Microprocessors and Microsystems 61 (2018) 96–107

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

Improving the area of fast parallel decimal multipliers

Mário Véstias*,[b], Horácio Neto[a]

[a] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[b] INESC-ID, ISEL, Instituto Politécnico de Lisboa, Portugal

ARTICLE INFO

Keywords:
Decimal multiplication
Parallel multiplication
Excess-6 coding
5221 coding
FPGA

ABSTRACT

Financial and commercial applications depend on decimal arithmetic because they must produce results that match exactly those obtained by human calculations. Decimal multiplication is a frequently used operation in these applications and also in the design of decimal floating-point units. In this paper we propose a new architecture for parallel decimal multiplication that improves the area of previous decimal multipliers while keeping the best performances. A decimal adder [1] based on a mixed BCD/excess-6 representation of the operands is utilized. A new partial product generation unit is proposed based on a 5221 recoding of the multiplier digits. With the proposed multiplier, we are able to improve on state-of-the-art parallel decimal multipliers targeting LUT-6 FPGAs. Compared to previous decimal multipliers, implementation results for 2, 4, 8, 16, 32 and 34-digits show that the proposed multiplier achieves over 20% better area without performance degradation.

1. Introduction

The beginning of 2000s was an important mark for decimal computing. Previously, the few attempts to bring decimal hardware to processors were somehow unsuccessful since the tradeoff between applicability and hardware cost did not justify a dedicated decimal arithmetic unit. Starting in 2000s, this scenario has changed with the new financial and business applications and the technological innovations that permitted multicore binary/decimal computing [2,3].

The trend to implement decimal arithmetic in hardware has been initiated with the inclusion of special hardware units for decimal floating-point arithmetic in the IBM eServer z900 [4], the IBM POWER6 [5] and IBM z10 [6]. The last revision of the IEEE standard for floating-point arithmetic [7] has also included specific definitions and rules for decimal operations.

Many financial and commercial applications like accounting, banking, currency conversion among others are quite numerical data computing and so are typically executed in high-performance mainframes. In many databases of these applications, numbers are in decimal format and so their calculations must follow the conventions of decimal arithmetic and must keep a word length enough to support the precision required by these applications. Also, since many decimal numbers cannot be represented exactly as binary numbers with a finite number of bits, arithmetic operations must be done directly over decimal numbers [2]. Current applications may need over 32 precision digits to represent exactly a large set of decimal values found in the databases of these applications.

To execute decimal operations using binary arithmetic hardware, specific software algorithms based on binary arithmetic are used. As such, software libraries for decimal arithmetic are supported by Intel [8], ANSI C [9] and GCC [10]. However, software solutions are very slow, typically executing three or four orders of magnitude slower than binary arithmetic implemented in hardware [2]. Despite this, the requirement for decimal arithmetic hardware depends on the application requirements. Many commercial applications do not require high decimal arithmetic performance and so decimal floating point still relies on software approaches using typical binary arithmetic.

However, the fast increase of commercial and financial transactions requires fast decimal arithmetic computing to meet real-time requirements and exact computations. The problem is that working with 34 digits of precision requires a large area for high-performance which for some processor designers is still a high cost without enough return. Therefore it is important to find methods for the design of fast and low cost decimal arithmetic units and possibly good merging solutions of binary and decimal arithmetic.

Whether the next generation of arithmetic units include decimal units or not depends on the volume and characteristics of applications requiring decimal arithmetic computing. Reducing the performance/area gap between binary and decimal arithmetic will also contribute for the dissemination of decimal arithmetic units.

Decimal multiplication is a very frequent operation in most decimal applications and one of the basic decimal operations specified in the

* Corresponding author.
E-mail addresses: mvestias@deetc.isel.pt (M. Véstias), hcn@inesc-id.pt (H. Neto).

https://doi.org/10.1016/j.micpro.2018.05.015
Received 11 January 2018; Received in revised form 2 May 2018; Accepted 21 May 2018
Available online 22 May 2018

IEEE 754–2008 standard. It is used directly as a basic operation and in the implementation of other arithmetic operations (e.g., division, square-root, exponentiation or logarithm) and as the main building block in decimal floating-point multiplication. Benchmarks [11] indicate that decimal multiplication can consume up to 27% of the total computation time of typical decimal applications.

Decimal multiplication is more complex to implement in hardware than binary multiplication due to the inherent difficulty to efficiently represent decimal numbers using a binary number system. Both bit and digit carries, as well as invalid results, must be considered in decimal multiplication in order to produce the correct result, which complicates the generation of partial products and their reduction.

Decimal hardware multipliers used in commercial processors are based on an iterative algorithm in order to reduce the hardware size of the implementation, which consequently reduces considerably its performance.

In iterative multipliers [12], the multiplicand is successively multiplied by one digit of the multiplier to generate a partial product, which is accumulated to produce the final decimal product. A few decimal multipliers have been proposed based on sequential units, such as [13,14], where a set of multiplicand multiples is generated in a preprocessing step and then selectively added according to the value of the multiplier digits.

Alternatively, an iterative decimal multiplier can be implemented using an $N \times 1$ BCD multiplier and a BCD carry chain adder that accumulates the partial results [15].

Parallel decimal multipliers have been proposed to improve performance, e.g. [16–18]. Basically, parallel multipliers generate all partial products in parallel which are then reduced to the final product using a decimal carry-save addition tree or a multioperand decimal adder.

In this paper, we propose a new method for parallel decimal multiplication. The method is based on the new decimal adder [1] which is optimized to be implemented in 6-input LUT (*Look-Up Table*) FPGA (*Field Programmable Gate Array*) and to use efficiently its carry-chain. The partial product computation is based on a 5221 encoding of the multiplier that takes full advantage of pre-computed $5\times$ and $2\times$ multiplicand multiples.

The results obtained with the multiplier implemented with the proposed methods reveal improvements in area for the same performance, when compared to the best state-of-the-art decimal multipliers.

This paper is organized as follows. Section 2 describes state-of-the-art proposals. Section 3 introduces the BCD/excess-6 decimal adder. Section 4 describes the proposed decimal multiplier. Section 5 presents the results of the new proposed decimal multiplier and compares them with state-of-the-art parallel decimal multipliers. Section 6 concludes the paper and proposes future directions for decimal multiplication.

2. Related work

Decimal fixed-point multiplication is generally accomplished by first generating partial products, which are then added and reduced to the final product.

Following the traditional hand-made multiplication, the partial products can be obtained using digit by digit multiplication. Digit multiplication can be implemented using lookup tables [19] (larger tables were considered in [20]), but the required circuitry and the delay associated with lookup table implementations makes them a valid solution only for small operand sizes. Alternatively, a signed digit radix-10 recoding was proposed in [21] to simplify the generation of partial products. However, the performance and area of hardware for decimal multiplication of proposals based on digit by digit multiplication are still far from those achieved by hardware implementations using multiplicand multiples to generate partial products.

The most common approach for partial product generation consists on generating all multiples of the multiplicand and then selecting the appropriate multiple for each multiplier digit. While simple, the method requires some multiples whose generation is not carry-free. In [22] only even multiples (2X, 4X, 6X, 8X) are produced a priori, while the remaining multiples are generated on demand by adding X to one of the even multiples. This is a slightly different method since some of the multiples are obtained from adding two other multiples. Following this approach, some authors have proposed to generate a restricted set of multiples from which the others are obtained by adding two of the previously calculated multiples.

In [14] multiples X, 2X, 4X, 5X are precomputed without carry-propagation. All other multiples are obtained from summing two of these precomputed multiples using a decimal 3:2 counter. In [16] only multiples X, 2X, 5X are precomputed. The other multiples are obtained by adding two of these multiples including their complements, which are obtained with an additional10′s complement operation. Negation is implemented by a9′s complement recoder, and the one increment is only applied to the least significant digit. The selection of the correct multiples to be added is obtained from a recoding of the multiplier digit, A, as $A = Y^U 5 + Y^L$, where $Y^U \in \{0, 1, 2\}$ and $Y^L \in \{-2, -1, 0, 1, 2\}$.

In [23] a new parallel decimal multiplier is proposed using two unconventional decimal encodings (4221 and 5211) and two architectures (radix-10 and radix-5) are applied to generate and reduce the partial product. In the radix-10 architecture, the multiplier is recoded into a signed-digit (SD) set [-5, 5], and n+1 partial products are selected according to the recoded multiplier. In the radix-5 architecture the multiplier is encoded as $A = Y^U 5 + Y^L$ as in [16]. In this case, ***2n*** partial products are generated. The partial product reduction uses only binary adders and recoders due to the specific encodings utilized. The 4221 and 5211 coding speeds-up the carry-save adder reduction, since the9′s complement is generated by simple bit inversion.

In [24] a decimal multiplier is proposed using a redundant decimal addition algorithm based on a weighted bit-set encoding. The method also generates double-BCD numbers using decimal multiples 2X, 4X, and 5X. The authors also proposed a decimal redundant adder to reduce the generated ***2n*** BCD partial products to a redundant number in the range of [0, 15]. A last step is done to convert the final redundant product to BCD encoding.

In [25] both multiplier and multiples of the multiplicand are recoded in signed-digit numbers. Then, a multioperand signed-digit addition algorithm is proposed to reduce the partial products.

Partial product reduction depends on decimal addition. The first approaches used a direct carry-propagation addition [26]. To speed-up the addition, carry-free adders started to be considered in [19], which proposed a carry save adder.

Many other approaches were proposed to speed-up decimal addition, e.g. [14,27]. However, these techniques were not designed for multioperand decimal addition, and therefore are not well suited for partial product reduction. In [28] three different techniques were proposed for multioperand decimal carry save addition. Two of them perform speculative addition by speculating BCD correction values and correcting intermediate results while adding the input operands. The third technique uses a binary carry-save adder tree and produces a binary sum. Combinational logic is then used to correct the sum and determine the carry into the next more significant digit. The non-speculative adders of the third approach have the best area-delay. In [29] a mixed binary and BCD addition is proposed. All digits in the same column of the multioperand tree are added in binary and then converted to decimal. The decimal operands obtained after the conversion are added using normal decimal adders. This multioperand adder was used to implemented a decimal multiplier in [18].

Decimal carry look-ahead adders were considered in [14] to implement a serial decimal multiplier and in [16] to implement a carry-save adder tree for a combinatorial decimal multiplier. In [17] the partial products are represented in decimal 4221 encoding. The reduction method reduces three partial products to two equally weighted

4221 decimal digits. The process ends with two double-length 4221 encoded decimal operands that are recoded to BCD before the final BCD addition.

In [30–32] a different approach was proposed for decimal multiplication, using binary multipliers. BCD operands are first converted to binary, multiplied with a binary multiplier and then converted back to BCD. To reduce the overhead associated with the converters, large operand multiplications are subdivided into 4 × 4 blocks and each pair of 4−digit sub-operands is converted to binary, multiplied in binary and then converted to BCD. The method is able to use existing binary multipliers, such as the embedded multipliers available in FPGAs. However, is costly in terms of the area and delay associated with the binary to BCD conversion [32] and vice-versa [33].

Several works were proposed for decimal multiplication targeting FPGA technology. The initial approaches simply adapted the methods for partial product generation and reduction proposed for ASIC (*Application-Specific Integrated Circuit*). In [15] both combinatorial and sequential decimal multipliers were proposed targeting a Virtex-4 FPGA from Xilinx. Partial product generation is implemented using digit by digit multiplication. Each digit of the multiplier is binary multiplied by one digit of the multiplicand. The 7-bit binary results are then converted as in [34] to obtain a two-digit BCD number. Partial products are reduced using a tree of fast BCD carry adders [35].

In [36] a parallel BCD multiplier on a Xilinx Virtex-II FPGA was proposed. The multiplier is based on the methods proposed in [17] and [23], that is, it uses signed digit radix-10 recoding of the multiplier and pre-computes all multiples of the multiplicand from -5X to 5X. The decimal partial products are reduced using a 4221 carry-save adder tree.

In [37] two methods were considered for implementation on FPGA. One uses signed digit radix-10 recoding and the other signed digit radix-5 recoding of the multiplier [16,17,23]. The authors conclude that the solution based on radix-5 recoding is better. The multiplier has a partial product generation unit and a partial product reduction unit. The former generates two partial products for each multiplier digit recoded in sign-digit radix-5. Only the multiple 2X is precomputed, while the remaining BCD multiplicand multiples are computed on-the-fly. All partial products are then added using the partial product reduction unit that consists of a tree of BCD carry-ripple adders. The architecture was mapped on a 6-input LUT FPGA.

In [38] the Karatsuba-Ofman's algorithm was used to reduce the area of the parallel decimal multipliers on FPGA. Improvements of around 30% were achieved, but with a penalty on the delay.

Recently, a novel BCD multiplier was proposed in [39] using a 1 × 1 digit multiplier. The two digit results are organized according to their two-digit positions to generate the 2-digit column-based partial products. Partial product reduction uses a binary decimal compressor structure. The results show some improvements compared to previous results.

In this paper, we propose a new method for partial product generation based on the 5221 recoding of the multiplier digits. Combining this technique and the recent decimal adder proposed in [1], we were able to improve the state-of-the-art parallel decimal multipliers targeting 6-input LUT FPGAs.

3. Decimal adder

The typical decimal addition method of two BCD digits, w and z, adds the numbers in binary and corrects the result, that is, if $w + z \leq 9$ the result is correct, but if $w + z \geq 10$ then the result must be corrected by adding six. If $10 \leq w + z \leq 15$, then the carry out is only correct after the second addition, which significantly penalizes its propagation delay.

An alternative is to always pre-add six. In this case, the result is correct if $w + z + 6 \geq 16$, but if $w + z + 6 \leq 15$ then the sum digit must be corrected by subtracting six. However, the carry out bit is always correct and does not depend on the correction step, and therefore there is no penalization on its propagation delay. Further, the carry out bit also identifies if the result is correct or a correction step is needed to convert the sum digit back to BCD, that is, the carry out is 1 whenever $w + z + 6 \geq 16$ (no correction needed).

A $w + z + 6$ BCD adder is equivalent to a BCD + excess-6 adder $w + z'$, where w is a BCD digit and $z' = z + 6$ is an excess-6 digit.

The excess-6 code represents each decimal digit with the sum of its BCD code with the constant 6. Table 1 shows the 8421, 5221 and excess-6 representations, used in this work.

Table 1
Decimal digit representations.

Digit	BCD (8421)	5221	excess-6
0	0000	0000	0110
1	0001	0001	0111
2	0010	0010	1000
3	0011	0011	1001
4	0100	0110	1010
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1110	1111

Therefore, adding two decimal digits, one represented in BCD and the other represented in excess-6, always produces a correct carry. The sum digit may have to be corrected depending on the required representation. For example, if the sum is to be represented in BCD and the carry signal is zero then the result must be corrected from excess-6 by subtracting six. Otherwise, it is already correct. The same applies if the final representation should be in excess-6 but with contrary carry conditions.

This method was used in [1] to efficiently design decimal adders. To better understand the decimal multiplier proposal, we briefly describe the decimal adder to be used in this work.

Considering a single digit adder in which each of the digits can be independently represented in BCD or in excess-6, the constant 6 may have to be added or subtracted to one of the operands so that the addition is adequately performed (see Table 2).

The BCD/excess-6 adder is implemented with the propagate and generate signals of the carry-chain of the FPGA. The propagate signals of the BCD/excess-6 adder are conditionally generated as follows (see [1] for further details):

$$p[3] = \begin{cases} w[3] \oplus (w[2] \vee w[1]) \oplus z[3] & \text{if } wbcd = zbcd = 1 \\ \overline{w[3]} \oplus (w[2]\, w[1]) \oplus z[3] & \text{if } wbcd = zbcd = 0 \\ w[3] \oplus z[3] & \text{if } wbcd \neq zbcd \end{cases}$$

$$p[2] = \begin{cases} \overline{w[2]} \oplus w[1] \oplus z[2] & \text{if } wbcd = zbcd = 1 \\ w[2] \oplus w[1] \oplus z[2] & \text{if } wbcd = zbcd = 0 \\ w[2] \oplus z[2] & \text{if } wbcd \neq zbcd \end{cases}$$

$$p[1] = \begin{cases} \overline{w[1]} \oplus z[1] & \text{if } wbcd = zbcd \\ w[1] \oplus z[1] & \text{if } wbcd \neq zbcd \end{cases}$$

$$p[0] = w[0] \oplus z[0]$$

where *wbcd* and *zbcd* indicate if the digits, w and z, respectively, are represented in BCD (logic 1) or in excess-6 (logic 0).

Table 2
Functionality of a single digit BCD/excess-6 Addition.

w	z	Action
BCD	BCD	$w \rightarrow w + 6$
BCD	excess-6	none
excess-6	BCD	none
excess-6	excess-6	$w \rightarrow w - 6$

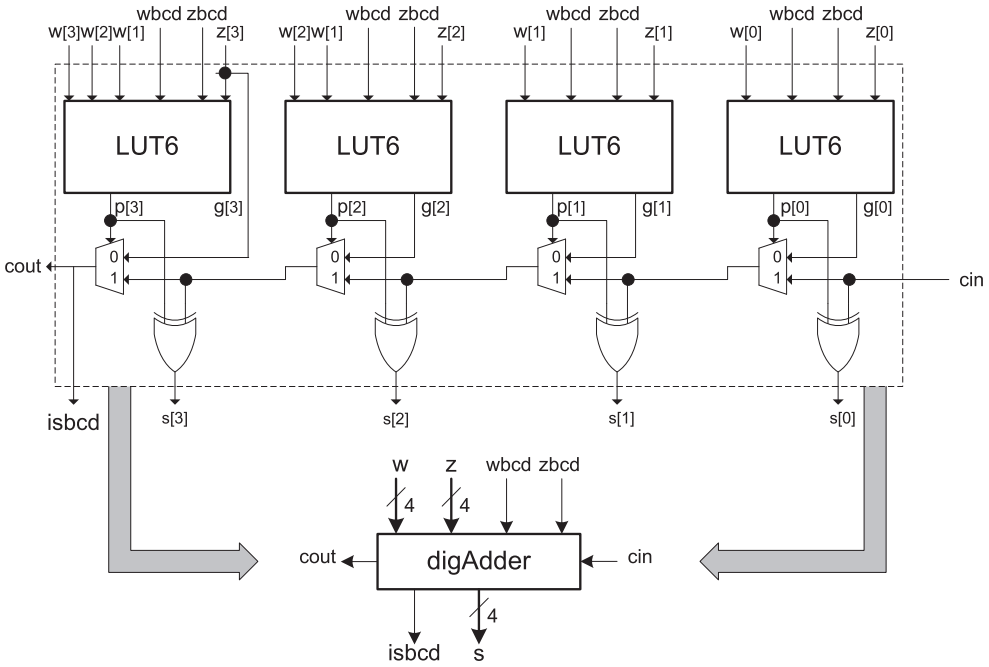

Fig. 1. Single digit BCD/excess-6 adder - digAdder.

The generate signals are always

$$g[3] = z[3]$$
$$g[2] = z[2]$$
$$g[1] = z[1]$$
$$g[0] = z[0]$$

The propagate signals are, at most, a function of six variables, and the generate signals are still one-bit functions. Therefore, all the propagate-generate pairs can be produced by one 6-input LUT, and the single digit adder can be designed using four 6-input LUTs and a carry-chain. The delay of the single digit adder is the delay of one LUT plus the delay of a four bit carry chain (see Fig. 1).

An *isbcd* output is generated to specify if the result digit is BCD (*isbcd = 1*) or excess-6 (*isbcd = 0*). The *isbcd* output is the same as the digit carry out, *cout*. The result may have to be adjusted to BCD or to excess-6 if *isbcd* is zero or one, respectively.

Given two N-digit decimal numbers

$$W = w_{N-1}w_{N-2}...w_0$$

$$Z = z_{N-1}z_{N-2}...z_0$$

whose digits may be represented in BCD or excess-6, specified with the extra inputs

$$Wbcd = wbcd_{N-1}wbcd_{N-2}...wbcd_0$$

$$Zbcd = zbcd_{N-1}zbcd_{N-2}...zbcd_0$$

the addition S = W + Z is implemented with a chain of N single digit BCD/excess-6 adders. The result is the decimal addition

$$S = s_{N-1}s_{N-2}...s_0$$

and an extra output

$$ISbcd = isbcd_{N-1}isbcd_{N-2}...isbcd_0$$

that specifies if the corresponding digit is BCD ($isbcd_i = 1$) or excess-6 ($isbcd_i = 0$) (see Fig. 2).

Each single digit BCD/excess-6 adder uses four 6-input LUTs. So, a BCD/excess-6 decimal adder for *N* digits needs $4 \times N$ 6-input LUTs and the carry chain. The delay of the adder is the delay of one LUT plus the delay of a *N* times four bit carry chain.

4. Decimal multiplier

Given a decimal multiplier operand with *n* decimal digits (a_i)

$$A = a_{n-1}a_{n-2}...a_0 = \sum_{i=0}^{n-1} a_i \times 10^i \quad (1)$$

and a decimal multiplicand operand also with *n* decimal digits (b_i)

$$B = b_{n-1}b_{n-2}...b_0 = \sum_{i=0}^{n-1} b_i \times 10^i \quad (2)$$

the multiplication $A \times B$ results in a product P with *2n* decimal digits (p_i)

$$P = \sum_{i=0}^{2n-1} p_i \times 10^i \quad (3)$$

Each decimal digit is coded with four bits whose weights depend on the code used to represent each digit. Formally, a digit $x_i \in \{0, 9\}$ is determined by

$$x_i = \sum_{j=0}^{3} x_i[j] \times w_i[j], \quad (4)$$

where $x_i[j]$ is the bit of x_i at position j and $w_i[j]$ is the weight of bit $x_i[j]$.

Several codes ($w[3]w[2]w[1]w[0]$) have been proposed and used to implement decimal multiplication. The most common were (8421 or BCD-*Binary-Coded Decimal*), (5421), (4221), (5221), (4311) and (3321). In this paper, we are particularly interested in codes (5221) and BCD (8421).

The proposed multiplier (see Fig. 3) follows the common steps of decimal multiplication: partial product generation and partial product reduction.

In this work we propose a new method for partial product generation based on the 5221 coding of the multiplier. We have decided for this coding since it only requires multiples 5X and 2X, which are easy to obtain with a single level of LUT, as will be shown below. The BCD/excess-6 adder is used both in the partial product generation and in the partial product reduction blocks.

The partial product generation circuit receives multiplicand X, multiples 2X and 5X of the multiplicand and generates N partial

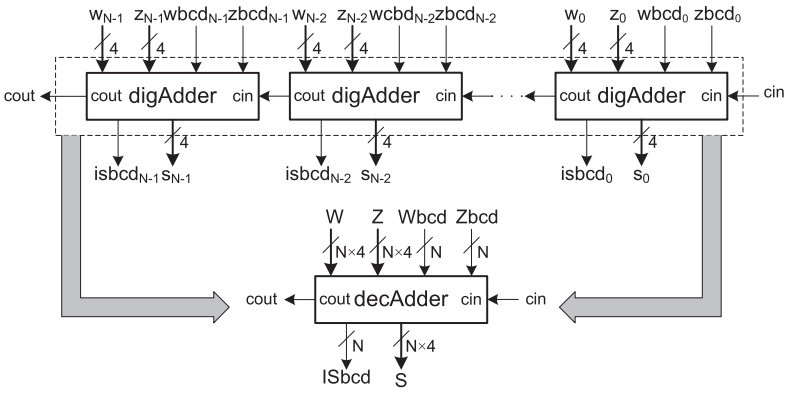

Fig. 2. BCD/excess-6 adder for N digits - decAdder.

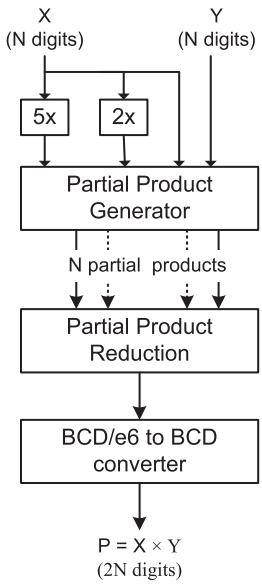

Fig. 3. Top level architecture of the proposed parallel fixed-point decimal multiplier.

products coded in BCD/excess-6. The partial product reduction circuit adds the partial products using a tree of BCD/excess-6 decimal adders and produces a result also in BCD/excess-6. A final circuit is used to convert the product to BCD.

4.1. Partial product generation

The proposed method considers a 5221 coding of multiplier digits for partial product generation. Thus, the generation of each partial product results from the sum of at most three multiples, $M0$, $M1$, $M2$, from the set {X, 2X, 5X} (see Table 3).

With this digit encoding, two decimal additions and a multiplexer are enough to generate each partial product (see the architecture of the partial product generator for a single digit multiplier in Fig. 4).

Given a multiplicand X and a digit Y_i of the multiplier, block PP1 selectively calculates one of the following sums $0 + 0$, $0 + 2X$, $0 + 5X$ or $2X + 5X$, depending on the input selectors Sel_3 and Sel_2, according to the combinations of columns M_2 and M_1 in Table 3 (multiple 5X is represented in excess-6 to simplify the implementation of block PP1, as explained in the following section).

The third multiple (column M_0 of Table 3) can be 0, X or 2X. A multiplexer is used to choose between X and 2X depending on selector Sel_1. Block PP2 then adds the result from PP1 to the multiple from the multiplexer if $Sel_0 = 0$ or to 0 if $Sel_0 = 1$.

Table 3
Values of the selectors of the partial product generator.

Y_{i5221}	Y_{iBCD} = M2 + M1 + M0	Sel_3	Sel_2	Sel_1	Sel_0
“0000”	0 = 0 + 0 + 0	0	0	‘-’	1
“0001”	1 = 0 + 0 + 1	0	0	0	0
“0010”, “0100”	2 = 0 + 2 + 0	0	1	‘-’	1
“0011”, “0101”	3 = 0 + 2 + 1	0	1	0	0
“0110”	4 = 0 + 2 + 2	0	1	1	0
“1000”, “0111”	5 = 5 + 0 + 0	1	0	’-’	1
“1001”	6 = 5 + 0 + 1	1	0	0	0
“1010”, “1100”	7 = 5 + 0 + 2	1	0	1	0
“1011”, “1101”	8 = 5 + 2 + 1	1	1	0	0
“1110”	9 = 5 + 2 + 2	1	1	1	0

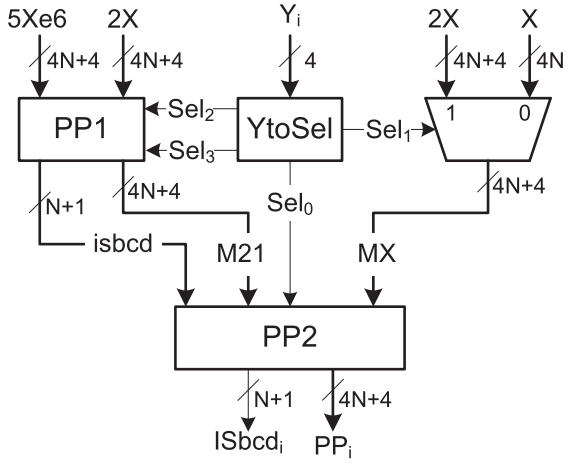

Fig. 4. Proposed Partial Product Generator for a single multiplier digit Y_i (PPGdig).

In our implementation, BCD multipliers Y_i are not explicitly converted to 5221 coding. Instead, we generate selectors Sel_3, Sel_2, Sel_1 and Sel_0 directly from the BCD multiplier. This permits to generate the selectors with only one level of logic. Selectors depend on the multiplier digit $Y_i = y[3]y[2]y[1]y[0]$ as indicated in Table 3 (right column). The logic expressions are as follows:

$$Sel_0 = \overline{y[3]}\,\overline{y[2]}\,\overline{y[0]} \vee y[2]\,\overline{y[1]}\,y[0] \quad (5a)$$

$$Sel_1 = y[2]\,\overline{y[1]} \vee y[2]\,y[0] \vee y[3]\,y[0] \quad (5b)$$

$$Sel_2 = y[3] \vee \overline{y[2]}\,y[1] \vee y[2]\,\overline{y[1]}\,\overline{y[0]} \quad (5c)$$

$$Sel_3 = y[3] \vee y[2]\,y[1] \vee y[2]\,y[0] \quad (5d)$$

The complete partial product generator produces all N partial

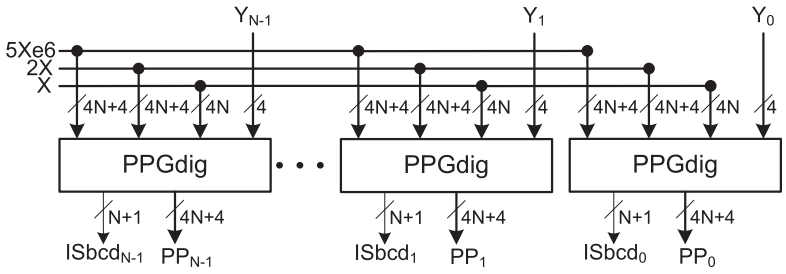

Fig. 5. Proposed partial product generator.

products in parallel using N (single digit) partial product generators (see Fig. 5).

Each *PPGdig* block receives multiples X, 2X and 5X and one digit of the multiplier. The generated partial products are in BCD/excess-6 format. Therefore, each partial product output consists of N+1 BCD/excess-6 digits, $\{PP_0, PP_1, ..., PP_{N-1}\}$, and N+1 bits, $\{ISbcd_0, ISbcd_1, ..., ISbcd_{N-1}\}$, one for each digit, indicating if the digit is represented in BCD or excess-6.

4.1.1. Implementation of block PP1

Block PP1 selectively adds multiples 5X and 2X according to Table 3. Considering the expressions of the propagate and generate signals of the BCD/excess-6 decimal adder described in Section 3, the most simplified expressions are obtained when one of the operands is represented in BCD and the other in excess-6. This will allow us to integrate the selection with the adder. Therefore, in the implementation of block PP1, multiple 5X is represented in excess-6 and 2X is represented in BCD. When one or both of the operands is 0, the value 6 (number 0 represented in excess-6) must be added if the other operand is in BCD (see Table 4).

So, block PP1 is a special case of a BCD/excess-6 decimal adder in which one of the operands is in BCD, the other is in excess-6 and the operands to be added are selected from the set {0, 6, 2*X*, 5*X*} and added according to Table 4.

Considering the equations of the BCD/excess-6 decimal adder, the assumptions in Tables 3 and 4, and multiples $5X = 5x_{n-1}5x_{n-2}\cdots5x_0$ and $2X = 2x_{n-1}2x_{n-2}\cdots2x_0$, the expressions of the propagate and generate signals for a single digit *i* of block PP1 are as follows:

$$g_i[0] = 2x_i[0]\ Sel_2 \tag{6a}$$

$$p_i[0] = (Sel_3\ 5x_i[0]) \oplus (2x_i[0]\ Sel_2) \tag{6b}$$

$$g_i[1] = 2x_i[1] \vee Sel_2 \tag{6c}$$

$$p_i[1] = (\overline{Sel_3} \vee 5x_i[1]) \oplus (2x_i[1]\ Sel_2) \tag{6d}$$

$$g_i[2] = 2x_i[2]\ Sel_2 \tag{6e}$$

$$p_i[2] = (\overline{Sel_3} \vee 5x_i[2]) \oplus (2x_i[2]\ Sel_2) \tag{6f}$$

$$g_i[3] = 2x_i[3]\ Sel_2 \tag{6g}$$

$$p_i[3] = (Sel_3\ 5x_i[3]) \oplus (2x_i[3]\ Sel_2) \tag{6h}$$

Connecting these propagate and generate signals with a carry chain provides a circuit of a single digit of block PP1 with a carry in and a carry out (see the implementation of a single digit of block PP1 in Fig. 6). The complete block PP1 is implemented as a chain of these single digit blocks.

In terms of resources occupied, a single digit of block PP1 uses four 6-input LUTs and a carry chain. Therefore, for a decimal input number with N digits, the block generates N+1 BCD/excess-6 digits using $4 \times (N + 1)$ 6-input LUTs plus the carry chain.

Table 4
Additions executed by block PP1.

Sel_3	Sel_2	excess-6 + BCD
0	0	6 + 0
0	1	6 + 2X
1	0	5X + 0
1	1	5X + 2X

4.1.2. Implementation of block PP2

Block PP2 adds the result from block PP1, *M21*, with a multiple $M0 \in \{0, X, 2X\}$. The multiple M0 to be added is selected with inputs Sel_1 and Sel_0 according to Table 3. Selecting from {*X*, 2*X*} is done using a multiplexer with selector Sel_1 (X if $Sel_1 = 0$ or 2X if $Sel_1 = 1$). The logic to select between the output of the multiplexer and 0 is integrated in block PP2 with input Sel_0 (output of multiplexer, *MX*, is selected if $Sel_0 = 0$, the value 0 is selected if $Sel_0 = 1$).

In the implementation of PP2, the output from PP1 is a decimal number where each digit is represented in BCD/excess-6 coding. To simplify the implementation of PP2, the second operand is represented in BCD, that is, X, 2X and 0 are all represented in BCD.

Hence, block PP2 is also a special case of a BCD/excess-6 decimal adder in which the digits of one of the operands are either in BCD or excess-6 (those from PP1) and the other operand is always in BCD. The value of the BCD operand may be the output of the multiplexer, *MX*, or 0, depending on selector Sel_0 (see implementation for a single digit of block PP2 in Fig. 7).

The expressions of the propagate and generate signals of digit *i* of PP2 were determined from equations of the BCD/excess-6 decimal adder with $w_i = MS_i = MX_i \cdot \overline{Sel_0}$, $z_i = M21_i$, $wbcd =' 1'$ and $zbcd = isbcd$:

$$g_i[0] = M21_i[0] \tag{7a}$$

$$p_i[0] = M21_i[0] \oplus MS_i[0] \tag{7b}$$

$$g_i[1] = M21_i[1] \tag{7c}$$

$$p_i[1] = isbcd_i \oplus MS_i[1] \oplus M21_i[1] \tag{7d}$$

$$g_i[2] = M21_i[2] \tag{7e}$$

$$p_i[2] = isbcd_i \oplus MS_i[2] \oplus (MS_i[1]\ isbcd_i) \oplus M21_i[2] \tag{7f}$$

$$g_i[3] = M21_i[3] \tag{7g}$$

$$p_i[3] = MS_i[3] \oplus ((MS_i[2] \vee MS_i[1])\ isbcd_i) \oplus M21_i[3] \tag{7h}$$

Generate signals are obtained directly from M21 without logic, while the propagate signals are at most 6-input variables. Therefore, each digit of block PP2 can be implemented with four 6-input LUTs and a carry chain. The complete block PP2 is implemented as a chain of these single digit blocks. For N-digit inputs, the block generates N BCD/excess-6 digits with $4 \times N$ 6-input LUTs.

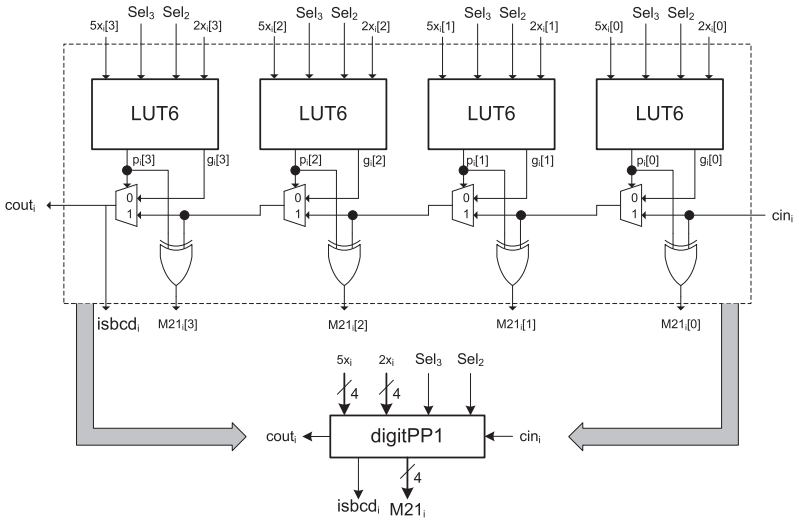

Fig. 6. Implementation for a single digit of block PP1.

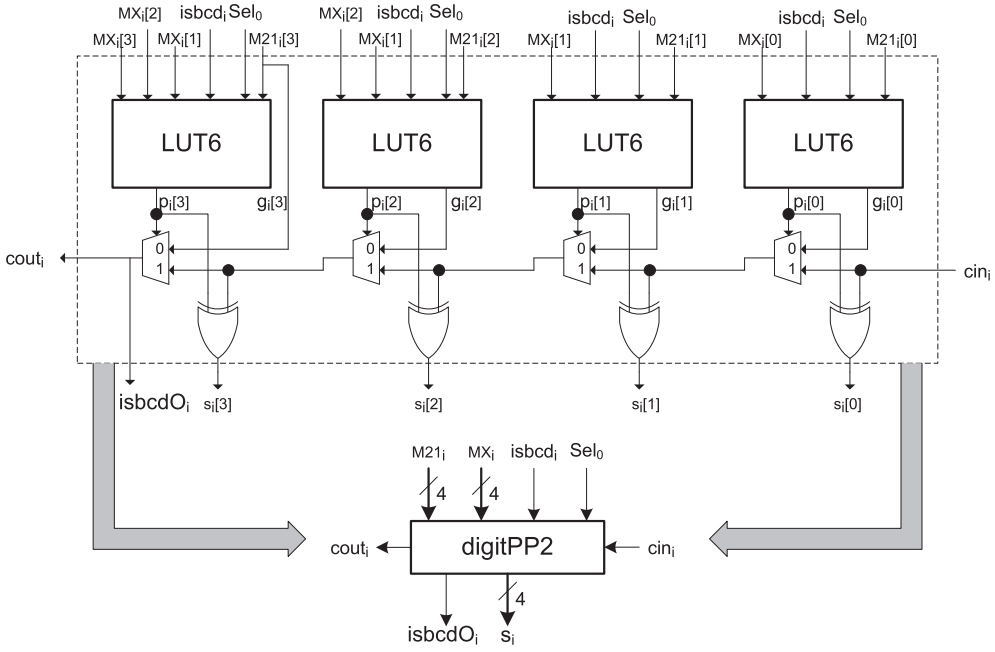

Fig. 7. Implementation of a single digit of block PP2.

4.1.3. Generation of multiples 2X and 5Xe6

The generation of partial products is based on the availability of multiples 5X in excess-6 and 2X in BCD of the multiplicand. Considering a digit $x = x[3]x[2]x[1]x[0] = x[3-0]$, multiples

Table 5
Multiplication by 2 and 5.

N	$x[3-0]$	2N	$y[4]$	$y[3-0]$	5N	$y[6-4]$	$y[3-0]$
0	0000	0	0	0000	0	000	0000
1	0001	2	0	0010	5	000	0101
2	0010	4	0	0100	10	001	0000
3	0011	6	0	0110	15	001	0101
4	0100	8	0	1000	20	010	0000
5	0101	10	1	0000	25	010	0101
6	0110	12	1	0010	30	011	0000
7	0111	14	1	0100	35	011	0101
8	1000	16	1	0110	40	100	0000
9	1001	18	1	1000	45	100	0101

$$2x = y = y[4]y[3]y[2]y[1]y[0] = y[4]y[3-0]$$

$$5x = y = y[6]y[5]y[4]y[3]y[2]y[1]y[0] = y[6-4]y[3-0]$$

are obtained according to Table 5.

Generically, digit y_i of Y = 2X is given by

$$\begin{aligned}
y_i[3] &= x_i[3]\,x_i[0] \vee x_i[2]\,\overline{x}_i[1]\,\overline{x}_i[0] \\
y_i[2] &= \overline{x}_i[2]\,x_i[1] \vee x_i[1]\,x_i[0] \vee x_i[3]\,\overline{x}_i[0] \\
y_i[1] &= x_i[3]\,\overline{x}_i[0] \vee \overline{x}_i[3]\,\overline{x}_i[2]\,x_i[0] \vee x_i[2]\,x_i[1]\,\overline{x}_i[0] \\
y_i[0] &= x_{i-1}[3] \vee x_{i-1}[2]\,x_{i-1}[1] \vee x_{i-1}[2]\,x_{i-1}[0],\ i > 0 \\
y_0[0] &= 0
\end{aligned}$$

Each of the y_i functions can be implemented with a single 4-input LUT and two of these functions can be grouped and implemented with one 6-input LUT. Therefore, the generation of a multiple 2X with N digits requires $2N$ LUT6.

Each digit y_ie6 of Y = 5Xe6 is generated directly as an excess-6 no. as follows

$$
\begin{aligned}
y_i e6[3] &= x_i[0] \vee x_{(i-1)}[3] \vee x_{(i-1)}[2] \\
y_i e6[2] &= x_i[0]\, x_{(i-1)}[3] \vee x_i[0]\, x_{(i-1)}[2] \\
&\quad \vee \overline{x}_i[0]\, \overline{x}_{(i-1)}[3]\, \overline{x}_{(i-1)}[2] \vee x_i[0]\, x_{(i-1)}[1] \\
y_i e6[1] &= \overline{x}_i[0]\, \overline{x}_{(i-1)}[2] \vee \overline{x}_{(i-1)}[2]\, \overline{x}_{(i-1)}[1] \\
&\quad \vee x_i[0]\, x_{(i-1)}[2]\, x_{(i-1)}[1] \\
y_i e6[0] &= x_i[0] \oplus x_{(i-1)}[1] \\
y_0 e6[0] &= y_0 e6[3] = x_0[0] \\
y_0 e6[1] &= 1 \\
y_0 e6[2] &= \overline{x[0]}
\end{aligned}
$$

Each of the $y_i e6$ functions can be implemented with a single 4-input LUT and two of these functions can also be grouped and implemented with one 6-input LUT. Since the least significant digit consumes a single LUT to generate $y_0 e6[2]$, the implementation of a multiple 5Xe6 with N digits requires $2N + 1$ LUT6.

4.2. Partial product reduction

Given a set of N partial products $\{PP_0, PP_1, ..., PP_{N-1}\}$, partial product reduction consists on calculating the multioperand addition of all partial products PP_i decimally left shifted by i decimal places. For the proposed method, the sum of the N partial products, that is, the final decimal product, P, is calculated as

$$P = \sum_{i=0}^{N-1} PP_i \times 10^i$$

The multioperand addition of the proposed decimal multiplier is designed using an adder tree. The tree requires $L = log_2 N$ levels of adders, from the first level 0 down to level L-1, with $\frac{N}{2^{i+1}}$ adders at level i.

Considering N operands of size N + 1 to be added, the first level uses N/2 adders of size (N + 1). At this level, each operand PA_i of the N/2 operands, $\{PA_0, PA_1, ..., PA_{N/2-1}\}$, are decimally shifted $2 \times i$ positions and added in pairs using N/4 adders of size N + 2, and so on. The last level uses a single adder of size 3N/2. For example, in the particular case of a 4 × 4 decimal multiplier (see Fig. 8) four partial products, PP_0, PP_1, PP_2 and PP_3, have to be added. The first level uses two adders of size 5 and the last level uses a single adder of size 6.

The complete partial product reduction tree for N operands of size N + 1 uses $\frac{N}{2} \times log_2(N) + N^2 - N$ BCD/excess-6 single digit adders (*digAdder*). (see Fig. 9).

The critical path of an adder tree with N partials is given by $log_2 N$ digit adders plus $4 \times 2N$ carry chain bits. Therefore, the delay of the critical path is the sum $log_2(N) \times LUT6_{delay} + 2N \times CC_{delay}$, where $LUT6_{delay}$ is the delay one LUT6 and CC_{delay} is the carry chain delay of a digit (4 bits).

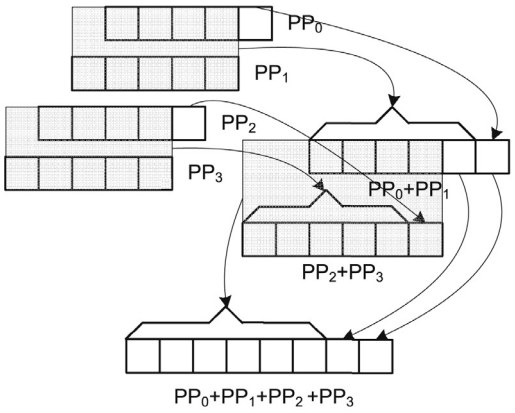

Fig. 8. Tree adder design for a 4 × 4 decimal multiplier.

4.3. BCD/Excess-6 to BCD converter

The final sum is the product of the decimal multiplication in BCD/excess-6 format. The result is finally converted to BCD to obtain the decimal product. According to the design of the conditional decimal adder, if *isbcd* is 0, the digit must be converted from excess-6 to BCD. Otherwise, the digit is already in BCD format. The expressions to convert a BCD/excess-6 digit, $d = d[3]d[2]d[1]d[0]$ to a BCD digit, $d_{bcd} = d_{bcd}[3]d_{bcd}[2]d_{bcd}[1]d_{bcd}[0]$ are as follows:

$$d_{bcd}[0] = d[0] \tag{8a}$$

$$d_{bcd}[1] = d[1] \oplus \overline{isbcd} \tag{8b}$$

$$d_{bcd}[2] = (d[2] \oplus d[1])\, \overline{isbcd} \vee d[2]\, isbcd \tag{8c}$$

$$d_{bcd}[3] = d[3]\, d[2]\, d[1]\, \overline{isbcd} \vee d[3]\, isbcd \tag{8d}$$

Given these expressions, each digit can be converted with just two 6-input LUTs. For N-digit operands, we have a final product with 2N digits and, consequently, a total of 4N 6-input LUTs. In this case, the delay is that of a single LUT6.

4.4. Characterization of the decimal multiplier

For an $N \times N$ decimal multiplier, we present in Table 6 the theoretical area occupation and delay of each block: multiples generator (MG), recode, partial product generator (PPG), partial product reduction (PPR) and converter (see Fig. 3, for the multiplier structure) and of the complete multiplier.

As already indicated, $LUT6_{delay}$ is the delay of one LUT6 and CC_{delay} is the carry chain delay of a digit (4 bits).

The multiples generator includes the generation of multiples 2X and 5Xe6. The recode is relative to the logic to generate the selectors from the multiplier digits. Both multiples generator and recode run in parallel. Converter is the logic for the final conversion from BCD/excess-6 to BCD.

As shown, the area cost of the partial product generator and reduction blocks increases with N^2, while the delay increases with N.

5. Results

All designs were described in VHDL and implemented targeting a Virtex-6 FPGA (−3 and −2 speed grade). We decided not to use a more recent FPGA to guarantee a fare comparison with other architectures. The area of the circuit in a more recent FPGA is the same, only the performance is better. The architecture was simulated, synthesized, placed and routed using ISE14.7 from Xilinx. Our results were compared with state-of-the-art decimal multipliers implemented in 6-input LUTs FPGA [37–39]. The decimal multiplier based on Karatsuba's algorithm [38] and that from [37] were implemented in a Virtex-6 speed grade -2 and -3, respectively. The work [39] is a pipelined implementation in a Virtex-6 device with speed grade −2. Therefore, we have considered the delay mentioned in the paper that results by multiplying the number of pipeline stages by the delay of the stage subtracted by an average delay (0.5 ns) of the flip-flop used in the pipeline stage. Also, to account for the difference in the speed grade from −2 to −3, we provide results of our designs for both −2 and −3 speed grades and compare with other works according to their speed grades.

The work [37] includes comparisons with other FPGA implementations [15,31,36] and have shown better results. Hence, these other works are not included in our comparison.

All designs consider registered inputs and outputs. The multiplier in [37] uses the final flip-flops to implement the final conversion from the internal representation to BCD. To register the outputs an extra level of LUTs is required. In our circuit, the extra level of LUTs implements both the converter and the register. So, for a fare comparison, we considered

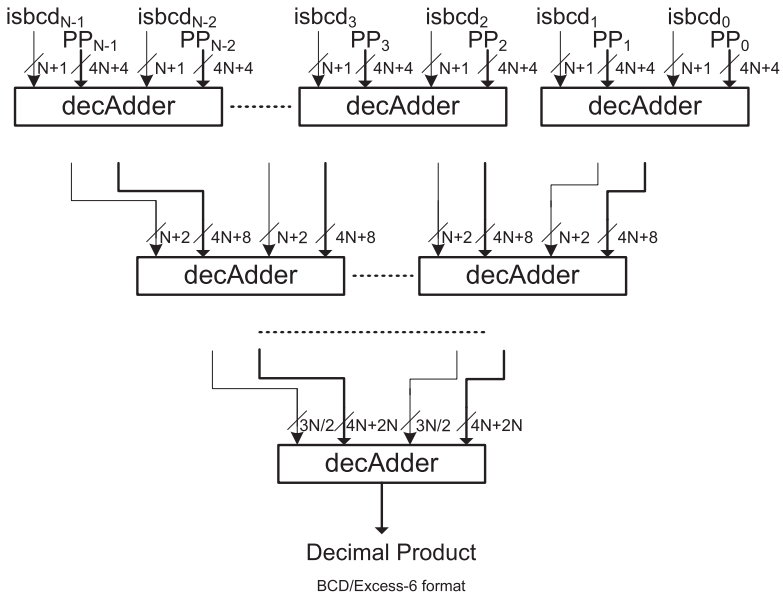

Fig. 9. Tree adder design of the decimal multiplier using BCD/excess-6 decimal adders.

two sets of implementations of our multipliers, one including the final BCD/excess-6 to BCD converter (proposed w/ conv) and the other without this converter (proposed w/o conv) The area and the delay of all implementations after P&R are presented in Tables 7 and 8.

The theoretical area was omitted in Table 8 since the figures are the same of Table 7.

The Karatsuba based decimal multiplier in [38] is worse than this new proposal, both in area and delay, except for the larger 32 × 32 multiplier, in which case the area is about 12% smaller but at the cost of around 1.6 × higher delay. This increase in delay is because the Karatsuba based decimal multiplier needs some extra additions that introduces extra delay. Also, the algorithm implementation is more irregular and so the more complex routing also degrades the propagation delay.

While the other solutions are worse in terms of area compared to the Karatsuba method (except for the smallest cases, 2 × 2 and 4 × 4), the proposal herein is able to achieve better results for multipliers of size 8 × 8 and 16 × 16.

Since the work in [38] uses the multiplier from [37] and the decimal multipliers proposed herein are smaller, using them in the proposed Karatsuba decimal multiplier could contribute to further reduce the area of that solution.

The proposed multiplier is smaller than the multiplier from [37] up to 20% (for the 32 × 32 multiplier). This is due to the reduction in the area of the adders (our proposed adder uses four 6-input LUTs for each digit instead of the five 6-input LUTs of the decimal adder proposed in [37]) and of the improved partial product generator. In terms of delay, the solutions are very close to each other, with a very slight advantage for our proposal considering the registered case. The multiplier from [37] has the same number of LUT6 in the critical path if we do not consider the final converter and one less otherwise. So, in terms of propagation delay, both circuits are very similar if the final converter is not considered and slightly worst with the converter.

With a non-registered output, the multiplier in [37] is faster, since the converter is implemented with the flip-flops without requiring an extra LUT level. Both architectures have the same number of logic levels, but the multiplier in [37] is able to implement the logic for the final conversion from the internal representation to BCD using the FF present in the LUTs, while our final converter needs an extra level of LUTs. However, if the design has to be registered, as is usually required, the multiplier in [37] needs an extra level of LUTs to implement the register, while the proposed design can simply use the FF present (still available) in the LUTs.

Compared to [39], the proposed multiplier is considerably better in terms of area and delay, with less 43% in area and less 37% in delay for the 16 × 16 multiplier.

To see the influence of the decimal adder in the area reduction of the proposed multiplier compared to the previous best multiplier [37], we determined the area of the multiplier from [37] using the BCD/excess-6 adder in the partial product reduction circuit (see Table 9). In this case, we need also to include the final BCD/excess-6 to BCD converter.

In this case, as expected, the area improvement is smaller, but still from 5% up to 12% instead of 16% to 22%. From this figures, we see the area improvement achieved with the new decimal multiplier due to the new architecture of the product generator and the remaining due to the new decimal adder.

Therefore, the proposed method is better both in area and delay

Table 6
Theoretical area and delay of decimal multiplier.

Block	Area cost in LUT6	Critical path
MG	4N	$1 \times LUT6_{delay}$
Recode	2N	$1 \times LUT6_{delay}$
PPG	$10N^2 + 9N$	$2 \times LUT6_{delay} + (N+1) \times CC_{delay}$
PPR	$4 \times \left(\frac{N}{2} \times \lceil log_2 N \rceil + N^2 - N\right)$	$log_2(N) \times LUT6_{delay} + 2N \times CC_{delay}$
Converter	4N	$1 \times LUT6_{delay}$
BCD Multiplier	$14N^2 + 15N + 2N\lceil log_2 N \rceil$	$(4 + log_2(N)) \times LUT6_{delay} + 2N \times CC_{delay}$

Table 7
Logic area (LUTs) and delay (ns) - LUTs(delay) - for each decimal multiplier for different number of digits and speed grade -3.

	Theory	Proposed w/ conv		Proposed w/o conv		[37]	
Size	Area	Area	Delay	Area	Delay	Area	Delay
2 × 2	90	91	4.2	83	3.6	—	—
4 × 4	300	301	5.4	285	4.8	336	4.9
8 × 8	1064	1065	7.0	1033	6.3	1268	6.4
16 × 16	3952	3954	8.9	3890	8.2	4880	8.4
32 × 32	15,136	15,146	14.0	15,018	13.2	—	—
34 × 34	17,102	17,135	14.9	16,999	14.1	21,611	14.4

Table 8
Logic area (LUTs) and delay (ns) - LUTs(delay) - for each decimal multiplier for different number of digits and speed grade -2.

	Proposed w/ conv		Proposed w/o conv		[38]		[39]	
Size	Area	Delay	Area	Delay	Area	Delay	Area	Delay
4 × 4	301	6.3	285	5.6	365	7.5	450	7.0
8 × 8	1065	8.1	1033	7.4	1197	10.4	1850	11.3
16 × 16	3954	10.3	3890	9.5	4088	16.0	6843	16.5
32 × 32	15,146	15.6	15,018	14.7	13,257	25.0	—	—

Table 9
Logic area (LUTs) of the decimal multiplier from [37] using the BCD/excess-6 adder for different number of digits.

Size	Proposed #2	[37]
4 × 4	301	316
8 × 8	1065	1160
16 × 16	3954	4400
32 × 32	15,146	17,056
34 × 34	17,135	19,333

than the best parallel decimal multipliers on 6-input LUT FPGAs. Therefore, we can conclude that the proposed methods represent important innovations in the design of decimal multipliers.

A detailed analysis of the scalability of the methods, reveals some further advantages of the proposed multiplier. Considering the relative area of the compared decimal multipliers (see Fig. 10), the multiplier from [37] has a faster area increase with the size of the multiplier. Only the Karatsuba based decimal multiplier in [38] considerably improves with the size of the multiplier, in accordance with the less than quadratic complexity of the Karatsuba-Ofman's algorithm. The multiplier presented in [39] has a slight improvement in the 16 × 16 multiplier compared to ours but still around 40% worst.

In terms of delay (see Fig. 11), the work [37] has the same scalability, since it increases at the same rate than ours, the delay of the multiplier in [39] increases much faster with the size of the operands. Only the Karatsuba multiplier keeps the delay rate from a 16 × 16 to 32 × 32 multiplier. This is due to the fact that with large operands the relative overhead associated with the extra adders decreases.

6. Conclusions and future work

We have proposed a new architecture for parallel decimal fixed-point multiplication on 6-input LUT FPGAs. The method is based on a 5221 coding of the multiplier to improve the efficiency of the partial product generation, and integrates and adapts the novel decimal adder that sums operands represented in BCD and/or excess-6 using only 4 LUTs per digit. The structure of this new BCD/excess-6 adder was adopted to efficiently implement the partial product generators and the final adder tree.

The results were compared with the best state-of-art implementations of a parallel decimal multiplier. From the results we conclude that compared to the best multiplier in terms of delay targeting FPGA the new method has better areas with slightly worse delays for an implementation with non-register outputs. In case, a registered output design is needed then our proposal has also better delays. In terms of

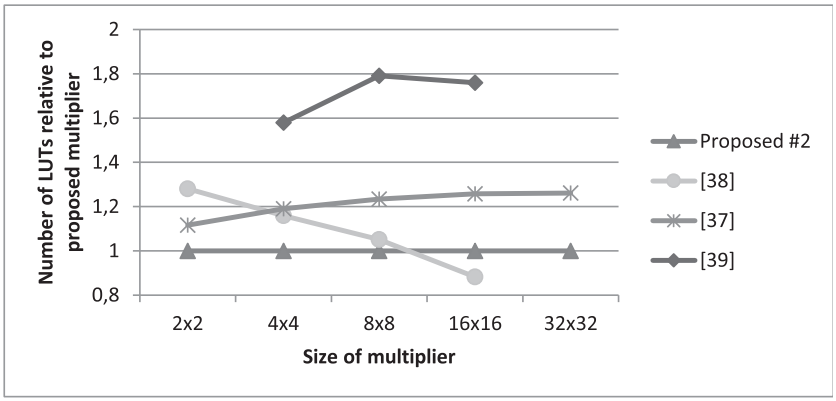

Fig. 10. Relative area between the proposed multiplier and previous multipliers.

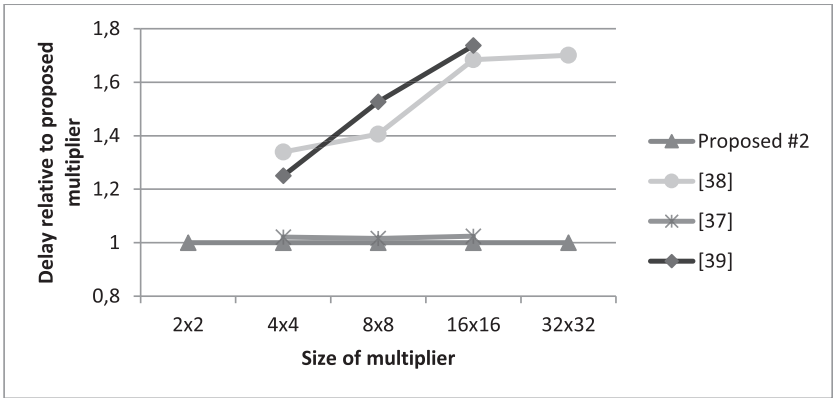

Fig. 11. Relative delay between our multiplier and previous multipliers.

area, our new proposal is always better except for a 32 × 32 multiplier in which case the Karatsuba's based multiplier has a smaller area but a delay which is almost 1.6 × larger.

The proposed multiplier architecture is quite efficiently implemented in 6-input LUT FPGAs. However, the proposed partial product generation method clearly reduces the design complexity and will most probably reduce the area of the circuit even in ASIC implementations. Therefore, we are planning to implement and evaluate the proposed multiplier also in ASIC.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEA-ELC/122098/2010.

References

[1] H.C. Neto, M.P. Véstias, Decimal addition on FPGA based on a mixed BCD/excess-6 representation, Microprocess. Microsyst. 55 (2017) 91–99.
[2] M.F. Cowlishaw, Decimal floating-point: Algorism for computers, Proceedings of the Sixth IEEE International Symposium on Computer Arithmetic, (2003), pp. 104–111.
[3] M. CORNEA, J. CRAWFORD, IEEE 754r decimal floating-point arithmetic: reliable and efficient implementation for intel architecture platforms, Intel Technol. J. 11 (2007) 91–94.
[4] F. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz, S.R. Carlough, The IBM z900 decimal arithmetic unit, Proceedings of the Asilomar Conference on Signals, Systems, Computers, (2001), pp. 1335–1339.
[5] IBM Power6, 2007, http://www2.hursley.ibm.com/decimal/.
[6] C.F. Webb, IBM Z10: the next- generation mainframe microprocessor, IEEE Micro 28 (2) (2008) 19–29.
[7] IEEE Standards Committee, 754–2008 IEEE standard for floating-point arithmetic, (2008), pp. 1–58. http://ieeexplore.IEEE.org/servlet/opac?punumber=4610933
[8] M. Cornea, C. Anderson, J. Harrison, P. Tang, E. Schneider, S. Tsen, A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format, Proceedings of the IEEE Eighteenth Symposium on Computer Arithmetic, (2007), pp. 29–37.
[9] ANSI C decnumber library v3.68, 2012, http://speleotrove.com/decimal/decnumber.html.
[10] GNU C compiler library. http://gcc.gnu.org/onlinedocs/gcc/Decimal-Float.html.
[11] L.-K. Wang, et al., Benchmarks and performance analysis of decimal floating-point applications, Proceedings of the Twenty-Fifth International Conference on Computer Design, (2007), pp. 164–170.
[12] T. Ohtsuki, et al., Apparatus for decimal multiplication, U.S. Patent 4677583, June, 1987.
[13] R.D. Kenney, M.J. Schulte, M.A. Erle, High-frequency decimal multiplier, Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, (2004), pp. 26–29.
[14] M.A. Erle, M.J. Schulte, Decimal multiplication via carry-save addition, Proceedings of the Fourteenth IEEE International Conference on Application Specific Systems, (2003), pp. 348–358.
[15] G. Sutter, E. Todorovich, G. Bioul, M. Vázquez, J.-P. Deschamps, FPGA implementations of BCD multipliers, Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs, (2009), pp. 36–41.
[16] T. Lang, A. Nannarelli, A radix-10 combinational multiplier, Proceedings of the IEEE Forty International Asilomar Conference on Signals, Systems, and Computers, (2006), pp. 313–317.
[17] A. Vázquez, E. Antelo, P. Montushi, A new family of high-performance parallel decimal multipliers, Proceedings IEEE 18th Symposium on Computer Arithmetic, (2007), pp. 195–204.
[18] L. Dadda, A. Nannarelli, A variant of a radix-10 combinational multiplier, Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), (2008), pp. 3370–3373.
[19] R.H. Larson, High-speed multiply using four input carry-save adder, IBM Tech. Discl. Bull 16 (7) (1973) 2053–2054.
[20] T. Ueda, Decimal multiplying assembly and multiply module, U.S. Patent 5379245, January, 1995.
[21] M.A. Erle, E.M. Schwarz, M.J. Schulte, Decimal multiplication with efficient partial product generation, Proceedings of the Seventeenth IEEE Symposium on Computer Arithmetic, (2005), pp. 21–28.
[22] F. Busaba, T. Slegel, S. Carlough, C. Krygowski, J. Rell, The design of the fixed point unit for the z990 microprocessor, Proceedings of the Fourteenth ACM Great Lakes Symposium, (2004), pp. 364–367.
[23] A. Vázquez, E. Antelo, P. Montuschi, Improved design of high-performance parallel decimal multipliers, IEEE Trans. Comput. 59 (5) (2010) 679–693.
[24] S. Gorgin, G. Jaberipur, A fully redundant decimal adder and its application in parallel decimal multipliers, Microelectron. J. 40 (10) (2009).
[25] L. Han, S.-B. Ko, High-speed parallel decimal multiplication with redundant internal encodings, IEEE Trans. Comput. 62 (5) (2013) 956–968.
[26] M. Schmookler, A. Weinberger, High speed decimal addition, IEEE Trans. Comput. C-20 (8) (1971) 862–866.
[27] B. Shirazi, D. Yun, C. Zhang, RBCD: redundant binary coded decimal adder, IEE Proc. 136 (2) (1989).
[28] R.D. Kenney, M.J. Schulte, High speed multioperand decimal adders, IEEE Trans. Comput. 54 (8) (2005) 953–963.
[29] L. Dadda, Multioperand parallel decimal adder: a mixed binary and BCD approach, IEEE Trans. Comput. 56 (10) (2007) 1320–1328.
[30] H. Neto, M. Véstias, Decimal multiplier on FPGA using embedded binary multipliers, Proceedings of the IEEE Field Programmable Logic and Applications, (2008), pp. 197–202.
[31] M. Véstias, H. Neto, Parallel decimal multipliers using binary multipliers, Proceedings of the IEEE Sixth Southern Programmable Logic Conference, (2010), pp. 73–78.
[32] M. Fazlali, H. Valikhani, S. Timarchi, H.T. Malazi, Fast architecture for decimal digit multiplication, Microprocess. Microsyst. 39 (4) (2015) 296–301, http://dx.doi.org/10.1016/j.micpro.2015.01.004.
[33] O. Al-Khaleel, Z. Al-Qudah, M. Al-Khaleel, C. Papachristou, High performance FPGA-based decimal-to-binary conversion schemes for decimal arithmetic, Microprocess. Microsyst. 37 (3) (2013) 287–298, http://dx.doi.org/10.1016/j.micpro.2013.01.002.
[34] G. Jaberipur, A. Kaivani, Binary-coded decimal digit multipliers, IET Comput. Digit. Tech. 1 (4) (2007) 377–381.
[35] G. Bioul, M. Vázquez, J.-P. Deschamps, G. Sutter, Decimal addition in FPGA, Proceedings of the V Southern Programmable Logic Conference, (2009), pp. 101–108.
[36] M. Baesler, T. Teufel, FPGA implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier, Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs, (2009), pp. 6–11.
[37] A. Vázquez, F. de Dinechin, Efficient implementation of parallel BCD multiplication in LUT-6 FPGAs, Proceedings of the International Conference on Field-Programmable Technology (FPT), (2010), pp. 126–133.
[38] M. Véstias, H. Neto, Parallel decimal multipliers and squarers using Karatsuba–Ofman's algorithm, Proceedings of the Fifteenth Euromicro Conference on Digital System Design, (2012), pp. 782–788.
[39] S. Gao, D. Al-Khalili, J. Langlois, N. Chabini, Efficient realization of BCD multipliers using FPGAs, Int. J. Reconfig. Comput. (2017) 12.

Mário Véstias received the Ph.D. in electrical and computer engineering in 2002, both from the Technical University of Lisbon, Portugal. He is a Coordinate Professor at the Polytechnic Institute of Lisbon, School of Engineering (ISEL), Department of Electronic, Telecommunications and Computer Engineering (DEETC), where he is responsible for undergraduate and graduate courses on computer architecture and digital systems design. He is also a senior researcher at the ESDA (Electronic Systems Design and Automation) group at the research institute INESC-ID in Lisbon. His main current research interests are Computer Architectures and Digital Systems for Embedded Reconfigurable Computing.

Horácio C. Neto is an Associated Professor at the University of Lisbon, School of Engineering (IST), Department of Electrical and Computer Engineering (DEEC). He is responsible for the Electronic Systems Design and Automation (ESDA) research group at INESC-ID, a research institute associated with the Engineering University, IST. His main research interests are Digital Systems Design and Computer Architecture, with emphasis in Reconfigurable Computing. He has a Ph.D. in Electrical and Computer Engineering from the Technical University of Lisbon.