined binary/decimal parallel multipliers for both SD radix–4 and radix–5 recodings. We assume 16 decimal digit (64–bit) input operands coded in BCD–8421.



(a) Evaluating $\times 2$ for each carry output.

(b) Grouping digits with the same multiplicative factor.

Figure 6. Proposed 16:2 combined binary/decimal CSAs.



(a) Combined binary/decimal 32:2 CSA.

(b) Combined binary/decimal 4:2 CSA.

Figure 7. Proposed 32:2 combined binary/decimal CSAs.

6.1. SD radix–10 architecture

The dataflow of the proposed SD radix–10 architecture is shown in Fig. 8(a). The generation of the 17 partial products is performed by an encoding of the multiplier into 16 SD radix–10 digits and an additional leading bit as described in Section 4. Each SD radix–10 digit controls a level of 64–bit 5:1 mux and 64 xor gates that select the corresponding multiple of the multiplicand. The block for the generation of multiples was described in Fig. 4(a). The alignment of partial products at the different decimal positions is performed by 4–bit wired left shifts. The array of 17 partial products is then reduced using the decimal 16:2 CSA of Fig. 6(b) and an additional decimal 3:2 CSA level for the highest columns or simpler CSAs trees for lower columns.

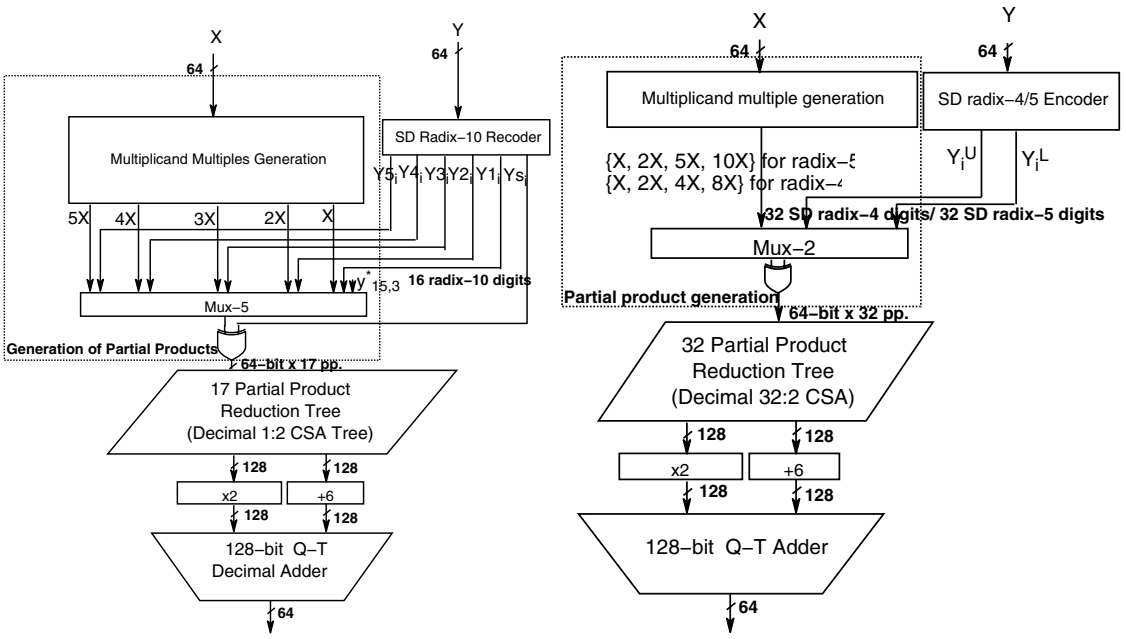
Before using a 128–bit decimal quaternary–tree (Q–T) adder to obtain the final products, a +6 digit operation is required to produce the correct decimal digits for conditional speculative decimal addition [17]. Sum operand digits are first recoded from BCD–4221 to BCD–8421 before adding +6. Combined recoding and digit addition of +6 has practically the same logical complexity than a single recoding. The $\times 2$ multiplication for the final decimal carry operand is performed in parallel with the first stage of the decimal carry–propagate adder (+6 digit addition). This $\times 2$ multiplication uses a BCD–4221 to BCD–5421 recoder [3] to obtain the operand expressed in BCD–8421. Furthermore, the latency of the 128–bit Q–T decimal adder block shown in Fig. 8 is similar to the equivalent 128–bit Q–T binary adder.

[3]This recoder presents a similar latency and hardware complexity as the BCD–4221 to BCD–5221 recoder

6.2. SD radix–4 and radix–5 architectures

The dataflow of the proposed combined binary SD radix–4/decimal SD radix–4 (or radix–5) architecture is shown in Fig. 8(b). We have different architectures depending on the scheme used to generate the partial products. A decimal radix–4 or a combined binary/decimal SD radix–4 multiplier can be implemented using the recoder of Fig. 2 and the multiplicand multiple generator of Fig. 4(b). A decimal radix–5 multiplier is implemented using the recoder of Fig. 3(a) and the multiples generator of Fig.4(c), while the combined binary radix–4/decimal radix–5 architecture is implemented using modules of Fig.3(b) and Fig.4(c).

In all cases, 32 partial products are generated. The array of 32 partial products is reduced using the proposed 32:2 CSA tree of Fig. 7(a) for the highest columns or simpler CSAs for the other columns. The final carry–propagate addition is performed with a 128–bit combined binary/decimal Q-T adder [17].



(a) Decimal radix-10 parallel multiplier. (b) Combined binary/decimal radix–4 (radix–5) multiplier.

Figure 8. Dataflow of the proposed decimal parallel multipliers.

7. Evaluation and Comparison

We have used an area–delay model for static CMOS gates based on logical effort to evaluate the area–delay figures of the proposed architectures and two representative binary parallel multipliers [10, 13]. We also use this model to compare our architectures with other proposals for decimal multiplication. It considers input and output loads, but neither interconnections nor gate sizing optimizations. Instead, we assume gates with the drive strength of the minimum sized inverter using buffers for high loads. The delay is given in FO4 units (delay of a 1x inverter with a fan–out of four 1x inverters) and area values in minimum size NAND2 gates units. We do not expect this model to give accurate area–delay figures, due to the high wiring complexity of parallel multipliers, however we consider that this model provide rough area/delay ratios good enough to compare different designs.

7.1. Evaluation of proposed multipliers

Evaluation results for partial product generation (16 digits) are shown in Table 2. Area–delay figures for the proposed SD radix–4 (radix–5) partial product generators are comparable to the Booth radix–4 binary scheme. On the other hand, the radix–10 implementation presents area–delay figures similar to the binary Booth radix–8 partial product generator. For decimal multiplication, SD radix–5 based implementations are preferable to SD radix–4 implementations, while for combined implementations SD radix–4 is the choice when a fast binary operation is required. Table 3 shows the evaluation results for different binary and decimal 16:2 and 32:2 CSA trees. We also provide results for the binary 22:2 CSA implemented in the radix–8 multiplier using 3 levels of binary 4:2 CSAs and a binary 3:2 CSA level. Area and delay ratios for proposed *p:2* CSA trees are given with respect to the equivalent *p:2* binary CSA tree. Delay ratios are close to 1.4 (40% more latency) in the case of decimal CSAs and 1.3/1.5 for combined binary/decimal CSAs. For area ratios, figures are close to 1.2 (20% more area) in the case of decimal implementations and around 1.4 for combined implementations. Evaluation results for parallel multipliers are shown in Table 4. From

Architecture	Delay		Area	
	(t_{FO4})	Ratio	(NAND2)	Ratio
Bin.radix–4	9	1.0	15000	1.0
Bin. radix–8	19	2.1	18000	1.2
Dec. radix–4	14	1.6	16000	1.1
Dec. radix–5	9.5	1.05	15000	1.0
Bin/Dec. radix-4	9/15.5	1.0/1.7	16500	1.1
Bin/Dec. radix–5	10.5/11	1.2/1.2	16000	1.1
Dec. radix–10	21	2.3	18000	1.2

Table 2. Area–delay for partial product generation.

CSA	Delay		Area	
	(t_{FO4})	Ratio	(NAND2)	Ratio
Bin. 16:2	20	1	13000	1
Dec. 16:2	29	1.4	15000	1.2
Bin./Dec. 16:2	27/31	1.3/1.5	17500	1.4
Bin. 22:2	25	–	18500	–
Bin. 32:2	27	1.0	25500	1.0
Dec. 32:2	38	1.4	30000	1.2
Bin./Dec. 32:2	35/41	1.3/1.5	33500	1.3

Table 3. Area–delay figures for partial product reduction.

Architecture	Delay		Area	
	(t_{FO4})	Ratio	(NAND2)	Ratio
Bin. radix–4	50	1.0	43000	1.0
Bin. radix–8	57	1.15	39500	0.90
Dec. radix–4	70	1.4	49500	1.15
Dec. radix–5	65	1.3	49000	1.10
Bin/Dec. radix–4	59/75	1.2/1.5	54000	1.25
Bin/Dec. radix–4/5	61/71	1.2/1.4	53500	1.25
Dec. radix–10	72	1.45	40000	0.90
Proposed in [8]	92	1.85	69000	1.60

Table 4. Area–delay fig. for 16–BCD digit multipliers.

these results we conclude that the decimal radix–10 architecture is the appropriate option for high performance decimal multiplications, since it shows a moderate area rather attractive for a feasible commercial implementation. For higher performance the decimal radix–5 multiplier is our choice, but at the expense of 30% of delay overhead with respect to a radix–4 binary implementation. When a combined binary/decimal implementation is required, the preferred option is the radix–5 architecture for low latency decimal multiplication and the radix–4 architecture for low latency binary multiplication.

7.2. Comparison with previous proposals

So far, the only known fully parallel implementation of a decimal multiplier is [8]. The recoding and the generation of partial products is practically similar to our SD radix–5 recoding scheme except that [8] requires a 9's complement operation to obtain negative multiples $-2X$ and $-X$. 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 in a simplified direct decimal carry–propagate addition. Synthesis results given in [8] show a critical path delay of 2.65ns and an equivalent area of 68.000 NAND2 gates, while ratios are 1.90 for delay and 1.50 for area respect to a radix–4 binary multiplier. The last row of Table 4 shows the evaluation results using our model and the comparison ratios with the binary radix–4 multiplier. We observe that our decimal multipliers have a speed–up between 1.30 and 1.40 respect to [8] using at most 0.70 times its area.

To provide fair comparative results for sequential proposals we have had to compare with techniques that can be directly applied to decimal parallel multiplication. Representative techniques are grouped in those proposed for partial product generation and those proposed for partial product reduction.

A. Partial product generation. Proposed methods based on the generation of all the multiples of the multiplicand [2, 4] use directly the BCD–8421 multiplier digits ($[0, 9]$) to select the corresponding multiplicand multiple ($\{0X, X, \ldots, 9X\}$). The parallel generation of complex multiples $\{3X, 6X, 7X, 9X\}$ requires 4 carry–propagate adders, while the selection of multiples uses more wiring and circuitry than the proposed partial product generation. Therefore, our proposed SD recodings of the multiplier simplify the partial product generation by reducing the hardware complexity of multiplicand multiples generation. On the other hand, the high hardware requirements of the proposals based on generation of partial products on demand [9, 16], make them impractical for parallel partial product generation. The signed–digit (SD) recoding of input operands proposed in [5] reduces these high demands. The partial product generation for 64–bit operands includes two 16–digit radix–10 SD recoders, 16×16 blocks for the generation of overlapped partial product digits and 16×16 recoders to transform these overlapped digits into signed digits. The estimated hardware complexity is of more than 40,000 NAND2 gates and the estimated delay is 16 FO4. Therefore, our proposed SD radix–10 and SD radix–5 partial product generators require 2.5 times less hardware than [5]. The proposed SD radix–5 is 1.7 times faster than [5] but generates 32 partial products while the proposed SD radix–10 scheme is 1.3 times slower than [5]. However, the partial product reduction using our proposed decimal CSA tree presents better figures than an equivalent SD tree adder in terms of both area and delay as we show below. Instead of a SD tree adder, a decimal CSA can be used for overlapped partial product reduction, but additional hardware complexity is required to manage signs for each digit position.

B. Partial product reduction. For partial product reduction, we have analyzed the methods described in Section 2 that allow a carry free decimal tree implementation. Proposed methods can be grouped in decimal signed–digit (SD) trees [5, 14], decimal 4–bit CLA trees [4, 7] and decimal CSA trees [6, 11]. Table 5 shows the area/delay estimations for these different decimal tree adders and sixteen 64–bit operands. The complexity of the signed–digit decimal

CSA tree (16 operands)	Delay Ratio	Area Ratio
Binary 16:2	1	1
SD tree† [5, 14]	2.85	3.50
CLA tree † [4, 7]	2.10	1.65
BCD–8421 † [11]	2.20	3.10
Non Spec. ‡ [6]	1.85	1.75
Proposed 16:2 (Fig. 6(b))	1.45	1.20

†Obtained from area/delay estimations of fast implementations of a SD adder (one digit) [15], a 4–bit decimal CLA [12] and a one digit BCD–8421 3:2 CSA [11]. Ratios given respect area–delay figures of a 4–bit binary 3:2 CSA.
‡Obtained from synthesis evaluation results provided in [6].

Table 5. Area–delay figures for 64–bit CSA trees.

adder [15] leads to decimal signed—digit tree adders [5, 14] with high area and delay figures, inappropriate for high speed multioperand addition. The decimal CSA tree proposed in [11] also presents high area and delay figures, due to the multiple and complex corrections and digit additions performed in the critical path. Decimal CLA trees [4, 7] present good area/delay trade-offs, but for high speed multioperand decimal addition the non speculative CSA tree [6] is a better choice. Compared with this implementation our decimal CSA uses 45% less area and is 30% faster and is, therefore, a good choice for high performance multioperand addition with moderate area. Moreover, area and delay ratios for the non speculative CSAs provided in [6] increase with the number of input operands, while for our decimal CSAs these ratios are roughly constant.

8. Conclusions

In this paper we have presented several techniques to implement decimal parallel multiplication in hardware. We propose three different SD encodings for the multiplier that lead to fast parallel and simple generation of partial products. For partial product reduction we have developed a decimal carry–save algorithm based on a BCD–4221 representation of decimal digit operands. It makes possible the construction of *p:2* decimal CSA trees that outperform the area–delay figures of existing proposals. Moreover, proposed techniques also allow the computation of combined binary/decimal multiplications with a moderate overhead. We have proposed an architecture for decimal SD radix–10 parallel multiplication and two combined architectures for binary/decimal SD radix–4 and binary SD radix–4/decimal SD radix–5 multiplication. The area–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 or combined binary/decimal multiplications the choices are the binary/decimal SD radix–4 or radix–5 implementations.

References

[1] IEEE standard for floating–point arithmetic. IEEE Standards Committee, Oct. 2006. Available at http://754r.ucbtest.org/drafts/754r.pdf.

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

[3] M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In *Proc. IEEE* 16^{th} *Symposium on Computer Arithmetic*, pages 104–111, July 2003.

[4] 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.

[5] M. A. Erle, E. M. Schwarz, and M. J. Schulte. Decimal multiplication with efficient partial product generation. In *Proc. IEEE* 17^{th} *Symposium on Computer Arithmetic*, pages 21–28, June 2005.

[6] R. D. Kenney and M. J. Schulte. High-speed multioperand decimal adders. *IEEE Trans. on Computers*, 54(8):953–963, Aug. 2005.

[7] R. D. Kenney, M. J. Schulte, and M. A. Erle. High-frequency decimal multiplier. In *Proc. IEEE Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 26–29, Oct. 2004.

[8] T. Lang and A. Nannarelli. A radix-10 combinational multiplier. In *Proc.* 40^{th} *Asilomar Conference on Signals, Systems, and Computers*, pages 313–317, Oct. 2006.

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

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

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

[12] M. Schmookler and A. Weinberger. High speed decimal addition. *IEEE Trans. on Computers*, C-20(8):862–866, Aug. 1971.

[13] E. M. Schwarz, R. M. Averill, and L. J. Sigal. A radix-8 CMOS S/390 multiplier. In *Proc. IEEE* 13^{th} *Symposium on Computer Arithmetic*, pages 2–9, July 1997.

[14] B. Shirazi, D. Y. Y. Yun, and C. N. Zhang. RBCD: Redundant binary coded decimal adder. *IEE Proc - Computers and Digital Techniques*, 136(2):156–160, Mar. 1989.

[15] A. Svoboda. Decimal adder with signed–digit arithmetic. *IEEE Trans. on Computers*, C-18(3):212–215, Mar. 1969.

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

[17] A. Vázquez and E. Antelo. Conditional speculative decimal addition. In *Proc.* 7^{th} *Conference on Real Numbers and Computers (RNC 7)*, pages 47–57, July 2006.