Volume 40, Number 10, October 2009
ISSN 0026-2692
ELSEVIER
MICROELECTRONICS JOURNAL
http://www.elsevier.com/locate/mejo
EDITOR-IN-CHIEF
B. Courtois
Available online at www.sciencedirect.com
ScienceDirect

Microelectronics Journal 40 (2009) 1471–1481

Contents lists available at ScienceDirect

Microelectronics Journal

journal homepage: www.elsevier.com/locate/mejo

A fully redundant decimal adder and its application in parallel decimal multipliers

Saeid Gorgin [a], Ghassem Jaberipur [a,b,*]

[a] Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran
[b] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

A R T I C L E I N F O

Article history:
Received 7 June 2008
Received in revised form
15 July 2009
Accepted 21 July 2009
Available online 22 August 2009

Keywords:
Decimal digit sets
Carry-free decimal addition
Redundant number systems
Decimal parallel multiplication
Weighted bit-set encoding

A B S T R A C T

Decimal hardware arithmetic units have recently regained popularity, as there is now a high demand for high performance decimal arithmetic. We propose a novel method for carry-free addition of decimal numbers, where each equally weighted decimal digit pair of the two operands is partitioned into two weighted bit-sets. The arithmetic values of these bit-sets are evaluated, in parallel, for fast computation of the transfer digit and interim sum. In the proposed fully redundant adder (VS semi-redundant ones such as decimal carry-save adders) both operands and sum are redundant decimal numbers with overloaded decimal digit set [0, 15]. This adder is shown to improve upon the latest high performance similar works and outperform all the previous alike adders. However, there is a drawback that the adder logic cannot be efficiently adapted for subtraction. Nevertheless, this adder and its restricted-input varieties are shown to efficiently fit in the design of a parallel decimal multiplier. The two-to-one partial product reduction ratio that is attained via the proposed adder has lead to a VLSI-friendly recursive partial product reduction tree. Two alternative architectures for decimal multipliers are presented; one is slower, but area-improved, and the other one consumes more area, but is delay-improved. However, both are faster in comparison with previously reported parallel decimal multipliers. The area and latency comparisons are based on logical effort analysis under the same assumptions for all the evaluated adders and multipliers. Moreover, performance correctness of all the adders is checked via running exhaustive tests on the corresponding VHDL codes. For more reliable evaluation, we report the result of synthesizing these adders by Synopsys Design Compiler using TSMC 0.13 μm standard CMOS process under various time constrains.

1. Introduction

There is a growing importance attached to hardware units capable of decimal computer arithmetic. This is mainly due to increasing computer applications in commerce, banking, and internet-based business [6]. Specifications for decimal number representation and exact definition of the related arithmetic operations are now part of the IEEE 754-2008 standard for floating point arithmetic [13]. The IBM z900 system [3] and lately the Power6 processor [8] are already equipped with decimal arithmetic hardware units, where multiplication is hitherto performed iteratively. However, the state of the art research on fully parallel decimal multipliers (e.g., [20, 30]) can encourage the designers of the future commercial processors to incorporate these fast multipliers in decimal hardware units.

The bulk of multiplication process is partial product reduction (PPR). This has been performed via decimal carry-save adders in iterative decimal multipliers [9] and in parallel ones [20]. Alternatively, iterative [17] and parallel [30] partial product reduction, and multi-operand decimal addition [18,7,4] use binary carry-save adders and special correction logic.

Most of the afore-cited contributions use semi-redundant adders, where only one of the operands is redundant. For example, the decimal carry-save adder receives one nonredundant BCD and one redundant carry-save numbers and produces a carry-save sum. However, fully redundant decimal adders [29,26,22,16,12] are also useful, where both operands as well as the sum are redundant numbers. For instance, a BCD (4; 2) compressor, the Svoboda adder [29], and a fully redundant three-operand adder have been used within the design of iterative decimal multipliers in [9,10, 17], respectively. Also 4-2-2-1 (4; 2) compressors have been used in a partial product reduction tree [4].

Binary fully redundant adders (e.g., signed digit adders or (4; 2) compressors) that are used for PPR in parallel binary multipliers [2,21] are known for two-to-one reduction ratio and VLSI-friendly recursive reduction tree. Likewise, fully redundant

* Corresponding author at: Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran.
E-mail addresses: Gorgin@sbu.ac.ir (S. Gorgin).
Jaberipur@sbu.ac.ir (G. Jaberipur).

0026-2692/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mejo.2009.07.002

decimal adders can potentially show the same properties for parallel decimal multipliers. However, there are drawbacks in using the previous such adders (i.e., [29,26,22,16,12]) within the design of parallel decimal multipliers. The adders in [29,26,22] are rather slow, and the signed digit representation shared by all these adders complicates the forward and backward conversion of conventional sign magnitude BCD representations of inputs and outputs. Therefore, following the state of the art research on parallel decimal multipliers (i.e., [20, 30] with semi-redundant decimal adders and [4] with fully redundant PPR cell), we are motivated to design a very fast fully redundant decimal addition scheme, based on an unsigned redundant decimal digit set and show its efficient application in parallel decimal multiplication.

The rest of this paper is organized as follows. We study the diversity of decimal digit sets and the issue of fully redundant decimal addition in Section 2. A theoretical foundation, in Section 3, paves the way for the discussion on the proposed fully redundant decimal addition scheme in Section 4. The application of the proposed adder in parallel decimal multipliers is taken up in Section 5. The logical effort [28] is used, in Section 6, to compare the performance of the proposed adder and all other ones studied in this paper. The latter evaluations are supported with 0.13 μm TSMC standard CMOS technology synthesis. Moreover, a 16-digit instance of the proposed parallel decimal multiplier is compared with that of the latest relevant work based on logical effort analysis. Finally, conclusions and prospects for further research are found in Section 7. The following definitions and abbreviations are used throughout the paper.

Faithful encoding of a digit set: all the bit combinations of the encoding represent valid digits of the digit set (e.g., all the 16 bit combinations of the 4-2-2-1 encoding represent values in [0, 9]).

Redundant encoding of a digit set: at least one member of the digit set has more than one bit representations (e.g., both 1000 and 0110 represent 4 in the 4-2-2-1 encoding of decimal digit set [0, 9]).

Redundant digit set: a digit set whose cardinality is more than the radix (e.g., decimal digit set [0, 15]).

Redundant number: a number whose digits belong to a redundant digit set.

Redundant adder: an adder that produces redundant sum, where zero or more of its operands may be redundant.

Carry-free addition: implemented by a redundant adder such that no carry generated in one position passes beyond the next higher weighted position.

Posibit: a normal bit with arithmetic values $\{0, 1\}$, shown in dot notation as ● and symbolically with lowercase letters.

Negabit: a negative bit with arithmetic values $\{-1, 0\}$, shown in dot notation as ○ and symbolically with uppercase letters.

BCD: binary coded decimal encoding: $[0, 9] \Rightarrow [0000, 1001]$.

DSD: decimal signed digit (e.g., $[-\alpha, \alpha]$ for $\alpha = 5, 6, 7$, or 9)

ODDS: overloaded decimal digit set [0, 15]. This abbreviation is also used for redundant decimal number systems and adders with such digit sets.

PPG: partial product generation.

PPR: partial product reduction.

WBS: weighted bit-set encoding [14] of a digit set is a collection of zero or more posibits and negabits scattered in one or more radix-2 positions.

2. Decimal digit sets and decimal carry-free addition

The dominant digit set for radix-10 positional number systems is naturally the human-friendly digit set [0, 9]. However, several unconventional decimal digit sets have been used in computer arithmetic. For example, Excess-3 [25] digit set [3,12] as a nonredundant encoding of [0, 9], and several redundant decimal digit sets demonstrate the diversity of approaches to decimal computer arithmetic. The latter include decimal carry-save digit set [0, 10] implied in [9, 20], overloaded decimal digit set [0, 15] of [17, 18], DSD sets $[-\alpha, \alpha]$ in [29], [26, 12], and [22], for $\alpha = 6, 7$ and 9, respectively, the double-decimal digit set [0, 18] implied in [30] and [4], and $[-8, 9]$ in [16].

Each decimal digit set can be represented by several binary encodings. Nonetheless, Table 1 depicts one practically used binary encoding for the aforementioned digit sets, where ● indicates a posibit, ○ stands for a negabit, and a decimal digit set $[\alpha, \beta]$ is characterized by its cardinality $\xi = \beta - \alpha + 1$, redundancy index $\rho = \xi - 10$ [23], and γ as the number of invalid bit combinations. For example, BCD is a nonredundant ($\rho = 0$) digit set with nonredundant and unfaithful ($\gamma = 6$) encoding, 4-2-2-1 is nonredundant ($\rho = 0$) with redundant and faithful ($\gamma = 0$) encoding and ODDS is redundant ($\rho = 6$) with nonredundant and faithful encoding.

The dominant encoding for decimal data in digital storage and devices is BCD. Therefore, all the decimal arithmetic hardware units, by and large, assume BCD I/O. However, other encodings are also used for representing the intermediate results such as accumulated partial product. The ODDS (the 8th entry in Table 1), is of particular interest in this study. The four bits of this faithful encoding are maximally utilized such that each of the sixteen possible bit combinations represents a distinct valid digit ($\gamma = 0$). This is, intuitively, expected to lead to area and energy savings. Moreover, the unsigned nature of this digit set goes with the process of unsigned multi-operand addition naturally required as the bulk of decimal multiplication.

The ODDS was used within an iterative decimal multiplier [17], where partial products are double-BCD numbers. Possible +6 corrections, due to regarding a radix-16 carry as a decimal one, convert the partial product to two ODDS numbers. The latter are added to the accumulated partial product, also an ODDS number, via a fully redundant three-operand ODDS adder. However, in a parallel decimal multiplier, all the partial products are produced at once. Each partial product is commonly generated as a double-decimal number that can be converted to an ODDS number. This would be a carry-free operation that is attainable in parallel for all double-decimal partial products. The process of ODDS partial product reduction can go on by fully redundant carry-free addition of pairs of ODDS partial products. Similar two-operand fully redundant carry-free addition schemes have been offered in [29,26,22,12,16] for other redundant decimal digit sets. However, we have come across three decimal multiplier designs that use

Table 1
Diversity of decimal digit sets.

Name and/or source		Digit set	Encoding	α	β	ξ	ρ	γ
1	BCD	[0, 9]	●●●●	0	9	10	0	6
2	Excess-3	[3,12]	●●●●	3	12	10	0	6
3	4-2-2-1	[0, 9]	●●● ●	0	9	10	0	0
4	[10]	[−5, 5]	●●●●●	−5	5	11	1	20
5	[29]	[−6, 6]	●●●●●	−6	6	13	3	18
6	[26, 12]	[−7, 7]	○●●●	−7	7	15	5	1
7	DSD [22]	[−9, 9]	●●●● ○○○○	−9	9	19	9	36
8	ODDS [17]	[0, 15]	●●●●	0	15	16	6	0
9	[9, 20]	[0, 10]	●●●● ●	0	10	11	1	6
10	[30, 4]	[0,18]	●●● ● ●●● ●	0	18	19	9	0

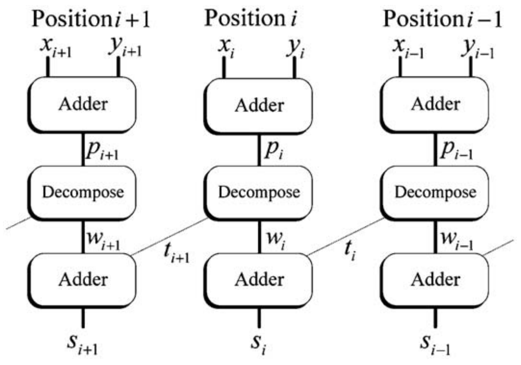


Fig. 1. An illustration of carry-free addition algorithm.

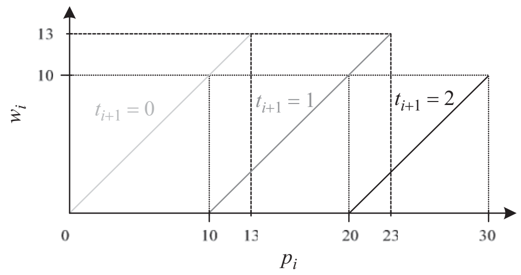


Fig. 2. An illustration of the five ranges for p_i in Algorithm 1.

two-operand fully redundant carry-free adders. The iterative multipliers in [9, 10] use decimal (4; 2) compressors and Svoboda adder [29], respectively and [4] reduces the partial products via 4-2-2-1 (4; 2) compressors, where the latter is composed of two 4-2-2-1 full adders of [30].

Algorithm 1 is an instance of the general carry-free addition algorithm (Fig. 1) that is adapted for ODDS numbers. It can also work for other redundant digit sets via assignment of appropriate values to comparison constants and transfers in Step II.

Algorithm 1 (*carry-free addition of ODDS numbers*):

Inputs: ODDS numbers $x=x_{k-1},\dots,x_0$ and $y=y_{k-1},\dots,y_0$ ($x_i, y_i \in [0, 15]$ for $0 \le i \le k-1$).

Output: ODDS number $s=t_k s_{k-1},\dots,s_0$, where $s_i \in [0, 15]$ ($0 \le i \le k-1$) and $t_k \in [0, 2]$.

For $0 \le i \le k-1$, do in parallel:

I. Compute the position-sum digits $p_i = x_i + y_i \in [0, 30]$.

II.
$$t_{i+1} = \begin{cases} 0 & \text{if } 0 \le p_i \le 9 \Rightarrow 0 \le w_i \le 9 \\ 0 \text{ or } 1 & \text{if } 10 \le p_i \le 13 \Rightarrow 0 \le w_i \le 13 \\ 1 & \text{if } 14 \le p_i \le 19 \Rightarrow 4 \le w_i \le 9 \\ 1 \text{ or } 2 & \text{if } 20 \le p_i \le 23 \Rightarrow 0 \le w_i \le 13 \\ 2 & \text{if } 24 \le p_i \le 30 \Rightarrow 4 \le w_i \le 10 \end{cases}.$$

III. Compute the interim sum digit $w_i = p_i - 10t_{i+1}$.

IV. Form the final sum digit $s_i = w_i + t_i$.

The five comparison constants (i.e., 9, 13, 19, 23, and 30), in Step II, raising to five different ranges of position-sum values (illustrated in Fig. 2) are selected with the aim of limiting the number of transfer values to three (i.e., the minimum possible for fully redundant representationally closed carry-free addition [15]).

The straight forward implementation of Algorithm 1 is obviously inefficient, especially for derivation of the transfer values in Step II, where p_i is to be checked against five ranges of values. However, the digit-size addition-like operations of Steps I through IV are merely used for algorithm description and need not be independently realized in hardware. For example, Steps I, II, and III may be fused into a single one, by noting that w_i and t_{i+1} are directly computable in hardware as functions of x_i and y_i, i.e.,

$$w_i = \omega(x_i, y_i); t_{i+1} = \tau(x_i, y_i).$$

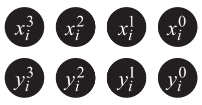
This allows the designer to choose the best possible merged implementation. It may be the case, with certain digit sets and/or encodings, that some form of addition is still part of the best hardware implementation scheme for ω and τ, but this is not required. For instance, at one extreme, we could think of developing an 8 bit combinational logic for computing ω and τ. But, the latency, area, and the power dissipation of the required circuitry seem to pass practical constraints, let alone the difficult design process. However, it may be the case that t_{i+1} and w_i do not necessarily depend on all the bits of x_i and y_i, which would obviously lead to simpler ω and τ functions. This possibility is explored in the next two sections.

3. Decimal digit set conversion via weighted bit-set partitioning

A 2×4 bit-array that contains the corresponding bits of equally weighted digits x_i and y_i is depicted by Fig. 3. This bit-array, as an instance of the weighted bit-set (WBS) encoding [14], represents the position-sum $p_i = x_i^3 x_i^2 x_i^1 x_i^0 + y_i^3 y_i^2 y_i^1 y_i^0$, where no addition operation takes place and WBS encoding of p_i is readily available at no cost.

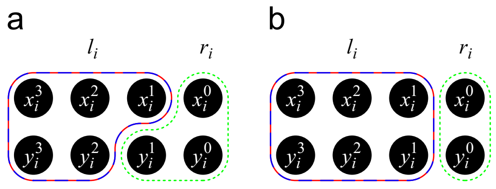
The WBS representation of p_i may be viewed as the initial representation of the sum digit, or the starting point in a process of digit set conversion [19] that ends in representing the sum in a target encoding. However, our digit set conversion approach is to partition the WBS representation of p_i into two weighted bit-sets: l_i, a transfer-generating weighted bit-set, is used for extracting t_{i+1}, and the remaining bits as the residual weighted bit-set r_i, are kept intact as a component of the interim sum w_i. Definition 1 formally describes the WBS partitioning.

Definition 1 (*WBS partitioning of the position-sum*). The eight bits in the WBS encoding of p_i are divided into two nonoverlapping weighted bit-sets designated as the left partition l_i and right partition r_i, such that $||p_i|| = ||l_i|| + ||r_i||$, where $||u||$ denotes the arithmetic value of the weighted bit-set u. If the two WBS partitions l_i and r_i share a binary position (i.e., each partition includes one of the two posibits of the same position), the



x_i^3 x_i^2 x_i^1 x_i^0
y_i^3 y_i^2 y_i^1 y_i^0

Fig. 3. WBS encoding of p_i.



a
b
l_i
r_i
x_i^3 x_i^2 x_i^1 x_i^0
y_i^3 y_i^2 y_i^1 y_i^0

Fig. 4. Odd (a) and even (b) partitioning.

partitioning is called an odd WBS partitioning, otherwise it is an even partitioning.

Fig. 4-a (-b) shows an odd (even) partitioning. The bits of l_i are to be used for extracting t_{i+1} such that the final sum computation (i.e., Step IV of Algorithm 1) leads to no new transfer. The following equations rule the required partitioning of p_i and the final composition of s_i:

$$\|p_i\| = \|l_i\| + \|r_i\| \tag{1}$$

$$\|l_i\| = \|z_i\| + 10t_{i+1} \tag{2}$$

$$\|w_i = \|z_i\| + \|r_i\| = p_i - 10t_{i+1} \tag{3}$$

$$s_i = \|z_i\| + \|r_i\| + t_i \tag{4}$$

For an efficient computation of the transfer t_{i+1} and the constituent bits of z_i, it is desirable to keep the number of bits of l_i (r_i) and accordingly the value of $||l_i||(||r_i||)$ to the minimum (maximum) possible.

Lemma 1 (*Maximum* $||r_i||$). *The maximum possible value for the weighted bit-set* r_i *is* $||r_i^o||_{\max} = 3\times 2^{j-1}-2(||r_i^e||_{\max} = 4\times 2^{j-1}-2)$ *for odd (even) partitioning of* p_i*, where* j *is the number of binary positions covered by* r_i.

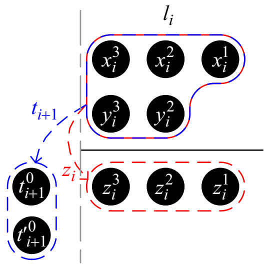
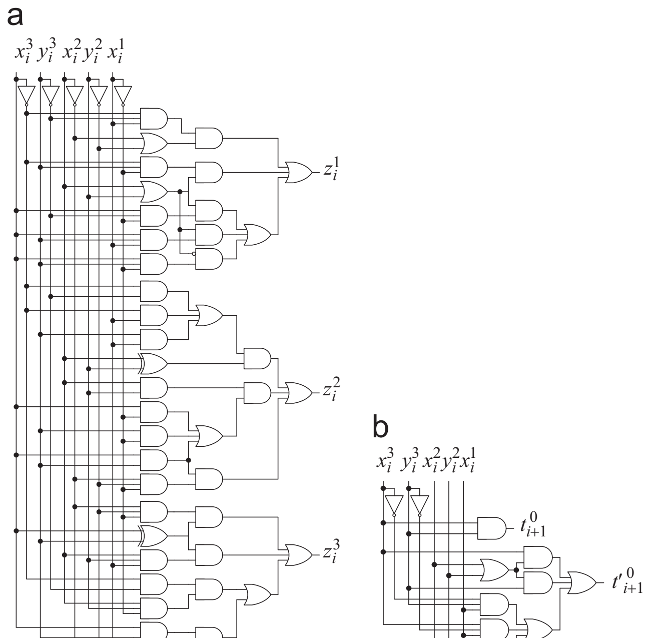


Fig. 5. Decomposition of l_i.



a
$x_i^3\ y_i^3\ x_i^2\ y_i^2\ x_i^1$
z_i^1
z_i^2
z_i^3
b
$x_i^3\ y_i^3\ x_i^2\ y_i^2\ x_i^1$
t_{i+1}^0
t'^0_{i+1}

Fig. 6. The logic for $z_i = z_i^3 z_i^2 z_i^1$ (a), and t_{i+1} (b).

Proof. Trivial by examining Fig. 4. □

The next Lemma is expressed, in general terms, for digit set [α, β] (e.g., $\alpha = 0$ and $\beta = 15$ in Algorithm 1) and the minimum and maximum transfer values $t_{\min}$ and $t_{\max}$ (e.g., $t_{\min} = 0$ and $t_{\max} = 2$ in Algorithm 1).

Lemma 2. *The interim sum* w_i *is bounded as* $\alpha - t_{\min} \le ||z_i|| + ||r_i|| \le \beta - t_{\max}$.

Proof. No new transfer is generated in the final sum computation (Eq. (4) if and only if $\alpha - t_{\min} \le w_i = s_i - t_i = ||z_i|| + ||r_i|| \le \beta - t_{\max}$ (e.g., $0 = 0-0 \le ||z_i|| + ||r_i|| \le 15-2 = 13$ in Algorithm 1). □

Theorem 1 (*Maximum* j). *The maximum number of binary positions covered by the WBS partition* r_i *is* $j^o_{\max} = 2$ *and* $j^e_{\max} = 1$ *for odd and even partitioning, respectively.*

Proof. The maximum value of $||r_i||$, as determined in Lemma 1, and that of $||z_i||$ (i.e., $||z_i||_{\max} = 9$ from Eq. (2)) should satisfy the constraint of Lemma 2. However, excluding the trivial case of $j = 0$ leads to even $||z_i||$ and thus $||z_i||_{\max} = 8$. Applying these maximum values in the inequality of Lemma 2 leads to the following inequalities, whose solution for j leads to $j \le 2$ ($j \le 1$), for odd (even) partitioning. $||z_i||_{\max} + ||r_i||_{\max} \le 13$

$$\Rightarrow \begin{cases} 8 + 3\times 2^{j-1} - 2 \le 13 & \text{for odd partitioning} \\ 8 + 4\times 2^{j-1} - 2 \le 13 & \text{for even partitioning} \end{cases} \qquad \square$$

Corollary 1 (*Bits in* r_i *and* l_i). *The number of bits* N_r^o, N_r^e, N_l^o, *and* N_l^e *in the right-odd, right-even, left-odd, and left-even partitions are* 3, 2, 5, *and* 6, *respectively.* □

4. Carry-free addition of overloaded decimal numbers

Recalling Corollary 1, the case of odd partitioning with $N_l^o = 5$ and $N_r^o = 3$ is obviously preferred to the case of even partitioning. WBS representation of p_i was depicted in Fig. 3 and its partitioning into five bits of l_i and three bits of r_i was shown in Fig. 4-a. However, for further clarification on the necessity for inclusion of at least five bits in l_i, observe that the next smaller collection (i.e., x_i^3, y_i^3, x_i^2 and y_i^2) may lead to $||l_i|| = 8$. For example, the case of $x_i = 11$ and $y_i = 3$, leads to $||z_i|| = 8$ and $||r_i|| = 6$. The latter two equalities violate the restriction of Lemma 2 when applied for ODDS (i.e., $w_i = ||z_i|| + ||r_i|| \le 13$). Decomposition of l_i to t_{i+1} and z_i, given that $||l_i||$ is even, does not allow for odd $||z_i||$ values (i.e., $||z_i|| \in \{0, 2, 4, 6, 8\}$). Recalling that $0 \le t_{i+1} \le 2$, let $z_i = z_i^3 z_i^2 z_i^1 0$ and $t_{i+1} = t'^0_{i+1} + t^0_{i+1}$. Then Eq. (2) can be illustrated as in Fig. 5. A 5-input/5-output truth table that is easily derivable from Fig. 5, leads to the required combinational logic as depicted in Fig. 6. The latencies of the five outputs are as follows, where the latency of a simple gate (i.e., AND, NAND, OR, and NOR) with at most three inputs is assumed to be ΔG, and that of an XOR gate is $2\Delta G$.

$$\Delta t'^0_{i+1} = 3\Delta G, \Delta t^0_{i+1} = \Delta G, \Delta z_i^3 = 4\Delta G, \Delta z_i^2 = 4\Delta G, \Delta z_i^1 = 4\Delta G$$

Fig. 7 shows the steps involved in the proposed addition scheme, where in the first stage, a half adder produces $v_i^1 = x_i^0 y_i^0$ and $v_i^0 = x_i^0 \oplus y_i^0$. In the second stage a full adder in position 0 and a half adder in position 1 are used. These operations, via two half adders and one full adder, take place in parallel with those of Fig. 6.

Since the bits of z_i are all available in $4\Delta G$ (see Fig. 6-a), it is desirable to deliver the three bits c_i^1, c_i^2 and q_i^1 at most with the same latency of $4\Delta G$. The latter two bits, as the output of two cascaded half adders meet the latter desired latency constraint.

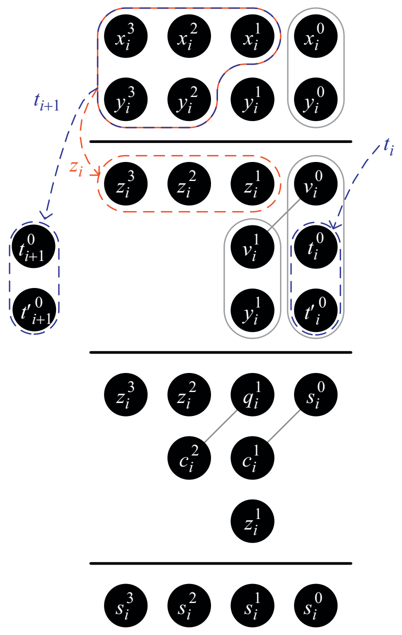


Fig. 7. A digit-slice of the proposed addition scheme.

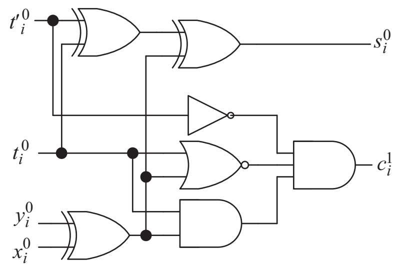


Fig. 8. Custom designed logic to compute c_i^1 in $4\Delta G$.

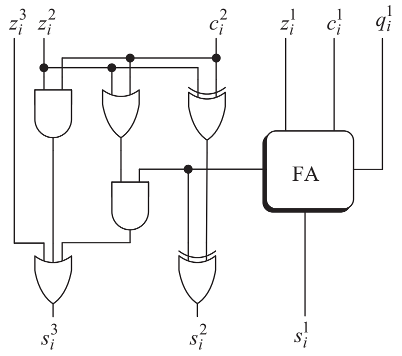


Fig. 9. Custom designed 3-bit adder.

However, to compute c_i^1 with the same latency of $4\Delta G$, we use Eq. (5) implemented by the logic of Fig. 8.

$$c_i^1 = (x_i^0 \oplus y_i^0 \oplus t_i^0)t'^0_i = \overline{\overline{x_i^0 \oplus y_i^0 t_i^0} + (x_i^0 \oplus y_i^0)t_i^0 + \overline{t'^0_i}} \tag{5}$$

The last stage, in Fig. 7, can use a 3-bit adder to derive the final sum. However, given that no new transfer is to be generated at this stage, the 3-bit adder shall not generate a carry-out. This will reduce the logic of the most significant bit to a single OR gate. Fig. 9 depicts the required custom designed 3-bit adder with $4\Delta G$ latency. Therefore, the overall latency of the proposed ODDS adder amounts to $8\Delta G$. Nevertheless, more reliable analysis based on logical effort is provided in Section 6.

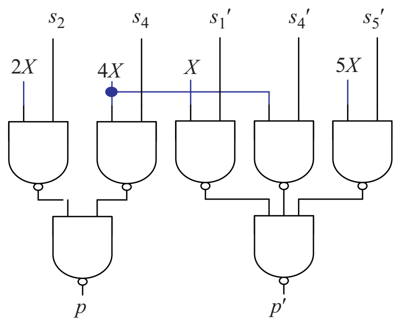


Fig. 10. ith digit-slice of the selector of multiples.

Table 2
Selection of the two components of multiples.

y_i	0	1	2	3	4	5	6	7	8	9
s, s'	0, 0	0, s'_1	s_2, 0	s_2, s'_1	0, s'_4	0, s'_5	s_2, s'_4	s_2, s'_5	s_4, s'_4	s_4, s'_5
P	0	0	$2X$	$2X$	0	0	$2X$	$2X$	$4X$	$4X$
P'	0	X	0	X	$4X$	$5X$	$4X$	$5X$	$4X$	$5X$

5. A parallel BCD multiplier with ODDS accumulated partial product

An $n \times n$ parallel decimal multiplication is typically a three stage process, where each partial product is a double-decimal number. A collection of n such numbers, produced at once, constitute an array of $2n$ nonredundant BCD numbers. The process of PPR, as the second stage, reduces these $2n$ BCD numbers to one redundant decimal number (e.g., decimal carry-save [20] or double-decimal [30]). The final stage of multiplication converts the redundant product to an equivalent BCD result.

In the following subsections, we use an unsigned partial product generation scheme based on the work of [9], explain the details of PPR via ODDS adders and compressors, and the final ODDS-to-BCD converter.

5.1. Partial product generation

We take advantage of the easy-multiple generator of [9] to produce unsigned multiples $2X$, $4X$, and $5X$ of the multiplicand X and we use a fast custom designed selector, whose ith digit-slice is depicted in Fig. 10. The selector signals s'_1, s_2, s_4, s'_4 and s'_5 are derived from the bits y_i^3, y_i^2, y_i^1 and y_i^0 of the ith multiplier digit y_i (equation-set (6)) corresponding to X, $2X$, $4X$, $4X$, and $5X$, respectively. Table 2 shows the ten possible multiples, the values of two components p and p' (equation-set (7)) and the corresponding selector signals s and s'.

$$s'_1 = \overline{y_i^3 y_i^2} y_i^0, s_2 = y_i^1, s_4 = y_i^3, s'_4 = y_i^3 \overline{y_i^0} + y_i^2 \overline{y_i^0},$$

$$s'_5 = y_i^3 y_i^0 + y_i^2 y_i^0 \tag{6}$$

$$p = s_2(2X) + s_4(4X), p' = s'_1(X) + s'_4(4X) + s'_5(5X) \tag{7}$$

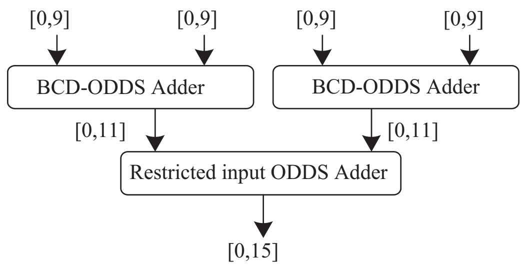


Fig. 11. (4; 1) BCD-to-ODDS compressor.

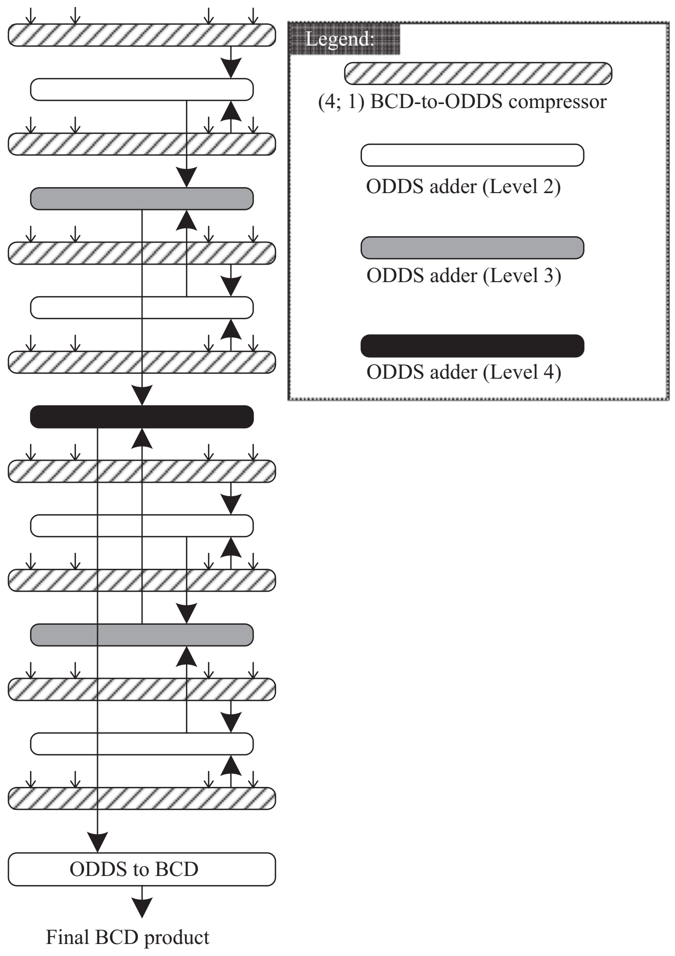


Fig. 12. (32; 1) BCD-to-ODDS compressor and ODDS-to-BCD converter.

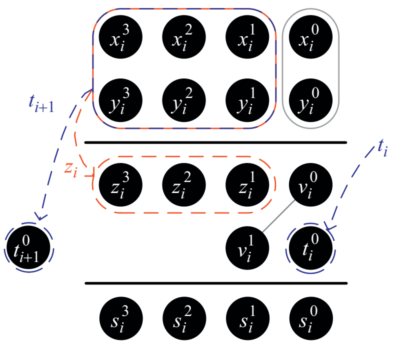


Fig. 13. Simplified ODDS addition for BCD inputs and [0, 11] output.

5.2. *Partial product reduction*

We propose two PPR architectures designated as delay-improved and area-improved. The former uses three varieties of ODDS adders and consumes more area. However, the latter heavily uses conventional binary (4; 2) compressors and leads to less area.

5.2.1. *ODDS delay-improved reduction method*

The fully redundant carry-free ODDS adder of Section 4 may be used to add two BCD numbers and produce an ODDS sum. However, in the first level of reduction, we can use special faster BCD-ODDS adders with BCD inputs and restricted digit set (i.e., [0, 11]) redundant sums. In the second level, these results are fed into special restricted-input ODDS adders with normal ODDS sums. These two levels together, as in Fig. 11, act like a (4; 1) BCD-to-ODDS compressor. This configuration (details are explained in Section 5.2.1.1, below) with latency of $13\Delta G$ performs faster than the alternative one that uses only ODDS adders. At this stage of PPR, the original array of $2n$ BCD numbers is reduced to an array of $\lceil n/2 \rceil$ ODDS numbers. This array may be reduced to one ODDS number via $\lceil \log_2 \lceil n/2 \rceil \rceil$ levels of ODDS adders. Fig. 12 shows the required recursive VLSI-friendly tree, for $n = 16$, and the final ODDS-to-BCD converter.

5.2.1.1. (4; 1) BCD-to-ODDS compressor. The building blocks of (4; 1) BCD-to-ODDS compressor, as was depicted in Fig. 11, consists of two kinds of restricted-input ODDS adder. The semi-redundant BCD-ODDS adder is a simplified case of ODDS adder, where position-sum digits $p_i \in [0, 18]$. This allows for a two valued transfer $t_{i+1} \in [0, 1]$. Furthermore the output digit set can be restricted to [0, 11]. Applying these bounds (i.e., $t_{\min} = 0$, $t_{\max} = 1$, $\alpha = 0$, and $\beta = 11$) in Lemma 2 leads to $0 \le ||z_i|| + ||r_i|| \le 11 - 1 = 10$. The latter, when applied in Theorem 1, leads to $j = 1$. Therefore, six bits contribute in the computation of the bits of z_i as shown in the following equations. Nevertheless, these equations are simpler than those of ODDS adder described in Section 4. The reason lies in several do not care cases due to the restricted-input digit set [0, 9]. Fig. 13 shows the partitioning and the details of addition scheme. The latency of this adder is about seven logic levels VS eight logic levels of the ODDS adder.

$$t_{i+1}^0 = x_i^1 y_i^3 + x_i^2 y_i^3 + x_i^3 y_i^2 + x_i^3 y_i^1 + x_i^3 y_i^3 + x_i^2 y_i^2 y_i^1 + x_i^2 x_i^1 y_i^2,$$
$$z_i^3 = \overline{x_i^3 x_i^2 x_i^1} y_i^3 + x_i^3 \overline{y_i^3 y_i^2 y_i^1} + x_i^2 \overline{x_i^1} y_i^2 \overline{y_i^1} + \overline{x_i^2} x_i^1 y_i^2 y_i^1 + x_i^2 x_i^1 \overline{y_i^2} y_i^1,$$

$$z_i^2 = x_i^3 y_i^3 + \overline{x_i^3 x_i^2 x_i^1} y_i^2 + \overline{x_i^3 x_i^2} y_i^2 \overline{y_i^1} + x_i^2 \overline{x_i^1 y_i^3 y_i^2} + \overline{x_i^2 x_i^1} y_i^2 y_i^1 + \overline{x_i^2} x_i^1 \overline{y_i^2} y_i^1 + x_i^2 x_i^1 \overline{y_i^2 y_i^1},$$

$$z_i^1 = x_i^3 y_i^3 + x_i^2 \overline{x_i^1} y_i^3 + x_i^3 y_i^2 \overline{y_i^1} + \overline{x_i^3 x_i^2 x_i^1} y_i^1 + \overline{x_i^3 x_i^1 y_i^2} y_i^1 + \overline{x_i^2} x_i^1 \overline{y_i^3 y_i^1} + x_i^1 \overline{y_i^3 y_i^2 y_i^1} + x_i^2 x_i^1 y_i^2 y_i^1,$$
$$v_i^0 = x_i^0 \oplus y_i^0, v_i^1 = x_i^0 y_i^0, s_i^0 = v_i^0 \overline{t_i^0} + \overline{v_i^0} t_i^0,$$
$$s_i^1 = \overline{z_i^1} v_i^1 + z_i^1 \overline{v_i^1 v_i^0} + z_i^1 \overline{v_i^1 t_i^0} + \overline{z_i^1} v_i^0 t_i^0,$$

$$s_i^2 = z_i^2 \overline{z_i^1} + z_i^2 \overline{v_i^1 v_i^0} + z_i^2 \overline{v_i^1 t_i^0} + \overline{z_i^2} z_i^1 v_i^1 + \overline{z_i^2} z_i^1 v_i^0 t_i^0,$$
$$s_i^3 = z_i^3 + z_i^2 z_i^1 v_i^1 + z_i^2 z_i^1 v_i^0 t_i^0$$

An ODDS adder with restricted inputs with digit set [0, 11] and output digit set [0, 15] can also be designed with two valued transfers in [0, 1], such that $p_i = 20$, 21 and 22 are decomposed to (1, 10), (1, 11) and (1, 12), respectively. The latter bounds (i.e., $t_{\min} = 0$, $t_{\max} = 1$, $\alpha = 0$, and $\beta = 15$) lead to $0 \le ||z_i|| + ||r_i|| \le 15 - 1 = 14$, and $j = 2$, when applied in Lemma 2 and

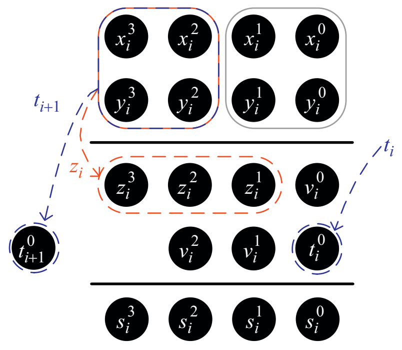


Fig. 14. [0, 11] input ODDS adder with [0, 15] output.

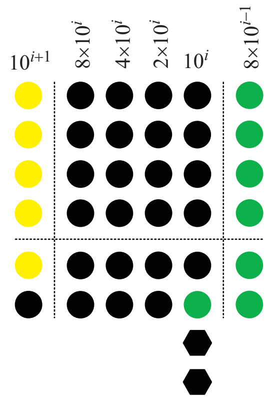


Fig. 15. ODDS (4; 2) compressor for the ith decimal digit-slice.

Table 3
Comparison between delay- and area-improved reduction methods.

Reduction method	# Of reduction levels			Delay (FO4)	Area (NAND2)
	(4; 2)	(4; 1)	(2; 1)		
Delay-improved	0	1	3	31.83	43390
Area-improved	3	0	3	36.00	33885

Theorem 1, respectively. Therefore, only x_3, y_3, x_2, and y_2 contribute in the computation of the bits of z_i as shown in the following equations. Fig. 14 depicts the partitioning and details of addition scheme. The latency of this adder is about six logic levels. Therefore, the overall delay of the (4; 1) BCD-to-ODDS compressor amounts to thirteen logic levels, while the same functionality based on regular ODDS adders leads to sixteen logic levels for the latency.

$$t_{i+1}^0 = x_i^3y_i^3 + x_i^3y_i^2 + x_i^2y_i^3,$$

$$z_i^3 = x_i^3\overline{y_i^3y_i^2} + x_i^2y_i^2 + \overline{x_i^3x_i^2}y_i^3, z_i^2 = x_i^3y_i^3 + x_i^2\overline{y_i^3y_i^2} + \overline{x_i^3x_i^2}y_i^2,$$
$$z_i^1 = x_i^3y_i^3 + x_i^3y_i^2 + x_i^2y_i^3,$$

$$v_i^0 = x_i^0 \oplus y_i^0, v_i^1 = (x_i^1 \oplus y_i^1) \oplus x_i^0y_i^0, v_i^2 = x_i^1y_i^1 + x_i^1x_i^0y_i^0 + x_i^0y_i^1y_i^0,$$

$$s_i^0 = v_i^0 \oplus t_i^0, s_i^1 = (z_i^1 \oplus v_i^1) \oplus v_i^0t_i^0,$$
$$s_i^2 = (z_i^2 \oplus v_i^2) \oplus (z_i^1v_i^1 + z_i^1v_i^0t_i^0 + v_i^1v_i^0t_i^0),$$

$$s_i^3 = z_i^3 + z_i^2v_i^2 + (z_i^2 + v_i^2)(z_i^1v_i^1 + z_i^1v_i^0t_i^0 + v_i^1v_i^0t_i^0).$$

5.2.2. *Area-improved reduction method*

To reduce area consumption, we propose a reduction scheme that heavily makes use of binary (4; 2) compressors. In this design, the $2n$ BCD numbers can be reduced to n ODDS numbers via a restricted-input ODDS adder. Then special ODDS (4; 2) compressors that are composed of four binary (4; 2) compressors can be used as the main reduction building block. However, each of these blocks requires a +0/6 correcting operation that prolongs the reduction time. The details of this alternative reduction method appear in Section 5.2.2.1, below. Table 3 shows the number of reduction levels, delay, and area consumption in the two delay- and area-improved methods that are based on (4; 1) BCD-to-ODDS (Section 5.2.1.1) and ODDS (4; 2) compressors (Section 5.2.2.1). The column labeled as (2; 1) refers to all the varieties of ODDS adders, which perform as (2; 1) compressors.

5.2.2.1. Partial product reduction via ODDS (4; 2) compressors. The $2n$ BCD numbers, at the first level of reduction process, are reduced to n ODDS numbers via special restricted-input ODDS adders. The circuitry of these adders turns out to be exactly the same as the adder of Fig. 14. At the next reduction level, a collection of four equally weighted ODDS digits can be compressed, via four binary (4; 2) compressors, to two ODDS digits and two correction 0/6 values. Fig. 15 depicts a dot notation representation of this ODDS (4; 2) compressor, for the ith decimal digit-slice, where a circular dot denotes a posibit and a hexagon dot represents a {0, 6} two valued digit (twit [14]). The logical values 0 and 1, of this twit are mapped to arithmetic values 0 and 6, respectively.

Fig. 16 shows how these compressors can be used for 32-to-1 reduction of BCD numbers. At the first level of reduction sixteen restricted-input ODDS adders (i.e., ODDS (2; 1) compressors) add pairs of BCD numbers and produce ODDS sums. In the next three reduction levels there are, per each decimal digit-slice, four, two, and one ODDS (4; 2) compressors that produce eight, four, and two ODDS digits and eight, four, and two {0, 6} twits, respectively.

These fourteen twits are sent to a 4 bit counter that is specially designed (Fig. 17) to receive the input twits as soon as they are produced by the ODDS (4; 2) compressors. However, each output bit is regarded as a {0, 6} twit to go with the counter's inputs. Furthermore the twits, with weights 2, 4, and 8, each can be decomposed to two bits with weights 2 and 10, 4 and 20, and 8 and 40, respectively.

The counter's output per the ith decimal digit-slice is shown in the first stage of Fig. 18, where the three bits in the first row come from the previous digit-slice. The three posibits and the single twit that are enclosed in a curve are partitioned to derive the transfer t_{i+1} via the following equations:

$$t_{i+1}^0 = y_i^3y_i^2 + y_i^3x_i^2 + y_i^2x_i'^0 + x_i^2x_i'^0 + y_i^3x_i'^0, t_{i+1}'^0 = y_i^3x_i^2y_i^2x_i'^0$$

The other bits can be computed as:

$$z_i^3 = y_i^3\overline{x_i^2y_i^2}\,\overline{x_i'^0} + \overline{y_i^3}x_i^2y_i^2\overline{x_i'^0} + y_i^3\overline{x_i^2}y_i^2x_i'^0 + y_i^3x_i^2\overline{y_i^2}x_i'^0,$$

$$z_i^2 = \overline{x_i^2y_i^2}x_i'^0 + \overline{y_i^3x_i^2}y_i^2\overline{x_i'^0} + \overline{y_i^3}x_i^2\overline{y_i^2x_i'^0} + y_i^3x_i^2y_i^2\overline{x_i'^0} + \overline{y_i^3}x_i^2y_i^2x_i'^0,$$

$$z_i^1 = y_i^3y_i^2\overline{x_i'^0} + y_i^3x_i^2\overline{x_i'^0} + y_i^3x_i^2y_i^2 + \overline{y_i^3}\,\overline{x_i^2}\,\overline{y_i^2}x_i'^0, v_i^1 = x_i^1 \oplus y_i^1,$$
$$v_i^2 = x_i^1y_i^1,$$

$$s_i^0 = (x_i^0 \oplus t_i^0) \oplus t_i'^0, s_i^1 = (z_i^1 \oplus v_i^1) \oplus (x_i^0t_i^0 + x_i^0t_i'^0 + t_i^0t_i'^0),$$

$$s_i^2 = (z_i^2 \oplus v_i^2) \oplus (z_i^1v_i^1 + (z_i^1 + v_i^1)(x_i^0t_i^0 + x_i^0t_i'^0 + t_i^0t_i'^0)),$$

$$s_i^3 = z_i^3 + (z_i^2v_i^2) + (z_i^2 + v_i^2)(z_i^1v_i^1 + (z_i^1 + v_i^1)(x_i^0t_i^0 + x_i^0t_i'^0 + t_i^0t_i'^0))$$

For the fifth reduction level, there is an ODDS number from the fourth level compressor and another one from the counter and the

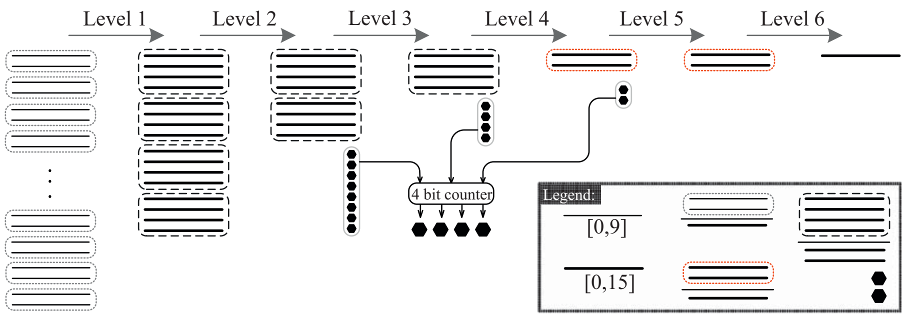


Fig. 16. The (32; 1) BCD-to-ODDS compressor.

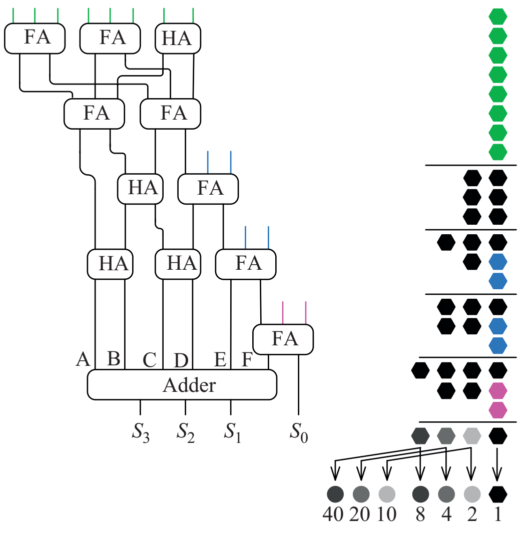


Fig. 17. 4-bit counter used as {0, 6} twit counter.

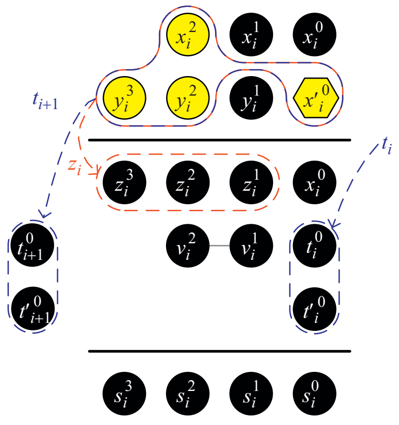


Fig. 18. Special ODDS adder.

subsequent special ODDS adder. One regular ODDS adder takes care of this and produces the final ODDS digits.

5.3. *Final ODDS-to-BCD conversion*

The ODDS-to-BCD conversion, as any other redundant to nonredundant conversion, cannot be performed in constant time. Given a k-digit ODDS result $s_{k-1}, \ldots, s_0$, the collective value of each $s_i = s_i^3 s_i^2 s_i^1 s_i^0$ $(0 \le i \le k-1)$ and a coming decimal carry c_i^{in} may be decomposed to a decimal carry c_i^{out} and a BCD digit $b_i^3 b_i^2 b_i^1 b_i^0$. Carry propagation is inevitable due to dependency of c_i^{out} on c_i^{in}. The conversion cell per each position i may be designed as a 5-input/5-output $2\Delta G$ logic (equation-set (8)).

$$\begin{aligned}
&b_i^0 = c_i^{in} \oplus s_i^0, b_i^1 = c_i^{in}\left(\overline{s_i^3}(s_i^1 \oplus s_i^0)\right) + \overline{c_i^{in}}(s_i^3 s_i^2 \overline{s_i^1} + \overline{s_i^3} s_i^1),\\
&b_i^2 = c_i^{in}\left((s_i^3 s_i^2 s_i^1 + \overline{s_i^3} s_i^2 \overline{s_i^0}) + \left((\overline{s_i^3 s_i^2})(s_i^1 s_i^0)\right) + s_i^2 \overline{s_i^1} s_i^0\right)\\
&\qquad + \overline{c_i^{in}}(\overline{s_i^3} s_i^2 + s_i^2 s_i^1),\\
&b_i^3 = c_i^{in}(\overline{s_i^3} s_i^2)(s_i^1 s_i^0) + \overline{c_i^{in}}(s_i^3 \overline{s_i^2 s_i^1}) + \left((s_i^3 \overline{s_i^2})(\overline{s_i^1 s_i^0})\right),\\
&c_i^{out} = c_i^{in} s_i^3 s_i^0 + s_i^3 s_i^2 + s_i^3 s_i^1 \qquad (8)
\end{aligned}$$

The inter-digit carry propagation may be speeded-up using standard carry look-ahead cells, where the ODDS propagate and generate signals are ruled by equation-set (9). The conversion latency is 14 and 16 ΔG for multipliers with the short and long decimal word sizes, based on IEEE 754-2008 standard (i.e., 16, and 34 digits), respectively. More accurate FO4 latency figures are provided in the next section.

$$g_i = s_i^3(s_i^2 + s_i^1), p_i = s_i^3 s_i^0 \qquad (9)$$

6. Comparison with previous results

The works on fully redundant decimal addition may be divided in two categories:

I. *Balanced signed digit set*: this category includes the pioneering contribution in [29], followed in [26], onto the recent work due to [22] and the most recent one in [12]. All these contributions use balanced signed digit sets [α, α], within the framework of signed digit number systems originally introduced by Avezionis [1], where α = 6, 7, 9 and 7, respectively.
II. *Other redundant digit sets*: the works by [9,4,16] and this work use decimal digit sets [0, 10], [0, 18], [−8, 9] and [0, 15], respectively.

The previous decimal carry-free addition schemes basically adapt Algorithm 1 for the corresponding digit sets. A major advantage of the ones in Category I, above, is the possibility of efficient use of addition circuitry for subtraction. This is obviously not as efficiently possible for the unsigned digit sets in Category II above. However, this is fine for decimal multiplication (see also Section 5). The same is true for an iterative multiplier due to [9], with a fully redundant BCD CSA as part of the accumulation logic, where the latter is implemented via two cascaded BCD full adders [24].

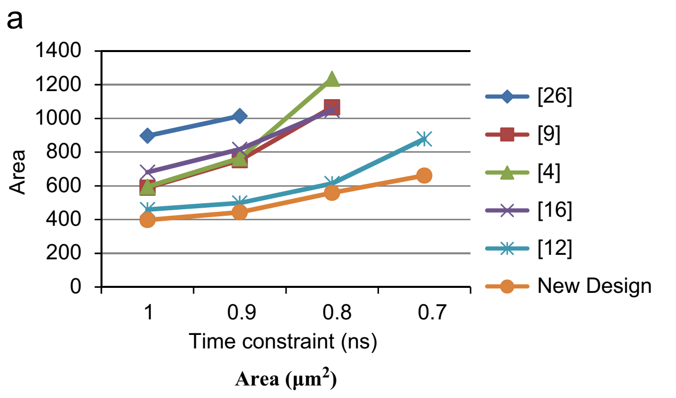
We have used the design information, on fully redundant decimal adders, provided by the previous contributions and reevaluated their performance based on logical effort analysis

Table 4
Performance comparison for fully redundant decimal adders/PPR cells.

Adder/PPR cell	Delay (FO4)	Ratio	Area (NAND2)	Ratio
[29]	16.11	2.05	142	1.38
[26]	13.56	1.72	122	1.18
[9]	8.89–10.45	1.13–1.14	109	1.06
[22]	14.74	1.87	428	4.15
[4]	9.00–12.17	1.15–1.32	194	1.88
[16]	9.06–10.44	1.15–1.13	150	1.46
[12]	8.56–9.96	1.09–1.08	184	1.79
Proposed	7.86–9.20	1–1	103	1

Table 5
Synthesis results for the eight studied designs.

Design	Delay (ns)	Ratio	Power dissipation				Area (μm^2)	Ratio
			Dynamic (mW)	Ratio	Static (pW)	Ratio		
[29]	1.16	1.84	2.13	1.75	70.98	2.20	1901	1.68
[26]	0.87	1.37	1.65	1.35	44.34	1.37	1480	1.31
[9]	0.75	1.18	1.48	1.22	37.87	1.17	1354	1.19
[22]	1.10	1.75	4.04	3.32	95.79	2.96	3139	2.77
[4]	0.77	1.22	1.54	1.27	45.78	1.42	1390	1.23
[16]	0.77	1.21	1.52	1.25	46.77	1.45	1289	1.14
[12]	0.69	1.09	1.17	0.96	37.24	1.15	1100	0.97
Proposed	0.63	1.00	1.22	1.00	32.31	1.00	1133	1



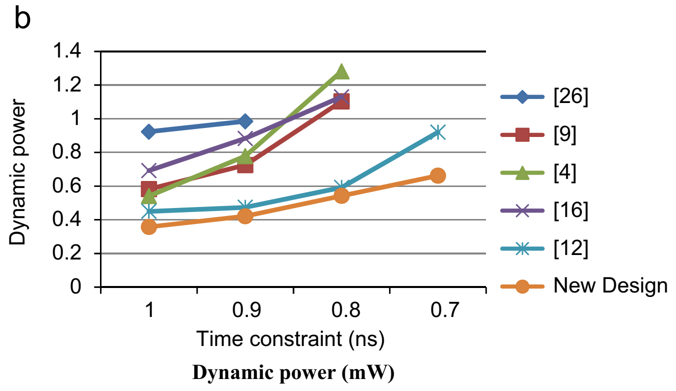


Fig. 19. Area (a) and power (b) of a digit-slice of six redundant decimal adders (curves continue as far as the corresponding designs meet the time constraints).

and same assumptions (details are described in Appendix for ease of verification). Table 4 shows that the ODDS adder outperforms the previous relevant works. Note that, for the five higher performance cases, we provide two latency figures corresponding to the minimum and unlimited branching (see details in Appendix). The actual delay for practical implementation falls in between.

We have checked the correctness of all the eight designs that are referred to in Table 4, by running exhaustive tests on the corresponding VHDL codes. Furthermore, to achieve more reliable performance measures, we have used these codes to synthesize all the adders by Synopsys Design Compiler using a target library based on TSMC 0.13 μm standard CMOS process. For dynamic and leakage power, Synopsys Power Compiler has been used. The synthesis was undertaken for a range of time constraints 0.6–1.0 ns, where the results are summarized in Table 5 for the time constraint 0.6 ns that is the longest not met by any of the eight designs. However, the experienced latencies that are naturally more than 0.6 ns, are shown along the corresponding other measures. Note that all figures, in Table 5, relate to one decimal digit-slice.

Similar results for time constraints 0.7, 0.8, 0.9 and 1.00 ns are compiled in curves of Fig. 19. Note that each curve stops where the corresponding design has not met the relevant time constraint. Also note that the works in [29, 22], not included in Fig. 19, do not even meet the 1.0 ns constraint.

The latter results notwithstanding, the ODDS adder may be better assessed via its application as a means of partial product reduction (PPR). The latest relevant work is due to [30], where a 16×16 digit decimal multiplier based on radix-5 signed digit encoding of multiplier digits is designed. The differences of the work in [30] and the two proposed ODDS multipliers (Sections 5.2.1 and 5.2.2) for each of the three stages of multiplication are listed below. Table 6 shows the comparison results, where the overall figures are less than the sum of corresponding stage figures. This is due to an independent evaluation along the critical path that traverses through all the three stages.

Table 6
Performance comparison of decimal multipliers.

Multiplier	Stage	Delay (FO4)	Ratio	Area (NAND2)	Ratio
[30]	(a) PPG	5.17	1	2736	1
	(b) PPR	40.87	1	30,179	1
	(c) Final	15.87	1	3585	1
	Overall	60.55	1	36,500	1
Proposed	(a) PPG	6.10	1.18	3295	1.20
	(b-1) PPR	31.83	0.78	43,390	1.44
	(b-2) PPR	36.00	0.88	33,885	1.12
	(c) Final	10.41	0.66	1940	0.54
	Overall(b-1)	46.88	0.77	48,625	1.33
	Overall(b-2)	51.06	0.84	39,120	1.07

(a) *Partial product generation (PPG)*: the proposed scheme precomputes three unsigned multiples 2*X*, 4*X*, and 5*X*, as is the case in [9]. However, Vazquez et al., produce $-X$, $\pm 2X$, 5*X*, and 10*X*. Although the clever use of 4-2-2-1 encodings of decimal digits leads to minimal cost negation, the signed multiples raise to an extra decimal digit per each partial product.

(b) *Partial product reduction (PPR)*: recalling that the hatched boxes in the recursive VLSI-friendly reduction tree of Fig. 12 are actually composed of two restricted-input ODDS adders the reduction ratio is 2-to-1 per reduction level (b-1 in Table 6). The reduction ratio in the proposed alternative area-improved method (b-2 in Table 6), based on (4; 2) compressors, is less than 2-to-1 due to +0/6 corrections (see Section 5.2.2.1). However, the reduction ratio in the scheme proposed in [30] is 3-to-2. The faster reduction step of the latter notwithstanding, this lower reduction ratio leads to slightly slower overall reduction process (Table 6).

(c) *Final redundant to BCD conversion*: the double-BCD-to-BCD converter of [30] is expectedly slower than the proposed ODDS-to-BCD converter. The reason is partly due to simpler

4 bit to 4 bit functionality of the latter VS the 8 bit to 4 bit transformation of the former. The actual comparison figures appear in Table 6.

7. Conclusions

We proposed a new fully redundant carry-free decimal addition scheme with overloaded decimal digit set (ODDS). This ODDS adder was designed based on:

- Immediate availability of position-sums in weighted bit-set (WBS) encoding.
- Partitioning the bits of the latter for fast computation of transfer digit and interim sum. The theoretical basis of this new technique was discussed.
- Fast digit-parallel computation of the sum.

The ODDS adder was shown, based on the logical effort analysis (Table 4) and synthesis results (Fig. 19), to outperform three of the previous fully redundant decimal adders due to [29,26, 22] and significantly improve upon three other relevant designs due to [9,4, 16]. However, the proposed ODDS adder cannot be efficiently adapted to perform subtraction. Therefore, to show the practical advantages of this adder, we managed to present its efficient application in parallel decimal multiplication.

In the proposed multiplier, a fast generator of unsigned multiples of multiplicand was designed, the ODDS adder and its restricted-input varieties were effectively used in parallel PPR, and a final converter of ODDS product to BCD was presented. Two relatively fast (delay-improved) and slow (area-improved) reduction schemes were proposed. The faster, that consumes more area, uses ODDS adders for 2-to-1 reduction within a VLSI-friendly reduction tree. However, the slower (with less area) employs binary (4; 2) compressors augmented with +0/6 correction logic. The logical effort measures were compared with the latest contribution in [30]. The latencies of the two proposed ODDS multipliers are 77% and 84% of that of the referenced multiplier. This is achieved at the cost of 33% and 7% more area, respectively.

Further research is ongoing on the possibility of using the proposed partitioning technique (Section 3) within the framework of decimal floating point arithmetic. Moreover, applying the same technique for power-of-two radices (e.g., radix-16) may also find grounds in redundant digit floating point arithmetic [11].

Appendix. Assumptions for logical effort analysis

The authors of the previous fully redundant decimal adders have evaluated their works via gate level analysis. Therefore, we primarily evaluated the latency of the ODDS adder in terms of logic levels (see Section 4). However, since the more recent works in [30,4,12] have been evaluated via logical effort analysis we also, for a more reliable comparison, used logical effort to evaluate the performance of the proposed adder and multipliers. In this endeavor, we do not aim at precise evaluation results that may be best achieved via a synthesis tool. Therefore, 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. The latency is 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. The latency and area of all the fully redundant decimal adders presented in [29,26,22], the proposed multipliers and that of [30] were evaluated under the following assumptions (a) and (b), respectively. However, since the latency of the proposed ODDS adder and the ones due to [9,4,12] all fall within 8–12 logic levels, we provide two latency figures due to the minimum and maximum branching. Note that in practical designs the latency would actually fall somewhere in between the two extremes.

(a) *Minimum branching*: the outputs of all the gates drive at most one other gate (i.e., the branching effort is one). This mode of latency evaluation is similar to what synthesis tools do when the user calls for the lowest possible time constraint. Therefore, the first level gates (i.e., towards the inputs) are replicated as much as needed [27], where the input signals, initiated from registers, are assumed to derive as many gates as required.
For further clarification, we applied this practice on a conventional (4; 2) compressor logic [5] that is depicted by Fig. 20. The result is shown in Fig. 21a with 4.69 FO4 delay.

(b) *Unlimited branching*: no limit for branching is enforced in favor of optimized area consumption. This option is also applied for a (4; 2) compressor. The delay, as shown in Fig. 21b, is 5.93 FO4.

For more details on logical effort evaluation see [31].

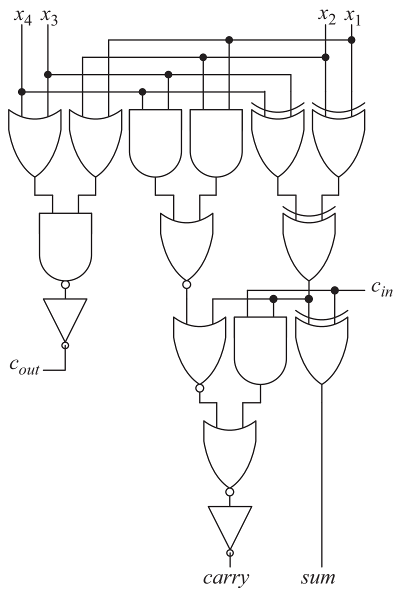


Fig. 20. A conventional (4; 2) compressor logic.

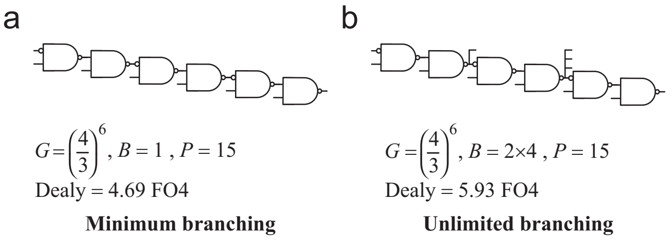

a
b
$G=\left(\frac{4}{3}\right)^6$, $B=1$, $P=15$
Dealy = 4.69 FO4
Minimum branching
$G=\left(\frac{4}{3}\right)^6$, $B=2\times4$, $P=15$
Dealy = 5.93 FO4
Unlimited branching

Fig. 21. Critical path delay estimation (XOR gates implemented via NAND2 gates).

References

[1] A. Avizienis, Signed-digit number representations for fast parallel arithmetic, IRE Transactions on Electronic Computers EC-10 (1961) 389–400 (September 1961).
[2] G.W. Bewick, Fast Multiplication: Algorithms and Implementation, Doctoral Dissertation, Stanford University, 1994.
[3] F.Y. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz, S.R. Carlough, The IBM z900 decimal arithmetic unit, Asilomar Conference on Signals, Systems, and Computers 2 (2001) 1335–1339 (November 2001).
[4] I.D. Castellanos, James E. Stine, Compressor trees for decimal partial product reduction, ACM Great Lakes Symposium on VLSI (2008) 107–110 (May 2008).
[5] C.H. Chang, J. Gu, Mi. Zhang, Ultra low-voltage low-power CMOS 4-2 and 5-2 compressors for fast arithmetic circuits, IEEE Transactions on Circuits and Systems I 51 (10) (2004) 1985–1997 (October 2004).
[6] M.F. Cowlishaw, Decimal Floating-point: Algorism for Computers, Proceedings of the Sixteenth IEEE Symposium on Computer Arithmetic (2003) 104–111 (June 2003).
[7] L. Dadda, Multioperand parallel decimal adder: a mixed binary and BCD approach, IEEE Transactions on Computers 56 (10) (2007) 1320–1328 (October 2007).
[8] L. Eisen, J.W. Ward III, H.-W. Tast, N. Mäding, J. Leenstra, S.M. Mueller, C. Jacobi, J. Preiss, E.M. Schwarz, S.R. Carlough, IBM POWER6 accelerators: VMX and DFU, IBM Journal Research and Development 51 (6) (2007) 633–684 (November 2007).
[9] M.A. Erle, M.J. Schulte, Decimal multiplication via carry-save addition, Conference on Application-Specific Systems, Architectures, and Processors (2003) 348–358 (June 2003).
[10] M.A. Erle, E.M. Schwartz, M.J. Schulte, Decimal multiplication with efficient partial product generation, Proceedings of the Seventeenth IEEE Symposium on Computer Arithmetic (2005) 21–28 (June 2005).
[11] H. Fahmy, M.J. Flynn, The case for a redundant format in floating-point arithmetic, Proceedings of the Sixteenth IEEE Symposium on Computer Arithmetic (2003) 95–102 (June 2003).
[12] S. Gorgin, G. Jaberipur, Fully redundant decimal arithmetic, Proceedings of the Ninteenth IEEE Symposium on Computer Arithmetic, Portland, USA (2009) 145–152 (June 8–10, 2009).
[13] Institute of Electrical and Electronics Engineers, IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, August 2008.
[14] G. Jaberipur, B. Parhami, M. Ghodsi, Weighted two-valued digit-set encodings: unifying efficient hardware representation schemes for redundant number systems, IEEE Transactions Circuits and Systems I 52 (7) (2005) 1348–1357 (July 2005).
[15] G. Jaberipur, B. Parhami, Stored-transfer representations with weighted digit-set encodings for ultrahigh-speed arithmetic, IEE Proceedings of the Circuits, Devices, & Systems 3 (2007) 102–110 (February 2007).
[16] A. Kaivani, G. Jaberipur, Fully redundant decimal addition and subtraction using stored-unibit encoding, Integration the VLSI journal (2009)10.1016/j.vlsi.2009.04.001.
[17] R.D. Kenney, M.J. Schulte, M.A. Erle, A high-frequency decimal multiplier, IEEE International Conference on computer Design: VLSI in Computers and Processors (ICCD) (2004) 26–29 (October 2004).
[18] R.D. Kenney, M.J. Schulte, High-speed multioperand decimal adders, IEEE Transaction on Computer 54 (8) (2005) 953–963 (August 2005).
[19] Peter Kornerup, Digit-set conversions: generalizations and applications, IEEE Transactions on Computers 43 (5) (1994) 622–629 (May 1994).
[20] T. Lang, A. Nannarelli, A radix-10 combinational multiplier, Proceedings of the Fortieth Asilomar Conference on Signals, Systems, and Computers (2006) 313–317 (November 2006).
[21] H. Makino, Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara, K. Mashiko, An 8.8 ns 54×54 bit multiplier with high speed redundant binary architecture, IEEE Journal of Solid-State Circuits 31 (6) (1996) 773–783 (June 1996).
[22] H. Nikmehr, B.J. Phillips, C.C. Lim, A decimal carry-free adder, In: Proceedings of the SPIE Conference Smart Material Nano-, Micro-Smart Systems, pp. 786–797, December 2004.
[23] B. Parhami, Generalized signed-digit number system: a unifying framework for redundant number representation, IEEE Transaction on Computer 39 (1) (1990) 89–98.
[24] M. Schmookler, A. Weinberger, High speed decimal addition, IEEE Transaction on Computers C-20 (8) (1971) 862–866 (August 1971).
[25] Hermann Schmid, Decimal computation, Wiley, New York, 1974.
[26] B. Shirazi, D.Y. Yun, C.N. Zhang, RBCD: redundant binary coded decimal adder, IEE Proceedings of the Computer & Digital Techniques (CDT) 36 (2) (1989) (March 1989).
[27] Ankur Srivastava, Ryan Kastner, Chunhong Chen, Majid Sarrafzadeh, Timing driven gate duplication, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12 (1) (2004) 42–51 (January 2004).
[28] I.E. Sutherland, R.F. Sproull, D. Harris, Logical Effort: Designing Fast CMOS Circuits, Morgan Kaufmann, Los Altos, ISBN 1558605576, 1999.
[29] A. Svoboda, Decimal adder with signed digit arithmetic, IEEE Transactions on Computers C-18 (3) (1969) 212–215 (March 1969).
[30] A. Vazquez, E. Antelo, P. Montuschi, A new family of high-performance parallel decimal multipliers, Proceedings of the Eighteenth IEEE Symposium on Computer Arithmetic (2007) 195–204 (June 2007).
[31] Vazquez Alvaro Alvarez, High-Performance Decimal Floating-Point Units, Ph.D. thesis, Santiago de Compostela, January 2009.