

Improving the Speed of Parallel Decimal Multiplication

Ghassem Jaberipur, Amir Kaivani

Abstract— Hardware support for decimal computer arithmetic is regaining popularity. One reason is the recent growth of decimal computations in commercial, scientific, financial, and internet-based computer applications. Newly commercialized decimal arithmetic hardware units use radix-10 sequential multipliers that are rather slow for multiplication intensive applications. Therefore, the future relevant processors are likely to host fast parallel decimal multiplication circuits. The corresponding hardware algorithms are normally composed of three steps: partial product generation (PPG), partial product reduction (PPR), and final carry propagating addition. The state of the art is represented by two recent full solutions with alternative designs for all the three aforementioned steps. In addition, PPR by itself has been the focus of other recent studies. In this paper we examine both of the full solutions and the impact of a PPR-only design on the appropriate one. In order to improve the speed of parallel decimal multiplication, we present a new PPG method, fine tune the PPR method of one of the full solutions and the final addition scheme of the other; thus assembling a new full solution. Logical Effort analysis and 0.13 μm synthesis show at least 13% speed advantage, but at a cost of at most 36% additional area consumption.

Index Terms— Decimal computer arithmetic; Parallel decimal multiplication; Partial product generation and reduction; Logic design.

1 INTRODUCTION

Decimal computer arithmetic is widely used in financial, commercial, scientific, and internet-based applications. One reason is the need for precise floating-point representation of many decimal values (e.g., 0.2) that do not have an exact binary representation [1]. Conventional decimal arithmetic has been, often until lately, implemented by extremely slow software simulation on binary processors [1]. However, hardware implementations have recently gained more popularity. For example, the IBM eServer z900 [2], IBM POWER6 [3] and IBM z10 [4] have special hardware units dedicated to decimal floating point arithmetic. The binary-coded decimal (BCD) encoding of decimal digits, that is commonly used in these software and hardware implementations, is recognized by the latest IEEE standard for floating-point arithmetic (IEEE 754-2008 [5]).

The computer arithmetic literature includes variety of articles on decimal arithmetic hardware such as BCD adders (e.g., [6], [7] and [8]), multi-operand BCD adders (e.g., [9] and [10]), sequential BCD multipliers (e.g., [11], [12], and [13]), and dividers (e.g., [14], [15], [16], and [17]). However, some single arithmetic operations (e.g., division or square root), and also function evaluation circuits (e.g., radix-10 exponentiation or logarithm), are often implemented by a sequence of simpler operations

including several multiplications. Therefore, hardware implementation of such operations and functions would call for high speed parallel decimal multiplication. Such multiplication schemes are generally implemented as a sequence of three steps: partial product generation (PPG), partial product reduction (PPR), and the final carry-propagating addition. We have encountered only two full solutions that thoroughly address all the three steps [18, 20]. They use similar PPG methods, but different approaches for the other two steps.

In this paper, we introduce a new fast decimal PPG method, modify and use the PPR scheme of [18] and adapt the BCD adder of [8] to suit the peculiarity of the input to our third step. We also evaluate the impact of applying the very recent PPR-only method of [19] on the full design of [20]. We use Logical Effort [21] area-delay analysis to evaluate and compare the proposed multiplier with that of [18], [20] and a modified version of the latter. The comparisons show that the proposed approach leads to considerable lower latency.

The rest of this paper includes, in Section 2, a general discussion on the three steps of decimal multiplication. In Section 3, we address the previous decimal PPG methods and propose a new technique. In Section 4, we examine the recent contributions on decimal multi-operand addition covering the special case of PPR. This is where we analyze the scheme offered in [18] and propose our improvement options. The final product computation is taken up in Section 5. A comparison of our results with those of the recent contributions, based on Logical Effort analysis and circuit synthesis, is presented in Section 6, and finally Section 7 draws our conclusions.

• G. Jaberipur is with the Electrical and Computer Engineering Department, Shahid Beheshti University. He is also affiliated with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. E-mail: jaberipur@sbu.ac.ir.

• A. Kaivani is with the Electrical and Computer Engineering Department, Shahid Beheshti University, Tehran, Iran. E-mail: a_kaivani@sbu.ac.ir.

2 DECIMAL MULTIPLIERS

Multiplication is implemented by accumulation of partial products, each of which is conceptually produced via multiplying the whole multiplicand by a weighted digit of the multiplier. Decimal multiplication, as in binary, may be accomplished sequentially, in parallel, or by a semi-parallel approach [22]. Following the recent trend [18 and 20], we focus on fully parallel BCD multiplication, which is generally a 3-step process:

- a) **Partial product generation (PPG):** A partial product $W_i = Y^i \times X$, as in Fig. 1, is generated via multiplying the multiplicand X by a multiplier digit Y^i , where the positional weight of W_i and Y^i is equal to the i^{th} power of the radix (e.g., 10^{*i*}). One could think of BCD multipliers producing all the partial products in parallel by a matrix of BCD digit multipliers [22], or through selection of precomputed multiples [23, 11, and 18]. In the case of [20], 4-2-2-1 decimal encoding is used to represent the partial products.
- b) **Partial product reduction (PPR):** This is the major step in parallel multiplication, which is a special case of the more general problem of multi-operand decimal addition [9 and 10]. A parallel solution for this problem uses a reduction tree that leads to a redundant intermediate representation of product (e.g., two equally-weighted BCD numbers [20]).
- c) **Final product computation:** The redundant product computed in Step b) is converted to the final BCD product (e.g., via a BCD adder [8]).

Fig. 1 illustrates Steps a), b), and c) for a k -digit BCD multiplication, where uppercase letters denote decimal digits, superscripts indicate the decimal weighted position of the digits, and subscripts are meant to distinguish the relative decimal weights of partial products.

3 DECIMAL PARTIAL PRODUCT GENERATION

$$\begin{array}{l}
 \text{a)} \quad \left[\begin{array}{cccc} & & X^{k-1} \dots X^2 X^1 X^0 \\ & & \times Y^{k-1} \dots Y^2 Y^1 Y^0 \\ \hline & & W_0^{k-1} \dots W_0^2 W_0^1 W_0^0 \\ & & W_1^{k-1} \dots W_1^2 W_1^1 W_1^0 \\ & & \vdots \\ & & W_{k-1}^{k-1} \dots W_{k-1}^2 W_{k-1}^1 W_{k-1}^0 \end{array} \right. \\
 \text{b)} \quad \left[\begin{array}{cccc} & & & \\ & & & \\ \hline & & S^{2k-2} \dots S^{k-1} \dots S^2 S^1 S^0 \\ & & C^{2k-2} \dots C^k \dots C^3 C^2 C^1 \\ \hline & & P^{2k-1} P^{2k-2} \dots P^{k-1} \dots P^2 P^1 P^0 \end{array} \right. \\
 \text{c)} \quad \left[\begin{array}{cccc} & & & \\ & & & \\ \hline & & & \\ & & & \end{array} \right.
 \end{array}$$

Fig. 1: The three steps of parallel BCD multiplication

Three common methods of decimal partial product generation are:

- **Lookup table:** This method, in principle, is based on lookup tables with 8-bit addresses and one hundred entries to hold the values of a 10×10 decimal multiplication table [24]. Larger lookup tables to implement multiply-add operations are used in [25]. The lookup table approach requires significant circuitry and delay [13]. Therefore, it is not very popular in modern designs for decimal multiplication, especially when it comes to high precision multipliers (e.g., 16- and 34-digit decimal multipliers recommended by IEEE 754-2008 format for decimal numbers [5]).
- **BCD digit-multipliers:** Fully combinational delay-optimized and area-optimized BCD digit-multipliers, with eight input bits and eight output bits, are offered in [22].
- **Precomputed multiples:** A straightforward approach for decimal partial product generation is to precompute all the possible ten multiples (0-9X) of the multiplicand X at the outset of the multiplication process. The appropriate partial product is picked by a ten-way selector. However, 0 and X are readily available, and low-cost carry-free operations can lead to BCD numbers $2X$ and $5X$ [23]. The multiples $4X = 2 \times 2X$ and $8X = 2 \times 2 \times 2X$ can also be generated as single BCD numbers in constant time. This is not possible for the other multiples (i.e., $3X$, $6X$, $7X$, and $9X$). For example let $X = (3)^+9$, where $(D)^+$ stands for a string of one or more decimal number D . It is easy to see that $3X$, $6X$ and $9X$ require word wide carry propagation. Likewise, it can be verified, with more effort, that the same is true for $7 \times (142857)^+9$. However, these hard multiples and also $8X$ can be expressed as double-BCD numbers composed of two equally weighted easy multiples (i.e., X , $2X$, $4X$, and $5X$). Note that generation of $9X$ as $(4X, 5X)$ is faster than $(8X, X)$, although both are performed in constant time.

Alternative improved versions of method c) [11 and 18] that originated in [23] lead to very efficient partial product generation that outperforms methods a) and b). We now briefly address these methods and present a new one in Section 3.1. Table I provides a latency-based categorization for generation of the ten multiples, where X is the multiplicand.

TABLE I: THE LATENCY OF BCD MULTIPLE COMPUTATION

Latency	Multiples
Immediate	0, X
Constant time	$2X$, $4X$, $5X$, $8X$
Word-width dependent	$3X$, $6X$, $7X$, $9X$

The immediate availability of 0 and X is obvious, while constant-time computation of the other multiples is described below:

- **T = 2X:** Doubling a BCD number X is a carry-free conversion of each digit of X (say X^i) to the corresponding digit of T ($T^i = t_3^i t_2^i t_1^i t_0^i$), such that $2 \times X^i = 10C^{i+1} + t_3^i t_2^i t_1^i 0$ [23]. The three most significant bits of T^i and C^{i+1} can be computed by a simple combinational logic found in [11] and $t_0^i = C^i$.
- **4X and 8X:** These multiples may also be computed, in constant time, via two and three successive multiplications by 2, respectively.
- **F=5X:** This multiple can be computed via multiplying all the digits of X by 5, in parallel. For the i^{th} digit X^i we have $5 \times X^i = 10C^{i+1} + R^i$, where $R^i \in \{0,5\}$ and $C^{i+1} \leq 4$. Therefore, $F^i = R^i + C^i \leq 9$ guarantees carry-free multiple computation [23]. The required simple combinational logic can be found in [22].
- **M = mX (m ∈ {3, 6, 7, 9}):** As explained before, these hard multiples cannot be derived, in constant time, as single BCD numbers. However, Eqn. set 1 shows how they can be configured as double-BCD numbers in terms of easy multiples.

$$3X=(2X, X), 6X=(5X, X), 7X=(5X, 2X), 9X=(5X, 4X) \quad (1)$$

The latter method of computation of multiples is used in [11], where $8X$ is alternatively represented as two equally weighted multiples (i.e., $8X = (4X, 4X)$). The main benefit of this PPG method is that the latency is constant (i.e., equal to that of precomputation of $4X$ or twice that of $2X$) and does not depend on the number of digits in the multiplicand. However, the depth of partial product array is doubled.

Example 1 (Double-BCD multiples): Fig. 2 depicts a 4×4 decimal multiplication. Easy multiples $5X$ and $2X$ are represented by 5-digit decimal numbers F and T , respectively. Hard multiple $7X$ ($6X$) consists of two equally weighted components F and T (F and X). In a general multiplication circuit, where the value of multiplier digits is not known beforehand, the easy multiples $5X$ and $2X$ should be augmented by a 0-word as the second component. ■

			X^3	X^2	X^1	X^0
	\times	6	7	2	5	
		F^4	F^3	F^2	F^1	F^0
		0	0	0	0	0
	T^4	T^3	T^2	T^1	T^0	
		0	0	0	0	0
	F^4	F^3	F^2	F^1	F^0	
	T^4	T^3	T^2	T^1	T^0	
	F^4	F^3	F^2	F^1	F^0	
	X^4	X^3	X^2	X^1	X^0	

Fig. 2: An example of double-BCD PPG

An alternative PPG scheme is offered in [18], where $\pm X$, $\pm 2X$, $5X$, and $10X$ are precomputed. Eqn. set 2 shows the generation of other required multiples as double-BCD numbers. Lang et al. use a BCD carry-save adder to convert, in constant time, these multiples to BCD-carry-save numbers, where each digit is composed of a main BCD part and a stored carry bit. For consistency with other methods, we take the liberty of considering the latter carry-save addition as the first reduction level of the PPR step.

$$\begin{aligned} 3X &= (2X, X), & 4X &= (2X, 2X), & 6X &= (5X, X), \\ 7X &= (5X, 2X), & 8X &= (10X, -2X), & 9X &= (10X, -X) \end{aligned} \quad (2)$$

The slowest multiple, in this scheme, is $-2X$. The negation operation requires 9's complementation, which entails a two-level logic for BCD operands [6]. This is relaxed in the work by Vazquez et al. [20], where three designs of decimal multiplication are discussed. The fastest design (called radix-5) uses the same PPG method as in [18], but directly produces 4-2-2-1 encoded multiples. This encoding allows $-2X$ and $-X$ to be simply generated by bitwise inversion of the digits of the corresponding positive multiples. Note that $4X$ no longer serves as a component of $8X$ and $9X$. Therefore, $4X$ can now be generated as a double-BCD number (i.e., $(2X, 2X)$) and is no more the slowest multiple.

3.1 THE NEW METHOD FOR COMPUTATION OF DOUBLE-BCD MULTIPLES

There are two distinct operations that contribute to the latency of partial product generation:

- **Precomputation:** The slowest multiple in [11] is $4X$, and that of [18] and [20] is $-2X$.
- **Selection:** [11] and [18] use standard multiplexers, while [20] uses a faster custom designed selector.

All the required multiples may be computed, as double-BCD numbers, from the fast easy multiples X , $2X$, and $5X$, except for $8X$ and $9X$. These two exceptions, however, are computed as double-BCD multiples $(4X, 4X)$ and $(4X, 5X)$ in [11], and as $(10X, -2X)$ and $(10X, -X)$ in [18] and [20]. The following disadvantages may be recognized in these methods:

- i) Slow $4X$ generation in [11]
- ii) Slow $-2X$ generation in [18]
- iii) Additional PPR complexity in [18 and 20] due to sign bits and additional ten's complementation bits in position 0 of partial products

To circumvent these disadvantages, we propose the direct computation of $E = 8X$ and $N = 9X$. Let $X^i = x_3 x_2 x_1 x_0$ represent the i^{th} digit of the multiplicand. Also let $E = 10E_h + E_l$, $N = 10N_h + N_l$, $E_h^i = e_{40} e_{30} e_{20} e_{10}$, $E_l^i = e_{3e} e_{2e} e_{1e} e_0$, $N_h^i = n_{40} n_{30} n_{20} n_{10}$, $N_l^i = n_{3n} n_{2n} n_{1n} n_0$, where E_h^i , E_l^{i+1} , N_h^i , and N_l^{i+1} , all weigh 10^{i+1} . Therefore, E and N are double-BCD multiples. The e and n bit-variables of the digits of E and N can be easily derived from straightforward truth tables. The results are shown by Eqn. sets 3 and 4, respectively.

$$\begin{aligned}
e_0 &= 0, & e_1 &= \overline{x_2 x_1 x_0} + \overline{x_2 x_1 x_0} + x_3 x_0 + \overline{x_2 x_1 x_0}, \\
e_2 &= \overline{x_3 x_0} + \overline{x_1 x_0} + \overline{x_2 x_1}, \\
e_3 &= \overline{x_3 x_2 x_1 x_0} + \overline{x_2 x_1 x_0}, \\
e_{10} &= e_1, e_{20} = \overline{x_3 + x_2 x_1 x_0} + \overline{x_2 x_1 x_0}, \\
e_{30} &= \overline{x_3 + x_2 x_1} + \overline{x_2 x_0}, & e_{40} &= e_0 \\
n_0 &= x_0, & n_1 &= \overline{x_3 x_0} + \overline{x_2 x_1 x_0} + \overline{x_1 x_0}, \\
n_2 &= \overline{x_2 x_1 x_0} + \overline{x_2 x_1} + \overline{x_2 x_0} \\
n_3 &= \overline{x_3 x_2 x_1 x_0} + \overline{x_2 x_1 x_0} \\
n_{10} &= \overline{x_3 x_0} + \overline{x_2 x_0} + \overline{x_1 x_0}, & n_{20} &= n_1 \\
n_{30} &= \overline{x_3 x_0} + \overline{x_2 x_0} + \overline{x_2 x_1}, & n_{40} &= \overline{x_3 x_0}
\end{aligned} \tag{3}$$

$$\begin{aligned}
n_0 &= x_0, & n_1 &= \overline{x_3 x_0} + \overline{x_2 x_1 x_0} + \overline{x_1 x_0}, \\
n_2 &= \overline{x_2 x_1 x_0} + \overline{x_2 x_1} + \overline{x_2 x_0} \\
n_3 &= \overline{x_3 x_2 x_1 x_0} + \overline{x_2 x_1 x_0} \\
n_{10} &= \overline{x_3 x_0} + \overline{x_2 x_0} + \overline{x_1 x_0}, & n_{20} &= n_1 \\
n_{30} &= \overline{x_3 x_0} + \overline{x_2 x_0} + \overline{x_2 x_1}, & n_{40} &= \overline{x_3 x_0}
\end{aligned} \tag{4}$$

The overall latency of the latter computation is determined by e_1 . In Subsection 6.1, we will show that generation and selection of the slowest multiple in the proposed design (i.e., $8X$) is faster than that of the slowest one in the other two most recent methods (i.e., $-2X$). Furthermore, lack of signed multiples in the proposed design obviates the need for extra sign-bit and enforced carry-in for ten's complementation. Table II summarizes the characteristics of the four PPG methods discussed above.

TABLE II: DOUBLE-BCD-MULTIPLE COMPUTATION METHODS

Method; References	Multiples	Slowest multiple	2-component multiples
Erle et al.; [11]	$0, X, 2X, 4X, 5X$	$4X = 2 \times (2 \times X)$	$(3, 6-9)X$
Lang et al.; [18]	$-2X, -X, 0, X, 2X, 5X, 10X$	$-2X = 2 \times (-X)$	$(3, 4, 6-9)X$
Vazquez et al.; [20]	$-2X, -X, 0, X, 2X, 5X, 10X$	$-2X = -(2 \times X)$	$(3, 4, 6-9)X$
Proposed	$0, X, 2X, 5X, (8X_i + 8X_h), (9X_i + 9X_h)$	$8X = 8 \times X$	$(3, 4, 6-9)X$

Recall that each partial product has two components (e.g., $W = U + V$). To generate these components we use, as shown in Fig. 3, two selectors S_1 and S_2 . The decoder input Y^j is the j th digit of the multiplier. The boxes denoted as $T, F, E,$ and N represent the logic for multiples $2X, 5X, 8X,$ and $9X,$ respectively. Note that some of these boxes use both X^i and X^{i-1} , and the boxes denoted by E_h^{i-1} and N_h^{i-1} generate the higher portions of 8 and 9 multiples of X^{i-1} .

The selector bits come from a special decoder that transforms the bits of Y^j to six signals $d_1, d_2, d'_2, d_5, d_8,$ and d_9 . In fact, as shown in Table III, the decimal value of Y^j is decoded to the d signals for selectors S_1 and S_2 .

TABLE III: SELECTION OF THE TWO COMPONENTS OF EACH MULTIPLE

Y^j	0	1	2	3	4	5	6	7	8	9
Activated signals into S_1 and S_2	0, 0	$d_1, 0$	0, d_2	d_1, d_2	d'_2, d_2	0, d_5	d_1, d_5	d'_2, d_5	d_8, d_8	d_9, d_9
U	0	X	0	X	T	0	X	T	E_l	N_l
V	0	0	T	T	T	F	F	F	E_h	N_h

These signals are computed by Eqn. set 5 and select the corresponding multiple of X , where $\alpha, \beta,$ and γ are auxiliary common variables. For example, for $Y^j = 7$, the signals d'_2 and d_5 are activated that lead T and F to the outputs of S_1 and S_2 , respectively.

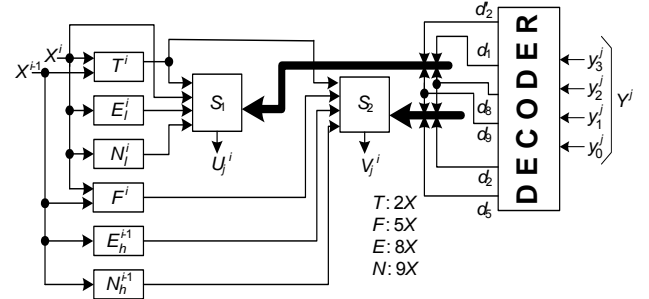


Fig. 3: A digit-slice $X^i Y^j$ of the proposed circuit for PPG.

$$\begin{aligned}
\alpha &= \overline{y_3^j + y_2^j}; & \beta &= \overline{y_1^j + y_0^j}; & \gamma &= y_2^j \cdot y_1^j \\
d_1 &= \alpha y_0^j + \gamma y_0^j \\
d_2 &= \alpha y_1^j + \beta y_2^j; & d'_2 &= \gamma y_0^j + \beta y_2^j \\
d_5 &= y_2^j y_0^j + \gamma \\
d_8 &= y_3^j \overline{y_0^j}; & d_9 &= y_3^j y_0^j
\end{aligned} \tag{5}$$

The details of a bit-slice of selection boxes S_1 and S_2 are depicted in Fig. 4, where $x^i, t^i, f^i, e^i,$ and n^i denote a bit in the i th digit of multiples $X, 2X, 5X, 8X,$ and $9X,$ respectively, and u^i and v^i stand for the corresponding bit in the components U^i and V^i of the selected multiple.

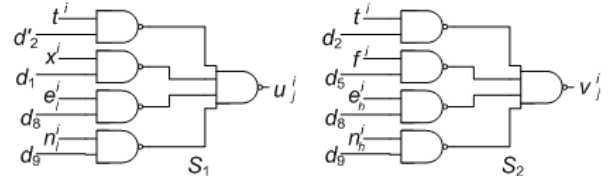


Fig. 4: The logic for a bit-slice of selection boxes of Fig. 3.

4 BCD PARTIAL PRODUCT REDUCTION (PPR)

BCD partial product reduction, within sequential BCD multiplication, iteratively adds one BCD operand to an accumulated partial product [11]. However, the bulk of work in parallel BCD multiplication lends itself to parallel PPR, which is a special case of parallel multi-operand addition. The latter general problem for BCD operands has been studied in [9] and [10], and specifically as PPR in [18], [20] and [19].

In the BCD multiplier that is designed by Vazquez et al. [20], the partial products are represented in decimal 4-2-2-1 encoding. The reduction block takes three equally weighted 4-2-2-1 encoded decimal digits and produces two such digits, also equally weighted.

This clever and fast reduction technique, which is further explained in the Appendix, ends up with two double-length 4-2-2-1 encoded decimal operands. These are recoded to two BCD operands, just before the final product computation. This method is rightly claimed to perform faster than the method used in [9]. Moreover, according to our analysis in the Appendix, it is faster than the more recent Dadda's scheme [10]. However, the most recent decimal PPR method [19], also explained in the Appendix, is shown to be faster than that of [20]. Therefore, we shall compare our proposed PPR method mainly with those of [18], [20], and [19]. In Subsection 4.1 we analyze the PPR method of [18] to pave the way for proposing the corresponding modifications in Section 4.2.

4.1 ANALYSIS OF PPR METHOD DUE TO LANG ET. AL

The PPR method of [18] uses BCD carry-save adders and binary to BCD counters. For a thorough analysis of such method we briefly explain the basic binary PPR and similarly build up the BCD PPR method.

4.1.1 BINARY PPR

The conventional binary PPR uses several levels of carry-save adders (CSA) or (4:2) compressors. Each bit-slice of a CSA is implemented by a binary full adder, where the carry input comes from the previous bit-slice of an upper level in the reduction tree. This full adder produces a sum-bit and a stored carry. The reduction process results in a carry-save (CS) representation of the product, which is converted to the final binary sum via a carry-propagating adder. For an $n \times n$ multiplication the reduction tree can be arranged such that the n least significant bits of the product are readily available before the final carry propagating addition begins on the most significant half. In a nutshell, binary PPR can be organized:

- Via CSA tree or (4:2) compressors
- To result in a half binary and half CS product

4.1.2 BCD PPR

The BCD partial product reduction can be organized similar to binary PPR. A digit-slice of a BCD-CSA may be designed as a BCD full adder (BCD-FA). This adder cell receives two BCD digits and a decimal carry, from the previous digit-slice, and produces a BCD sum digit and a stored 1-bit decimal carry. This seems to lead to a 2-to-1 reduction ratio VS the 3-to-2 reduction ratio of binary CSA. Unfortunately however, there are extra decimal carry bits that cannot be absorbed by the BCD-CSAs throughout the reduction process. These can be counted via some $(u:4)$ binary counters ($u \leq 9$) that produce extra BCD digits; hence upsetting the desired 2-to-1 reduction

ratio. For example, a 32-to-2 reduction of binary operands can be done in 4 or 8 levels via (4:2) compressors or CSAs, respectively. However, the 32-to-2 BCD reduction process, as just described, uses 7 levels of reduction as designed in [18].

Example 2 (BCD-CSA): Fig. 5 depicts a digit-slice of the reduction of six BCD numbers to three decimal carries and one BCD number, where the boxes with dashed (solid) lines represent the source (target) of a reduction step. ■

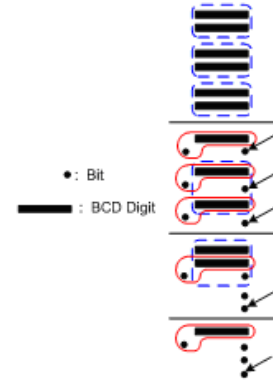


Fig. 5: A digit-slice of reduction of 6 BCD operands.

Reduction of m BCD operands with BCD-FAs leads to one BCD digit and $\lfloor m/2 \rfloor$ equally weighted decimal carries, which should be in turn converted to BCD digits. This may be delegated to a binary-to-BCD counter that is formally defined below:

Definition 1 (Binary-to-BCD counter): A $(u:4)$ binary-to-BCD counter counts the number of 1s among the u (≤ 9) equally-weighted bits $\{b_{u-1}, \dots, b_1, b_0\}$ and produces a BCD digit $W = \sum_{i=0}^{u-1} b_i$. ■

Fig. 6 illustrates the function of a $(9:4)$ binary-to-BCD counter in dot notation and the required circuitry with latency equal to that of seven logic levels.

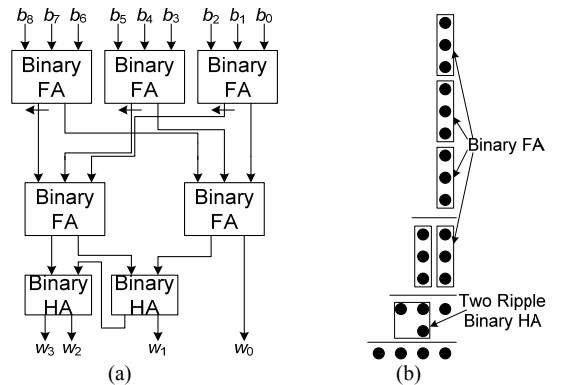


Fig. 6: The circuit (a) and dot-notation (b) of a binary $(9:4)$ counter.

The equally weighted decimal carries, which are not used by BCD-FAs of the same reduction level, may be converted to a BCD-digit as soon as the number of carries amounts to 9 or more. This process obeys the following recurrence, where d_i , c_i , and u_i are the number of BCD digits, total number of carries and unused carries in the i^{th} level, respectively, and $|u_i|_9$ stands for u_i modulo 9.

Initialize: $d_0 = m; u_0 = 0$

Recurrence:

$$\begin{aligned} d_{i+1} &= \lfloor d_i/2 \rfloor + \lfloor u_i/9 \rfloor; & c_{i+1} &= \lfloor u_i \rfloor_9 + \lfloor d_i/2 \rfloor; \\ u_i &= \max(c_i - \lfloor d_i/2 \rfloor, 0) \end{aligned} \quad (6)$$

In the i^{th} level of reduction, $\lfloor d_i/2 \rfloor$ carries are used by BCD-FAs. Each nine carries are reduced to one BCD digit by a (9:4) counter, or possibly a smaller counter required by particular circumstances (e.g., for the carries out of the BCD-FAs of the last levels).

The PPR method of [18] is actually a 32-operand concrete model for the above analysis. However, there are four design decisions that can enhance the overall speed:

- a) **Use of simplified first-level BCD-CSA:** The first level BCD-FAs of [18] are supplied with carry-in signals due to ten's complement representation of negative multiples. However, the proposed partial product generation scheme, in Section 3, obviates the need for such carry-in signals. This is due to lack of negative multiples. Therefore, we can use simplified BCD-FAs (i.e., with no carry-in signal) in the first reduction level to particularly gain some speed.
- b) **Use of improved reduction cell:** The fastest known BCD-FA, to the best of our knowledge, is the one introduced in [6]. We modify this BCD-FA such that its sum-path latency is reduced and use it as the main reduction cell.
- c) **Double-BCD reduced partial product:** The proposed design, in the next subsection, uses double-BCD representation at the end of reduction process, instead of the BCD-CS encoding of [18]. This slows down the final product computation, but leads to fewer number of reduction levels.
- d) **Wallace-like organization of the reduction tree:** The reduction process of [18], for $n \times n$ multiplication results in $2n$ -digit BCD-CS encoding of the final redundant partial product, where each redundant decimal digit in $[0, 10]$ is represented by an equally weighted pair of a BCD sum and a decimal carry. This redundant sum can be converted to standard BCD sum via a simplified BCD adder. However, the BCD reduction tree can be organized similar to that of Wallace tree [26] such that at the end of each reduction level one digit of the final product is already available. Therefore, the final carry-propagating chain is shorter than $2n$ digits.

In a nutshell, the BCD reduction process can be organized:

- Via BCD-FA reduction cells
- To result in l BCD and $2n - l$ BCD-CS product, where l is the number of reduction levels

Example 3 (PPR of a 4×4 multiplication): Fig. 7 depicts a Wallace-like reduction tree organization for a 4×4 multiplication with four 5-digit double-BCD partial products. At the end of the reduction process the three least significant digits of the product are readily available; thereby reducing the final carry propagation length. This is how it works: Wherever there are not enough inputs for full utilization of a BCD-FA we do use a simpler reduction cell with only one BCD input. This greedy reduction strategy resembles the earliest possible use of half adders in Wallace-tree reduction of binary partial products [26]. Therefore, a shorter double-BCD component of the reduced partial product is achieved at the end of reduction process. For example, see Level #1 of Fig. 7, where a simplified reduction cell is used in the third column from the right. A design decision that favors savings in area consumption could opt for full utilization of BCD-FAs only as is the case in [18], where no early BCD digit is produced. ■

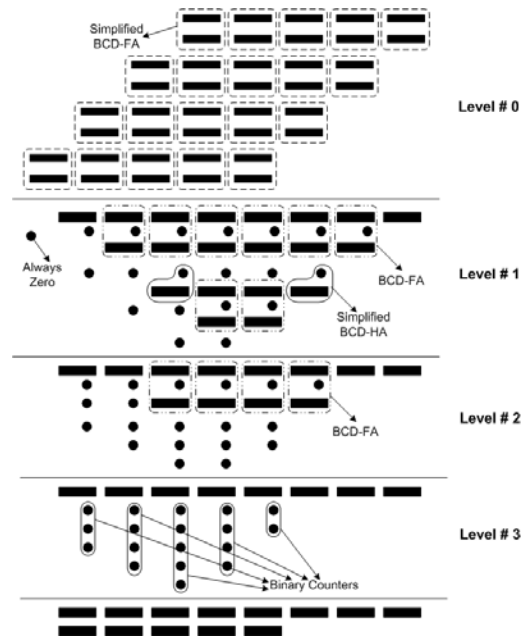


Fig. 7: Wallace-like organization of the BCD reduction tree.

It is important to note that improvement d) on PPR method of [18] is not applicable on the PPR method of neither [20] nor [19]. The reason is simple: The reduction cell of [20] produces two equally weighted BCD 4-2-2-1 digits out of three. It is essentially an in-place reduction method. The reduction method in [19] uses 4-2-2-1 (4:2) compressors that are actually composed of two reduction cells of [20]. Therefore, the same reason applies. However, the method that mimics the conventional binary PPR, which was just clarified via Example 3 and Fig. 7, produces an in-place BCD and a carry to the next decimal position. Therefore, with no carry coming from the right of decimal position i ($i \geq 0$), the rightmost BCD digit that is generated in level $i+1$ remains alone as the BCD digit in position i of the product.

4.2 THE PROPOSED BCD PARTIAL PRODUCT REDUCTION SCHEME

The two full solutions (i.e., [18 and 20]) have designed 16×16 parallel multipliers, with a partial product tree of 32 decimal numbers. Therefore, we also focus on the same size partial product reduction as depicted in Fig. 8 and explained in the following subsection.

4.2.1 32-TO-2 BCD REDUCTION TREE

There are a total of 32 BCD-FAs distributed in the six reduction levels of Fig. 8. The sixteen BCD-FAs, in the first level (i.e., in the rightmost column of Fig. 8), are simplified ones that do not use any carry-in. However, the other sixteen BCD-FAs use only half of the 32 generated carries. The sixteen unused carries should be converted to two BCD digits and fed into the tree. This is illustrated in Fig. 8, where a (9:4) counter and a (6:3) counter are used. These counters start counting the earlier generated carries in advance of the operation of the BCD-FA in the last level. The 16th carry (i.e., the carry-out of the very last BCD-FA) is not directed into the last counter. It is rather routed to a multiplexer that selects between the output b of the last counter and $b+1$. This increment operation, as shown in Fig. 8, is performed in two logic levels and off the critical path as is the case for the multiplexer (also two logic levels) and the final counter.

The result of applying Eqn. set 6 for the case of $d_0 = 32$ and those of similar calculations for the reduction method in [18] are shown in Table IV, above. For the sake of easier comparison however, the BCD-FAs used in PPG step of [18] are considered as level 0 of PPR.

4.2.2 68-TO-2 BCD REDUCTION TREE

Given that the IEEE 754-2008 draft standard for decimal floating point arithmetic proposes 16- and 34-digit BCD representations [5], we show how the above reduction method can be efficiently applied to 34×34 BCD multiplication. A similar unused carry handling is used in Fig. 9, where 33 out of the 34 unused carries are fed into a (6:3) and three (9:4) counters.

The only remained unused carry acts as the selector of the multiplexer. The total delays of the increment block and the last counter in Figs. 8 and 9, amount to 7 and 10 logic levels, respectively. This overlaps with the operation of two sum-cascaded BCD-FAs (12 logic levels). Note that the carry-out of the final BCD-FA arrives, into the multiplexer, two logic levels earlier than the corresponding sum digit is delivered. The reason is that the carry path in the BCD-FA of [6] goes through four logic levels VS six for the sum path. Therefore, only the BCD-FAs fall within the critical delay paths of Figs 8 and 9. Note that the rightmost counter in Fig. 9 is intentionally a (6:3) counter (with the latency of five logic levels) just to keep it off the critical path. The leftmost counter, instead, is a (9:4) counter, which also stays out of the critical path.

4.2.3 IMPROVED REDUCTION CELL

The fastest BCD-FA, in the sum path, is believed to be the one reported in [6], where based on using at most 4-input gates, six (four) logic levels is the delay of sum (carry) path. This is indeed an excellent performance given that the sum and carry delay of a 4-bit binary carry look-ahead (CLA) logic is roughly five logic levels. A better latency evaluation with Logical Effort analysis shows that the sum delay of BCD-FA is 6.88 FO4 VS the 6.60 FO4 for CLA. However, since there is a chain of sum paths of the BCD-FAs in the PPR tree, even a small latency improvement can lead to a considerable speed enhancement. Therefore, we are motivated to modify the implementation of [6] to obtain a faster BCD-FA on the sum path. Let $x_3x_2x_1x_0$, $y_3y_2y_1y_0$, c_0 and $s_3s_2s_1s_0$ denote the two BCD inputs, carry-in and the output sum, respectively. Also let $p_i = x_i + y_i$, $g_i = x_i y_i$ and $h_i = x_i \oplus y_i$ denote the propagate, generate and half-sum signals, respectively, for $0 \leq i \leq 3$. The critical delay path of the BCD-FA runs from x_2 and y_2 to s_2 . Therefore, aiming at a possible improvement, we focus on Eqn. 7 for s_2 , which is reproduced from [6], where $c_1 = g_0 + p_0c_0$ is the carry into position 1.

$$S_2 = \overline{p_2}g_1 + \overline{p_3}h_2\overline{p_1} + (\overline{p_3}\overline{p_2}p_1 + g_2g_1 + p_3p_2)c_1 + (g_3 + h_2h_1)\overline{c_1} \quad (7)$$

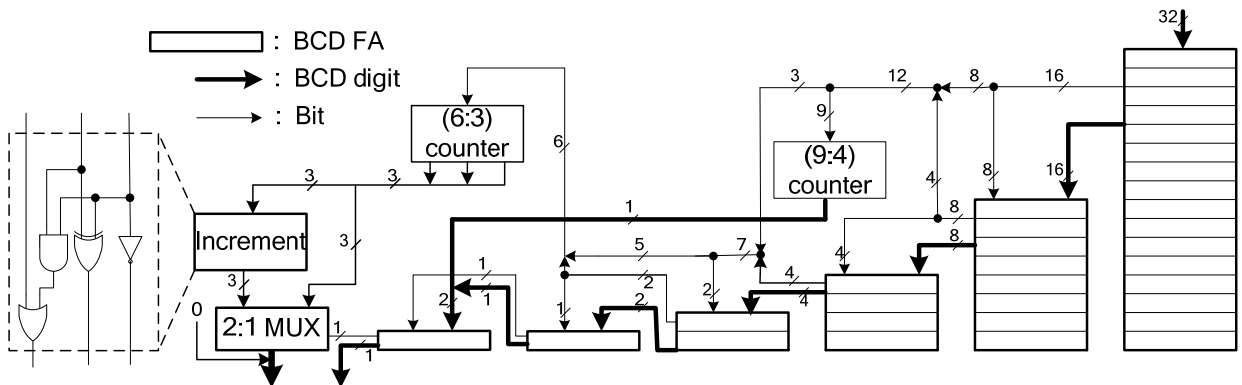


Fig. 8: The deepest digit-slice of the 32-to-2 BCD reduction tree.

TABLE IV: REDUCTION OF 32 BCD OPERANDS FOR A DIGIT-SLICE.

Level i	# of BCD-FAs		# of Counters		d_i	c_i	u_i	d_i	c_i	u_i
	proposed	Lang et al. [18]	proposed	Lang et al. [18]	proposed			Lang et al. [18]		
0	16	16	0	0	32	0	0	32	0	0
1	8	8	0	1	16	16	8	16	16	8
2	4	4	1	0	8	16	12	9	8	4
3	2	2	0	0	5	7	5	5	8	6
4	1	1	0	0	3	7	6	3	8	7
5	1	1	0	1	2	7	6	2	8	7
6	0	1	1	0	1	7	7	2	1	0
Remained BCD digits and carries					2	0	0	1	1	1

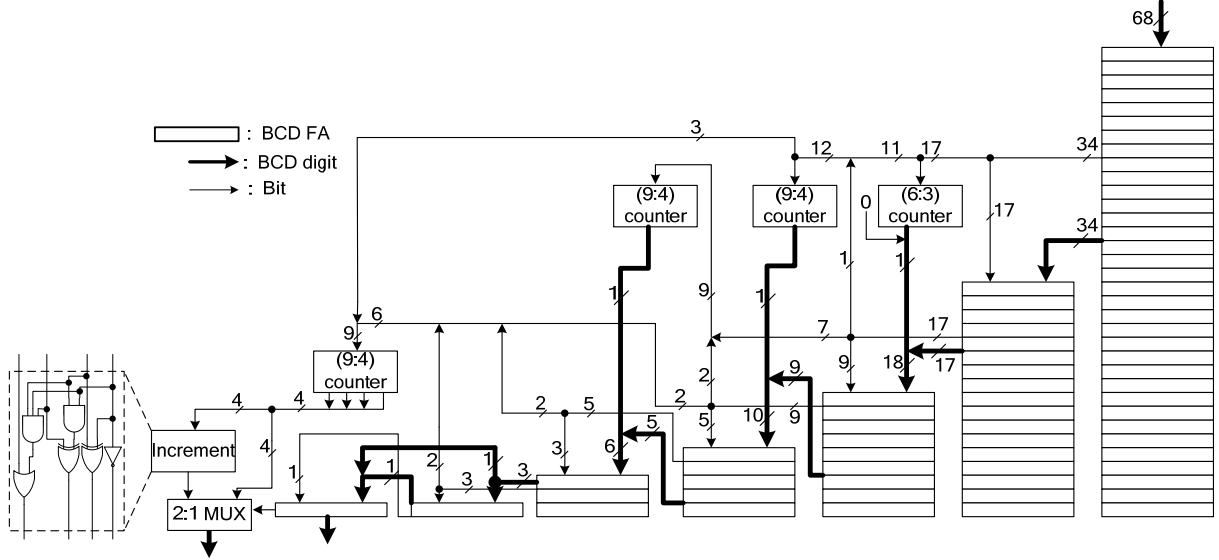


Fig. 9: A digit-slice of 68-to-2 BCD reduction tree.

Computation of the parenthesized factor $\Phi = (\overline{p_3} \overline{p_2} p_1 + g_2 g_1 + p_3 p_2)$ in Eqn. 7 lies within the critical path. A delay optimized computation of Φ , after obvious replacements in $\overline{p_3} \overline{p_2} p_1$, uses a NAND5 and a NAND4 in the critical path. However, since in a BCD digit $x_3 x_2 x_1 x_0$, the two bits x_3 and x_1 cannot simultaneously be equal to 1, we have $\overline{x_3} x_1 = x_1$ and similarly $\overline{y_3} y_1 = y_1$. The latter equalities, that are not used in [6], can be enforced when the three factors in $\overline{p_3} \overline{p_2} p_1$ are disintegrated into their constituent variables, as in Eqn. 8.

$$\begin{aligned} \overline{p_3} \overline{p_2} p_1 &= \overline{x_3} \overline{y_3} (x_1 + y_1) \overline{x_2} \overline{y_2} = (\overline{y_3} x_1 + \overline{x_3} y_1) (\overline{x_2} \overline{y_2}) \\ &= \overline{y_3} x_1 \overline{x_2} \overline{y_2} + \overline{x_3} y_1 \overline{x_2} \overline{y_2} \end{aligned} \quad (8)$$

The computation of Φ using Eqn. 8 uses two NAND4 gates in the critical path. This modification reduces the latency to 6.05 FO4, which is indeed less than that of a 4-bit CLA; a surprising result due to simplifications that stem from interdependencies between the bits of a BCD digit. The same modification can be applied to the logic of the simplified BCD-FA (used in the first reduction level) that reduces the latency from 6.49 FO4 to 5.89 FO4.

Recalling the discussions in the above three subsections, particularly the design decisions a) to d) in Subsection 4.1.2, five main differences between the proposed method and that of [18] are recognized:

- Use of simplified BCD-FAs in the first reduction level.
- Use of the improved faster BCD-FA.
- The main influential difference occurs in Level 6 (see Table IV), where the proposed method uses an off-the-critical-path counter instead of a BCD-FA used in [18].
- The result of PPR in [18], for an $n \times n$ multiplication is a BCD carry-save number with $2n$ digits. However, the proposed PPR method leads to a $2n$ digit result with l least significant BCD digits and $2n - l$ most significant double-BCD digits, where l is the number of reduction levels.
- There is a double-BCD number left at the end of reduction step in the proposed scheme VS a BCD-CS number in [18]. However, as will be pointed out in Subsection 6.3, taking advantage of the BCD adder of [8] blurs the potential advantage of simpler output of reduction step in [18].

The number of reduction levels for 32-to-2 BCD reduction is seven in [18], eight in [20], five in [19] and six in the proposed method. Since there is no indication as to the specific BCD-FA used in [18] we assume, for a fair comparison, that the fastest available six logic level BCD-FA [6] was used. In [20], the delay of each level, as shown in the Appendix, is five logic levels.

Finally, the simplified BCD-FA of the first level of the proposed method takes four logic levels and six logic levels is assumed for the BCD-FAs used in the rest of reduction levels. In [19] there are four levels of 4-2-2-1 BCD (4:2) compressors and one level of 4-2-2-1 CSA cell, such that the critical path goes through 6 carry- and 3 sum-paths with 36 (i.e., $6 \times 5 + 3 \times 2$) logic level delay.

Summing up in Table V, it is shown that the overall delay of the proposed method is the least, while more accurate Logical Effort comparison is given in Section 5. The rationale for N/A entry in Table V is that the PPR method of [19], as mentioned therein, is not efficiently suitable for non-power-of-two number of operands.

TABLE V: COMPARISON OF THE LATENCY OF THE PROPOSED PPR METHOD WITH THE PREVIOUS ONES

# of original BCD operands	32	68
Delay (logic levels)		
Proposed	$4+5 \times 6=34$	$4+6 \times 6=40$
Lang et. al. [18]	$7 \times 6 = 42$	$8 \times 6 = 48$
Vazquez et. al. [20]	$8 \times 5 = 40$	$10 \times 5 = 50$
Castella. et al. [19]	$6 \times 5 + 3 \times 2 = 36$	N/A

5. FINAL PRODUCT COMPUTATION

The outcome of partial product reduction for 16×16 decimal multiplication is not the same in the proposed and the two previous full solutions. We discuss the input to the third step and the respected final carry propagating adders in the two following subsections.

5.1 INPUT TO THE THIRD STEP

The input to the final product computation step can be expressed as a decimal number $P = 10^6 H + L$, where the 26-digit (6-digit) number H (L) is described below for each method:

- i) **Proposed:** H is a double-BCD number, and L is a single BCD number. Therefore, only a 26-digit BCD adder is required to complete the computation of the final BCD product. The corresponding reduction tree organization, but for a 4×4 multiplication, was shown in Fig. 7.
- ii) **Lang et al. [18]:** Both H and L are BCD-CS numbers. Therefore, the final product computation is done via a BCD-CS to BCD converter that requires carry propagation along 32 BCD digits.
- iii) **Vazquez et al. [20]:** Both H and L are double-BCD numbers. Therefore, a 32-digit BCD adder is required to compute the final product.

5.2 THE FINAL CARRY-PROPAGATING ADDER

The final product computation in Method ii) is naturally faster due to its simpler input VS the full BCD operands in Methods i) and iii). The final adder of Method iii) uses a simple preprocessing step that paves the way for taking advantage of standard word-wide fast

carry generation techniques exactly as is used in binary addition. In this preprocessing step, each BCD digit of one of the operands is speculatively added by 6 in a digit-parallel manner [8] that overlaps with the last part of the PPR step. We use the same technique in Method i). In order to save the time for $+6$ operation, we replace the (6:3) counter of Fig. 8 by a so called excess-six-(6:3) counter. The corresponding logic is depicted in Fig. 10, where HA (HA*) blocks are standard half adders (pseudo half adders that compute $\text{sum}+1$). With this modification, the incrementer of Fig. 8 has now 4 input bits. However, despite the added complexity, the overall delay of the counter path amounts to 9 logic levels, which is yet one less than that of the other path ending at the multiplexer (see Fig. 8). More reliable performance measures based on Logical Effort analysis are given in Subsection 6.3.

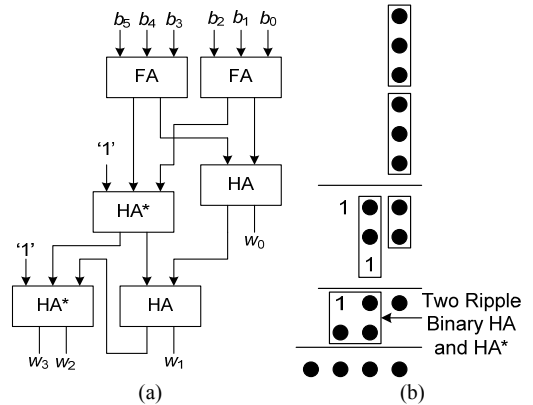


Fig. 10: Circuit (a) and dot-notation (b) of an excess-six-(6:3) counter.

The final 128-bit (32 BCD digits) adder in Method iii) uses a 7-level parallel prefix quaternary tree (QT) adapted from [27]. The latency of this QT amounts to 14.72 FO4. However, using a similar architecture in Method i), where there are only 26 BCD digits in the final addition, leads to a faster, though 7-level, QT with a latency of 14.33 FO4. The slight improvement is due to less branching in the critical path of the QT tree.

The adder used in Method iii) is based on [8], where a combined binary/decimal adder/subtractor is offered. We learn from the content of Table 1 in [8] that, for 64-bit operands, a simple parallel prefix Kogge-Stone (KS) [28] tree is faster than QT due to less branching in the former. However, the final choice in [20] is QT. The reason is that the latency of operand setup logic, due to the combined nature of the adder, is much less in the QT approach versus KS. Given that in the current work we do not need the combined binary/decimal adder, we follow [29] and use a 104-bit (corresponding to 26 BCD digits) KS carry network that would be faster than its QT counterpart.

6. COMPARISON WITH PREVIOUS WORKS

The full solution due to [20] has used Logical effort (LE) model [21]. Therefore, for a more accurate comparison than what is provided in Tables IV and V, we have used LE for static CMOS gates to estimate area and time measures of the proposed and previous works.

The method of Logical Effort is ideal for evaluating alternatives in the early design stages and provides a good starting point for more intricate optimizations [21]. Therefore, following [20], neither we undertake optimizing techniques such as gate sizing, nor consider the effect of interconnections. We rather allow gates with the drive strength of the minimum sized inverter, and assume equal input and output loads. However, the other full solution due to [18] has reported synthesis results based on Synopsis Design Compiler. Therefore, we have also synthesized the whole 16×16 multiplier based on the proposed design.

In the LE analysis, latency is commonly measured in FO4 units (i.e., the delay of an inverter with a fan-out of four inverters), and minimum size NAND2 gate units are assumed for area evaluation. For a fair comparison, based on the same assumptions, we have undertaken our own Logical Effort analysis on all methods. For instance, the delay of an XOR gate is $[2 \times \text{delay}(\text{NAND2})]$, and that of sum and carry path of a full adder are $[4 \times \text{delay}(\text{NAND2})]$ and $[\text{delay}(\text{NAND2}) + \text{delay}(\text{NAND3}) + \text{delay}(\text{NOT})]$, respectively. Moreover, for better evaluation of different techniques used in the three steps of multiplication, we present the LE comparison in three subsections followed by a comparison table for the overall evaluation.

6.1 PARTIAL PRODUCT GENERATION (PPG)

Recalling Table II on the characteristics of double-BCD multiple computation, we provide below the Logical Effort measures for latency and area of each method.

6.1.1 THE SCHEME OF LANG ET AL. [18]

Given that we have virtually moved one BCD carry-save addition from the PPG to PPR step (see Section 3, after Example 1), the critical path now consists of the following latency items that amount to 6.64 FO4.

- Pre-computation of multiple $-2X$ in two steps:
 - Computation of $2X$
 - BCD 9's complementation
- Selection of multiples by means of a 5-to-1 multiplexer

6.1.2 THE SCHEME OF VAZQUEZ ET AL. [20]

The multiples used in this method are the same as the two component multiples of [18]. However, due to 4-2-2-1 representation of partial products the latency of negative multiples is less than that of [18]. The total latency of PPG in this method, as we have evaluated, equals 5.47 FO4. The critical path consists of:

- Computation of $2X$
- Selection of multiples (two NAND2 gates)
- Negation of multiples, where necessary (an XOR gate)

6.1.3 THE PROPOSED SCHEME

As in [20], we take advantage of a custom designed selector that is faster than a direct multiplexer based one.

The total latency of the proposed PPG method is equal to 5.28 FO4, where the critical path consists of the two items below. The latter is achieved with 16-fold replication of the decoder of Fig. 3 per multiplier's digit. The other extreme would use only one decoder that leads to 6.29 FO4 as the PPG latency.

- Computation of E_i , the less significant component of multiple $8X$ (Eqn. 3)
- Selection of multiples (Fig. 4)

Table VI compares the latency and area of the three discussed methods.

TABLE VI: COMPARISON OF PARTIAL PRODUCT GENERATION FOR THREE 16-DIGIT MULTIPLIERS

Schemes	Delay (FO4)	Ratio	Area (NAND2)	Ratio
Proposed	5.28	1	43233	1
Lang et al. [18]	6.64	1.26	30000	0.69
Vazquez et al. [20]	5.47	1.03	15000	0.34

6.2 PARTIAL PRODUCT REDUCTION (PPR)

Recalling the discussion in Section 4 and the logic-level latency comparison in Table V, we provide below a Logical Effort analysis of the four PPR methods for 16×16 multiplication (i.e., reduction of 32 BCD operands to a double-BCD product).

6.2.1 THE PPR SCHEME OF LANG ET AL. [18]

Recalling the explanation in 6.1.1, the PPR in this method is performed via seven levels of BCD carry-save adder and two, off the critical path, (8:4) binary-to-BCD counters. The outcome of reduction process is a BCD carry-save number. The latency of each level, equal to that of one BCD-FA of [6], is evaluated as 6.88 FO4. A simple, but pessimistic, evaluation would lead to $7 \times 6.88 = 48.16$ FO4 latency. However, the least delay along the critical path of the overall 7-level PPR, is evaluated as 42.02 FO4.

6.2.2 THE PPR SCHEME OF VAZQUEZ ET AL. [20]

This method uses a special reduction block that reduces three equally weighted 4-2-2-1 decimal digits to two such digits that are also equally weighted. This block is implemented as a quadruple of binary full adders followed by a decimal multiply-by-2 logic (see Fig. 11 in the Appendix). Eight levels of reduction are required for 16×16 multiplication. The latency of each reduction level is evaluated as 5.27 FO4. However, the least delay along the critical path of the 8-level reduction amounts to 40.93 FO4.

6.2.3 THE PPR SCHEME OF CASTELLANOS ET AL. [19]

The critical delay path of PPR scheme of [19], as explained in the Appendix, does not always follow the carry-path of the reduction cells. Therefore, the least delay along the critical path, as we have evaluated, amounts to 36.98 FO4.

6.2.4 THE PROPOSED PPR SCHEME

The proposed scheme, similar to that of [18], uses BCD-FA as the main building block. However, there are some differences:

- **Number of reduction levels:** The reduction tree in this scheme is composed of six levels of BCD carry-save adders.
- **Simplified BCD-FA:** Given that all the precomputed multiples are positive, the first level uses simplified BCD-FAs (i.e., with no carry-in signals) with 5.89 FO4 latency.
- **Use of the modified BCD-FA:** We modified, in Section 4.2.3, the fastest known BCD-FA of [6]. This reduces the latency of the sum path from 6.88 to 6.05 FO4.
- **Narrower partial products:** Lack of negative partial products obviates the need for sign digits and the tail encoding bits for ten's complementation.

The least delay along the critical path of the 6-level reduction tree is evaluated as 32.35 FO4.

Table VII summarizes the above discussion and shows the relative reduction speed of the proposed design VS the other three evaluated schemes.

TABLE VII: COMPARISON OF 32-TO-2 PARTIAL PRODUCT REDUCTION

Schemes	Delay (FO4)	Ratio	Area (NAND2)	Ratio
Proposed	32.35	1	33163	1
Lang et al. [18]	42.02	1.30	35700	1.07
Vazquez et al. [20]	40.93	1.26	31345	0.94
Castella. et al. [19]	36.98	1.14	33335	1.00

6.3 FINAL PRODUCT COMPUTATION

The outcome of PPR step is a double decimal number that is to be converted to a BCD product. Each of the three methods has its own format of the redundant product; hence different conversion methods to be explained below.

6.3.1 THE 32-DIGIT BCD-CS-TO-BCD CONVERTER OF LANG ET AL. [18]

The last step of the multiplier in [18] converts a redundant BCD carry-save number to an equivalent conventional BCD number. This is done via a 32-digit carry look-ahead simplified BCD adder. The overall conversion latency is evaluated as 12.50 FO4.

6.3.2 THE 32-DIGIT DOUBLE-BCD-TO-BCD CONVERTER OF VAZQUEZ ET AL. [20]

The last step in the multiplier of [20] requires a 32-digit BCD addition. This adder is a 32-digit version of the adder proposed in [8]. The latency is evaluated to be 18.02 FO4.

6.3.3 THE PROPOSED 26-DIGIT DOUBLE-BCD-TO-BCD CONVERTER

The proposed PPR method leads to a reduced partial product with 6 least significant BCD digits and 26 most significant double-BCD digits. Therefore, the required converter is a 26-digit BCD adder. We modified the adder of [8] for 26 decimal digits in such a way that the 104-bit Kogge-Stone parallel prefix tree is used instead of the QT structure. Consequently, the latency of the proposed 26-digit double-BCD-to-BCD converter is evaluated to be 16.00 FO4.

6.4 THE OVERALL PARALLEL BCD MULTIPLIER

This subsection is meant to compare the overall performance of the proposed BCD multiplier with those of the previous works.

For a fair comparison, besides the three full solutions, we take into account the impact of PPR scheme of [19] on the full multiplier of [20], which is referred to in Table VIII as "Modified [20]". The proposed multiplier is 14%, 20% and 13% faster than those of [18], [20] and modified [20], respectively. However, the area consumption of the latter methods is 15%, 36% and 33% less than that of the proposed method, respectively. The additional area in the proposed method is mainly due to our more complex, but faster, PPG method. Besides, the unsigned partial products lead to faster PPR.

TABLE VIII: PERFORMANCE COMPARISON OF FOUR 16-DIGIT PARALLEL BCD MULTIPLIERS

Schemes	Delay (FO4)	Ratio	Area (NAND2)	Ratio
Proposed	53.53	1	79636	1
Lang et al. [18]	61.16	1.14	68000	0.85
Vazquez et al. [20]	64.42	1.20	51655	0.64
Modified [20]	60.47	1.13	53645	0.67

The results in Table VIII are derived following the same Logical Effort analysis and same assumptions on equivalent components.

The overall latency reported in [18] is 2.65 ns based on STM 90 nm standard CMOS technology, where 0.04 ns is equivalent to one FO4 [30]. Therefore, latency of the multiplier in [18] is estimated as 66.25 FO4.

The moderate difference between the latter and our direct FO4 evaluation of the same work in Table VIII (i.e., about 5 FO4) may be justified via the fact that Logical Effort analysis does not take into account the wiring delays. Therefore, we believe that the assumptions under which Vazquez et al. have reached at 92 FO4 delay for the multiplier of [18] must be effectively different. For example, given that [18] provides no implementation and performance characteristics for the employed BCD-FAs in seven cascaded levels, the possible different assumptions on the latency of BCD-FAs might be the source of substantial difference in the total evaluated latencies.

6.5 SYNTHESIS BASED COMPARISON

For more realistic results, we produced VHDL code for the proposed 16-digit parallel BCD multiplier and ran simulations and synthesis using the Synopsis Design Compiler. The target library is based on TSMC 0.13 μm standard CMOS technology. The result showed 3.71 ns latency and 445725 μm^2 area consumption for the proposed 16 \times 16 multiplier. Another synthesis result is reported in [18], where the overall latency is 2.65 ns based on STM 90-nm standard CMOS technology. If we scale up the latter, for 0.13 μm technology, we get 4.30 ns. The scaling is based on 40 ps and 65 ps per one FO4 for 90 nm and 0.13 μm CMOS technologies [30], respectively. Therefore, the proposed 3.71 ns multiplier shows 16% speed advantage over that of [18] (4.30 ns). Note that the synthesis results are normally dependent on many factors such as the applied library or optimizations. Therefore, since we are not aware of the actual circumstances in other methods, the comparison results might be slightly different from what is reported here. For instance, another synthesis result is reported in [29], where the work of [20] is modified in the final addition. This report shows a latency of 2990 ps (54.4 FO4) based on LSI Logic's gflxp 0.11 μm CMOS standard cell library and Synopsys Design Compiler.

Given that the range of a 16-digit BCD number is almost equal to that of a 54-bit binary number, Lang et al. have compared the latency of their 16 \times 16 BCD multiplier with that of a 54 \times 54 binary multiplier. The reported ratio is approximately 1.9. However, the reported latency for the binary multiplier in [18], if scaled to 0.13 μm technology, amounts to 2.27. Therefore, the same ratio in the case of the proposed multiplier is $3.71/2.27 \approx 1.6$.

7. CONCLUSIONS

We assembled a new full solution for parallel decimal multiplication by using the better of the two pioneering full solutions. We developed our own partial product generation (PPG) method, fine tuned the partial product reduction (PPR) method of [18] and adapted the carry-propagating addition method of [20] to suite the outcome of our PPR method.

The proposed PPG logic, which directly computes multiples 2, 5, 8, and 9 of the multiplicand, produces double-BCD unsigned partial products for all multiples and is faster.

In view of PPR, lack of negative multiples due to the proposed PPG method leads to simpler PPR step. Accordingly, we modified the PPR method of [18] and applied it to 34 \times 34 decimal multiplier as well as the more practical 16 \times 16 one. We also improved the BCD-FA of [6] for more speed and used it as the reduction cell of the proposed PPR tree. These modifications lead to less latency than those of all the previous PPR methods including the most recent one in [19].

Both the proposed scheme and that of [20] use double-BCD encoding to represent the result of PPR. However, for computation of the final product, it was shown that the proposed multiplier needs a 26-digit adder VS the

32-digit adder of [20]. Moreover, use of Kogge-Stone parallel prefix tree, instead of the 128-bit parallel prefix quaternary tree, showed some additional latency advantage. We provided, in Table VIII, the overall area and time measures for four multiplier designs, based on Logical Effort and same assumptions. The proposed multiplier sacrifices area for high speed, while the slower multipliers in [18], [20], and modified [20] with 14%, 20%, and 13% more latency, consume 15%, 36% and 33% less area, respectively. The latter figures are not obviously exact due to the fact that Logical Effort analysis does not take into account the wiring effects. However, the differences seem to be large enough to make our comparisons reliable. Nevertheless, we synthesized the proposed multiplier using the Synopsis Design Compiler based on the 0.13 μm standard CMOS technology. The result shows 16% speed advantage versus the scaled result for [18].

Further research may be envisaged on using the proposed multiplier in the design of multiplicative dividers, and evaluation of decimal functions (e.g., radix-10 exponentiation).

APPENDIX: PPR METHODS

We compare the performance of the PPR step of decimal multiplier of [20] with that of [19] and Dadda's scheme [10] for multi-operand decimal addition. The outcome of comparison might depend on the number of BCD operands to be reduced to two. Since the 16 \times 16 BCD multiplier of [20] generates 32 decimal partial products at the outset, the number of operands for all the three compared methods shall be 32. Dadda's scheme uses the reduction procedure of [31] to reduce a column of 32 equally weighted BCD digits to a 9-bit binary number. This is done via eight levels of CSA (16 logic levels) that produces a 4-bit nonredundant number at the least significant part and a 4-bit carry-save number at the most significant part. Therefore, a 4-bit adder (at least five logic levels) is required to produce the final 9-bit result. Then this binary sum, which is at most equal to $32 \times 9 = 288$, is converted to three BCD digits via a recursive binary-to-decimal converter [32]. This conversion adds another 12 logic levels to the critical delay path of the reduction process. This leads to a triple-BCD sum which should be reduced to a double-BCD result. Using the same Dadda reduction scheme, as suggested in [10], requires another 11 logic levels (one full adder, a 3-bit adder, and a 5-bit binary-to-decimal converter). Therefore, the overall delay amounts to 44 logic levels.

Fig. 11 shows a dot notation illustration of the reduction block used in [20], which may be called as (3:2) decimal 4-2-2-1 compressors. This is composed of four parallel full adders (i.e., the dashed ovals) and a correction logic called $\times 2$ in [20]. The overall delay of such reduction block amounts to 5 logic levels (i.e., 2 levels for CSA and 3 levels for $\times 2$ from [20]). Eight levels of these blocks are required for reduction of 32 decimal operands to a double-4-2-2-1 sum. The total delay amounts to 40 logic levels versus the 44 logic levels in Dadda's scheme.

The other figure in the second row of Table V, for 68 operands, corresponds to ten reduction levels.

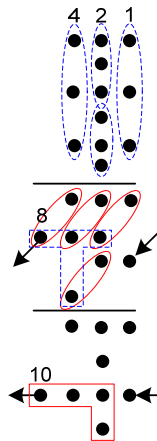


Fig. 11: Dot-notation representation of a reduction block in [20].

In [20], the use of 4-2-2-1 (4:2) compressors is proposed for combined binary/decimal multiplier, where two 4-2-2-1 CSAs may be replaced by such compressors. This idea is picked up in [19], where a PPR scheme is developed with reduction cells as in Fig. 12. A naive comparison of the latency of this reduction cell, with that of two reduction cells of [20], would conclude equal latency PPRs. However, [19] reports 7% less latency for 16-operands. The reason is that in the PPR tree of [20] the overall critical path solely goes through the carry paths of the reduction cells (i.e., 4-2-2-1 CSA cells) with 5 logic level delay per carry path. However, in the PPR tree of [19] some 4-2-2-1 CSA cells contribute in the overall critical path via their sum path with only 2 logic level delay.

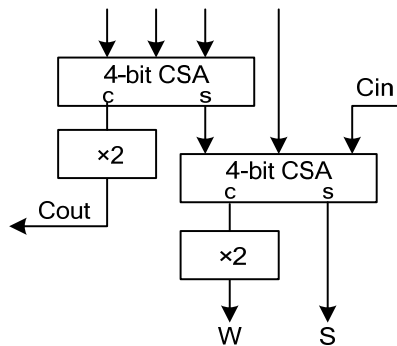


Fig. 12: Decimal (4:2) compressor. All signals are BCD 4-2-2-1 (Adapted from [19]).

ACKNOWLEDGMENTS

We wish to sincerely thank the anonymous reviewers of different versions of this work since submission of the original manuscript in Dec. 2006. This research has been funded in part by the IPM School of Computer Science under grant #CS1387-3-01 and in part by Shahid Beheshti University under grant #D/600/212.

REFERENCES

- [1] Cowlshaw, M. F., "Decimal Floating-Point: Algorithm for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 104-111, June 2003.
- [2] Busaba, F. Y., C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 Decimal Arithmetic Unit," *Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1335-1339, November 2001.
- [3] Shankland, S., "IBM's POWER6 gets help with math, multimedia," *ZDNet News*, October 2006.
- [4] Webb, C. F., "IBM z10: The Next-Generation Mainframe Microprocessor," *IEEE Micro*, Vol. 28, No. 2, pp. 19-29, March 2008.
- [5] IEEE Standards Committee, "754-2008 IEEE Standard for Floating-Point Arithmetic," pp. 1-58, August 2008. DOI: 10.1109/IEEESTD.2008.4610935 (<http://ieeexplore.ieee.org/servlet/opac?punumber=4610933>).
- [6] Schmookler, M., and A. Weinberger, "High Speed Decimal Addition," *IEEE Transactions on Computers*, Vol. C-20, No. 8, pp. 862-866, August 1971.
- [7] Thompson, J., K. Nandini, and M. J. Schulte, "A 64-bit Decimal Floating-Point Adder," *Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI'04)*, pp. 197-198, February 2004.
- [8] Vazquez, A., and E. Antelo, "Conditional Speculative Decimal Addition," *Proceedings of the 7th Conference on Real Numbers and Computers (RNC 7)*, pp. 47-57, July 2006.
- [9] Kenney, R. D., and M. J. Schulte, "High-Speed Multioperand Decimal Adders," *IEEE Transactions on Computers*, Vol. 54, No. 8, pp. 953-963, August 2005.
- [10] Dadda, L., "Multi Operand Parallel Decimal Adder: a mixed Binary and BCD approach," *IEEE Transactions on Computers*, Vol. 56, No. 10, pp. 1320-1328, October 2007.
- [11] Erle, M. A., and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition," *Conference on Application-Specific Systems, Architectures, and Processors*, pp. 348-358, June 2003.
- [12] Kenney, R. D., M. J. Schulte, and M. A. Erle, "A high-frequency decimal multiplier," *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pp. 26-29, October 2004.
- [13] Erle, M. A., E. M. Schwartz, and M. J. Schulte, "Decimal Multiplication with Efficient Partial Product Generation," *17th IEEE Symposium on Computer Arithmetic*, pp. 21-28, June 2005.
- [14] Liang-Kai, W., and M. J. Schulte, "Decimal floating-point division using Newton-Raphson iteration," *15th Int. Conf. Application-Specific Systems, Architectures and Processors*, pp. 84-95, 2004.
- [15] Nikmehr, H., B. Phillips, and C. C. Lim, "Fast Decimal Floating-Point Division," *IEEE Transactions on VLSI Systems*, Vol. 14, No. 9, pp. 951-961, September 2006.
- [16] Lang, T., and A. Nannarelli, "A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture," *IEEE Transactions on Computers*, Vol. 56, No. 6, pp. 727-739, June 2007.
- [17] Wang, L. and M. J. Schulte, "A Decimal Floating-Point Divider Using Newton-Raphson Iteration," *Journal of VLSI Signal Processing Systems*, Vol. 14, No. 1, pp. 3-18, October 2007.
- [18] Lang, T., and A. Nannarelli, "A radix-10 Combinational Multiplier," *Asilomar Conference on Signals, Systems, and Computers*, pp. 313-317, November 2006.
- [19] Castellanos, I. D., and J. E. Stine, "Compressor Trees for Decimal Partial Product Reduction," *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, pp. 107-110, May 2008.

- [20] Vazquez, A., E. Antelo, and P. Montuschi, "A New Family of High-Performance Parallel Decimal Multipliers," *Proc. of the 18th IEEE Symposium on Computer Arithmetic*, pp. 195-204, June 2007.
- [21] Sutherland, I. E., R. F. Sproull and D. Harris, "Logical Effort: Designing Fast CMOS Circuits," Morgan Kaufmann, 1999, ISBN: 1558605576.
- [22] Jaberipur, G., and A. Kaivani, "Binary-Coded Decimal Digit Multipliers," *IET Computers & Digital Techniques*, Vol. 1, No. 4, pp. 377-381, July 2007.
- [23] Richards, R. K., "Arithmetic Operations in Digital Computers," Van Nostrand, New York, 1955.
- [24] Larson, R. H., "High Speed Multiply Using Four Input Carry Save Adder," *IBM Technical Disclosure Bulletin*, pp. 2053-2054, December 1973.
- [25] Ueda, T., "Decimal Multiplying Assembly and Multiply Module," U.S. Patent #5379245, January 1995.
- [26] Wallace, C. S., "A Suggestion for Fast Multiplier," *IEEE Transactions on Electronic Computers*, Vol. 13, pp. 14-17, 1964.
- [27] Mathew, S. K, M. Anders, R. K. Krishnamurthy and S. Borkar. "A 4Ghz 130nm address generation unit with 32-bit sparse-tree adder core," *IEEE Journal of Solid-State Circuits*, Vol. 38, No. 5, pp. 689-695, May 2003.
- [28] Kogge, P. M., H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp.786-793, August 1973.
- [29] Hickmann, B., A. Krioukov, M. Schulte, and M. Erle, "A Parallel IEEE P754 Decimal Floating-Point Multiplier," *25th International Conference on Computer Design (ICCD 2007)*, pp. 296-303, Oct. 2007.
- [30] Grecu, C., P. P. Pande, A. Ivanov and R. Saleh, "Timing analysis of network on chip architectures for MP-SoC platforms," *Microelectronics Journal*, Vol. 36, No. 9, pp. 833-845, September 2005.
- [31] Dadda L., "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 19, pp. 349-356, Mar. 1965.
- [32] Nicoud, J.D, "Iterative Arrays for Radix Conversion," *IEEE Transactions on Computers*, vol. C-20, No. 12, pp.1479-1489, November 1971.



Ghassem Jaberipur, associate professor of Computer Engineering in the department of Electrical and Computer Engineering of Shahid Beheshti University (Tehran, Iran), received his B.S in Electrical Engineering and PhD in Computer Engineering from Sharif University of Technology in 1974 and 2004, respectively, M.S in Engineering from UCLA in 1976, and M.S in Computer Science from University of Wisconsin in Madison in 1979. Currently, his main research interest is in Computer Arithmetic.



Amir Kaivani received his B.S. in Computer Engineering from Islamic Azad University of Mashhad in 2005, and M.S. in Computer Engineering from Shahid Beheshti University (SBU) in 2007. He is currently PhD student in Electrical Engineering at SBU. His research interests include computer arithmetic, quantum computing and reversible circuit design