Efficient implementation of Parallel BCD Multiplication in LUT-6 FPGAs

Álvaro Vázquez [1], Florent de Dinechin [2]

LIP, Projet Arénaire, INRIA/ENS de Lyon/CNRS/UCBL/Université de Lyon
46, allée d'Italie, 69364 Lyon Cedex 07, France
[1] `Alvaro.Vazquez-Alvarez@inria.fr`
[2] `Florent.de.Dinechin@ens-lyon.fr`

***Abstract*—Decimal multiplication is one of the most frequent operations used by many financial, business and user-oriented applications but current implementations in FPGAs are very inefficient in terms of both area and latency when compared to binary multipliers. In this paper we present a new method for implementing BCD multiplication more efficiently than previous proposals in current FPGA devices with 6-input LUTs. In particular, a combinational implementation maps quite well into the slice structure of the Xilinx Virtex-5/Virtex-6 families and it is highly pipelineable. The synthesis results for a Virtex-6 device indicate that our proposal outperforms the area and latency figures of previous implementations in FPGAs.**

I. Introduction

Decimal arithmetic is pervasive in human-oriented applications but has a limited use in numerical data processing. Computer arithmetic is mainly binary due to its better numerical properties and a simpler implementation in two-state digital systems. Very recently, a renewed interest on efficient decimal arithmetic implementations has emerged due to an increasing demand for financial and business computing [1]. Motivated by this prospect, an intense research and development on decimal hardware algorithms and architectures is being carried out. Thus, some processor developers are gradually incorporating hardware support for decimal floating-point arithmetic (DFP) into their high-end products [2], [3]. Besides, a specification of decimal arithmetic has been incorporated to the revision of the IEEE 754 Standard for Floating-Point Arithmetic (IEEE 754-2008) [4].

An important and frequent operation in many applications is decimal multiplication. It is complex to implement in hardware due to the larger range of decimal digits ($[0, 9]$) and the inefficiency of binary codes to represent decimal values, so that decimal multipliers have lower performance and larger area than comparable binary multipliers. For example, because of the high-area requirements of a pipelined parallel implementation, decimal multiplication in IBM Power6 and Z/system high-end processors [2], [3] is performed serially using a BCD (Binary Coded Decimal) carry-propagate adder and hardware assists. Several decimal serial and parallel multipliers have been proposed for ASIC [5], [6], [7], [8], [9], [10], [11], [12], [13] and FPGA [14], [15], [16], [17], [18], [19] platforms. FPGA implementations are generally based on techniques originally developed for VLSI architectures. The special built-in characteristics of FPGA architectures [20] make it difficult to use many well-known methods to speedup computations (for example, carry-save and signed-digit arithmetics). Therefore, beyond adapting existing techniques we explore new decimal multiplication algorithms more suitable for FPGAs.

This paper presents the algorithm, architecture and FPGA implementation of a novel unit to perform fast decimal fixed-point or integer multiplication. We have designed a decimal parallel multiplier which results in an area-efficient implementation on 6-input LUT FPGAs. Although sequential and online implementations consume low area, they have long latencies that make them inadequate for high-performance applications. Parallel implementations have been widely used to speedup the binary multiplication in high-performance processors by means of deeper pipelines. Following this line, our design is fully pipelined.

The structure of the paper is as follows. In Section II we present an outline of prior work. In Section III we propose a new method for BCD multiplication. Section IV introduces the resultant combinational and pipelined architectures and presents the synthesis results of an implementation on a Virtex-6 FPGA [20]. A comparison with other recent FPGA implementations is shown in Section V. Finally, the conclusions are summarized in Section VI.

II. Survey of BCD multiplication algorithms for FPGA

The IEEE 754-2008 Standard [4] specifies three basic formats (Decimal32, Decimal64, and Decimal128) and two different encodings, namely BID (Binary Integer Decimal) and DPD (Densely Packed Decimal), for representing decimal floating-point numbers. The value of a finite decimal floating-point number F_X in the Standard is given by

$$F_X = (-1)^{s_x} \cdot X \cdot 10^{E_X - bias} \tag{1}$$

where s_x is a sign bit, X is an integer coefficient, and E_X is a biased positive exponent. In the BID encoding the coefficient X is coded as an unsigned binary integer, so it needs to be converted to decimal before being displayed to the user. This encoding is preferred for software implementations of decimal floating-point arithmetic [21]. On the other hand, the DPD encoding is more oriented to hardware implementations, and uses BCD to represent the coefficient as a p-digit decimal

978-1-4244-8982-4/10/$26.00 ©2010 IEEE

integer $X = \sum_{i=0}^{p-1} X_i \, 10^i$ ($p \in \{7, 16, 34\}$ for the three basic formats). Each digit X_i is coded in BCD as

$$X_i = \sum_{j=0}^{3} x_{i,j} \cdot 2^j \qquad (2)$$

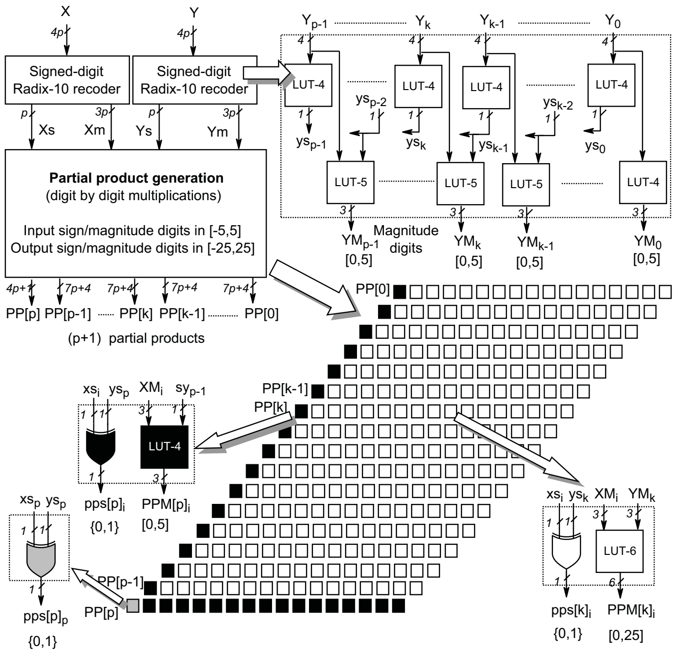
where $X_i \subset [0,9]$ is the i^{th} decimal digit and $x_{i,j} \in \{0,1\}$ is the j^{th} bit of the 4-bit BCD digit i.

In order to obtain an area-efficient FPGA implementation of parallel decimal multiplication, we have first performed a survey of the most representative BCD multiplication algorithms. A straightforward method to implement a BCD multiplication $P = X \times Y$ in hardware is to use an existing binary multiplier and conversions from BCD to binary and back to BCD. Although these conversions are costly in terms of latency and hardware resources, this solution was considered by Vestias and Neto [19] for FPGAs with embedded binary multipliers. To reduce the hardware cost, large multiplications are subdivided in several 4×4-digit decimal multiplications, each one implemented with two BCD to binary converters, an embedded DSP block, and a binary to BCD converter.

These conversions can be avoided if the BCD operands are multiplied directly. This typically involves two steps: generation and reduction of decimal partial products. These steps can be implemented serially [6], [7], [9], [11] or in a parallel fashion [5], [8], [10], [12], [13]. In serial implementations one partial product is generated at each iteration, multiplying the multiplicand by one digit of the multiplier, and accumulated to the intermediate partial product. On the other hand, in parallel implementations all partial products are generated simultaneously and reduced in an array or tree structure of adders.

Several FPGA implementations of BCD multipliers [14], [15], [17], [18] adapt some techniques for decimal partial product generation and reduction originally proposed for ASIC. Sutter et al. [18] present combinational and sequential implementations of BCD multipliers on a Xilinx Virtex-4 device. In this proposal, BCD input operands are multiplied digit by digit in a binary way. Each 7-bit product is corrected separately as proposed in [11] to form a two-digit BCD product. These two-digit products are implemented in a Virtex-4 FPGA using LUTs configured as multiplexers or 18-Kbit RAM blocks. Pairs of decimal partial products are generated in this way multiplying each BCD digit of the multiplier by the multiplicand. All the partial products are then reduced using a tree configuration of a specially designed fast BCD carry-chain adder [22]. An extension of this architecture to decimal floating-point multiplication was presented in [17].

A sequential BCD multiplier proposed in [6] for ASIC, has been recently implemented by James et al. [15] on a variety of FPGAs. In this approach, a set of simple BCD multiplicand multiples ($\{0X, 1X, 2X, 4X, 5X\}$) are precomputed. Each decimal partial product is generated sequentially by adding two elements of this set, which are selected according to the value of the corresponding BCD multiplier digit ($[0,9]$). The decimal partial products are accumulated to an intermediary



X
Y
Signed-digit Radix-10 recoder
Signed-digit Radix-10 recoder
Xs
Xm
Ys
Ym
Partial product generation (digit by digit multiplications)
Input sign/magnitude digits in [-5,5]
Output sign/magnitude digits in [-25,25]
PP[p] PP[p-1] PP[k] PP[k-1] PP[0]
(p+1) partial products
LUT-4
LUT-5
LUT-6
Magnitude digits
YM[0,5]
pps[p]i {0,1}
PPM[p]i [0,5]
pps[k]i {0,1}
PPM[k]i [0,25]
pps[p]p {0,1}

Figure 1. Signed-digit radix-10 recoding architecture

BCD partial product using a decimal carry-save 4:2 adder. This adder can be build using two rows of 4-bit BCD carry-propagate adders.

Baesler and Teufel [14] implement a parallel BCD multiplier on a Xilinx Virtex-II FPGA. This multiplier is based on the methods proposed in [8], [12], [13], namely, signed-digit radix-10 recoding of the BCD multiplier operand and pre-computation of decimal multiples $\{-5X, \ldots, -1X, 0X, 1X, \ldots, 5X\}$ for partial product generation, and decimal partial product reduction by means of a special BCD-4221 carry-save adder tree.

McIlhenny and Ercegovac [16] present an implementation of a decimal online floating-point multiplier onto a Virtex 5 FPGA. The multiplication is performed in digit-serial fashion, obtaining a radix-10 digit of the result per cycle, with the most significant digit first.

We finally considered a magnitude range reduction of the BCD operand digits as the best way to simplify and speed-up the generation of all the decimal partial product in parallel. We compare the following two options: a signed-digit radix-10 recoding (from $[0,9]$ to $[-5,5]$) of both input operands [7], and a signed-digit radix-5 recoding of the BCD multiplier operand only [10], [12], [13]. We have mapped the different proposals into 6-input LUTs and other available hardware on a state-of-the-art FPGA slice.

A. Signed-digit radix-10 recoding of X and Y

In this case, the corresponding block diagram of the architecture is shown in Fig. 1. Every BCD (4-bit) input digit is recoded between $[-5,5]$ in sign (1-bit) and magnitude (3-bit) format. The recoded operands are multiplied digit by digit, performing the multiplication of signs (logical XOR of

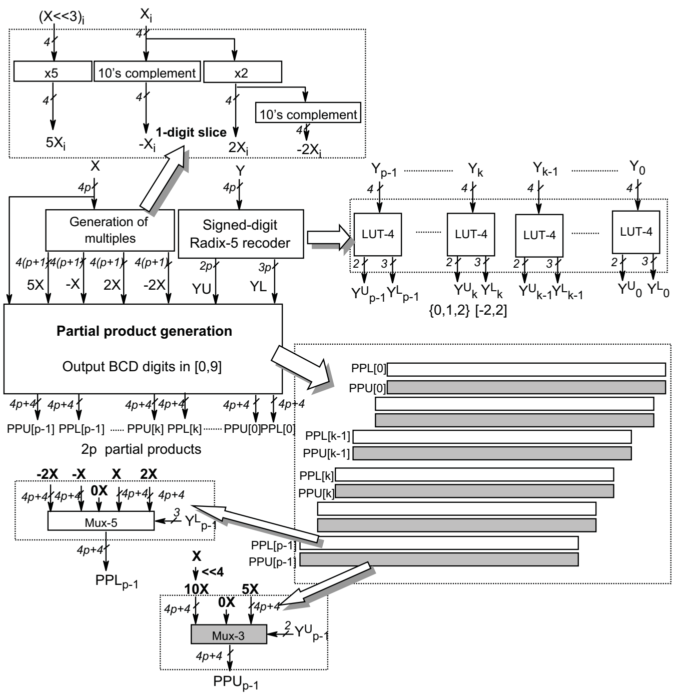


Figure 2. Signed-digit radix-5 recoding architecture

sign bits) and magnitudes separately. The multiplication of magnitudes requires 6-bit input lookup tables (LUT-6). The digits of the decimal partial product array generated, shown at the bottom of Fig. 1, are sign and magnitude numbers between $[-25, 25]$. This results in 6 LUTs per digit-by-digit multiplication since the partial product digit magnitudes are represented in BCD. The hardware cost for p-digit BCD operands is shown in column 2 of Table I.

B. Signed-digit radix-5 recoding of Y

Fig. 2 shows the block diagram for this case. Each BCD digit Y_k of the multiplier operand is recoded into two digits as

$$Y_k = 5\ Y_k^U + Y_k^L \qquad (3)$$

with $Y_k^U \in \{0, 1, 2\}$ and $Y_k^L \in \{-2, -1, 0, 1, 2\}$. This recoding needs five 4-input LUTs per digit (2 LUTs for Y_k^U and 3 LUTs for Y_k^L). Two partial products are generated per BCD multiplier digit. The upper partial products are generated as $X \times Y_k^U$, while the lower partial products are $X \times Y_k^L$, where X the BCD multiplicand. In a straightforward implementation [10], two BCD multiplicand multiples $2X$ and $5X$ are pre-computed before partial product generation using combinational logic (each digit of a multiple generator can be mapped into four LUT-4 or two LUT-6). Two negative multiples $-X$ and $-2X$ are also pre-computed as the 9's complement of X and $2X$ respectively. Next, each upper partial product is generated using a row of multiplexers controlled by Y_k^U that selects $5X$ or $10X$. The $0X$ multiple is selected by a reset of the output register. The lower partial products are generated from digits Y_k^L by selecting the correspondent multiple $-2X$, $-1X$, $1X$ or $2X$. The hardware cost (p-digit operands) is shown in column 3 of Table I, and is higher than the hardware cost of the first scheme.

C. Partial product reduction

However, before taking any decision about the preferred design, we also have to take into account the significant contribution of the partial product reduction to the total cost of the multiplier. Faster parallel multipliers use multiple adders connected in a tree configuration ($O(\log_2 n_{pp})$) to reduce the n_{pp} partial products generated into a final non-redundant product. The number of partial products generated is roughly doubled ($n_{pp} = 2p$ instead of $n_{pp} = p+1$) in the second case, though all of them are BCD operands (digits in $[0, 9]$) instead of signed-digit operands (digits in $[-25, 25]$).

The partial products generated by the scheme of Fig. 1 can be efficiently reduced using a decimal signed-digit adder tree [7]. Besides, these partial products have to be converted from $[-25, 25]$ to an appropriate range for decimal signed-digit addition. A decimal signed-digit adder [23] reduces two input digits ($\subset [-a, a]$ with $a \in \{5, 6, 7, 8, 9\}$) into a digit in the same range, producing a transfer digit into the next decimal position. This type of adders does not fit well in the building blocks of current FPGAs, leading to tree structures far from optimal in terms of area and delay. In Table I we show the estimated complexity of decimal signed-digit adder tree proposed by Erle et al. [7]. We use a first level of decimal signed-digit adders for the conversion of partial product digits from $[-25, 25]$ to $[-6, 6]$.

On the other hand, the reduction of the BCD partial products generated by the scheme of Fig. 2 can be performed using a variety of adders which can be mapped more efficiently on a FPGA. Proposed implementations uses either carry-ripple or carry-save adder trees. Decimal carry-save adder trees [5], [6], [10], [13] require additional logic for decimal correction and have much more interconnections which complicate the FPGA implementation and routing. Lang and Nannarelli [10] use a carry-save tree built of 4-bit BCD adders and 8-bit counters to reduce the $2p$ BCD partial products generated. The result of this reduction is a double word operand. The final BCD product is obtained from the sum of these two terms in a BCD carry-propagate adder. The estimated hardware cost of this design is shown in Table I in terms of LUT-6. An alternative is to implement a BCD carry-ripple adder tree [18] that can make use of the dedicated fast carry chain. Its hardware cost, shown in Table I, depends on how efficient are mapped these BCD adders on FPGA. Fast FPGA implementations of BCD carry-chain adders [22] require more than double the hardware of an equivalent binary carry-ripple adder. Besides, since the logic depth of these BCD adders is also doubled, the delay of a tree implementation is incremented in a proportional amount. To overcome both drawbacks, we designed a new BCD carry-ripple adder tree [24] that roughly halves the hardware cost and latency of previous implementations.

Table I
HARDWARE COST OF BCD MULTIPLIERS (p-DIGIT OPERANDS)

	Ref. [7] (Fig. 1) (# LUT-6)	Ref. [10] (Fig. 2) (# LUT-6)	Proposed (Fig. 3) (# LUT-6)
Recoding	$8p$	$3p$	$3p$
Gen. of Multiples	–	$8p$	$2p$
Partial product generation	$6p^2+6p$ p^2+2p+1 XORs	$8p^2+2p$	$8p^2+2p$
Reduction Tree	Signed-Digit†	Carry-Save‡	Carry-Ripple*
Total:			
$p=16$ (16×16-digit mult)	10720 LUTs 289 XORs	7322 LUTs	4880 LUTs
$p=34$ (34×34-digit mult)	46580 LUTs	31335 LUTs	21556 LUTs

$p=16$ and $p=34$ for IEEE 754-2008 decimal64 and decimal128 formats.
†Signed-Digit Adder Tree: $hc_{sd}\times(2p^2+p\times(\lceil\log_2 p\rceil/2))+hc_{cra1}\times 2p$
‡Carry-Save Adder Tree: $hc_{cra1}\times(9p^2/4+p\times(\lceil\log_2 p\rceil+3)/2)+5p/8$
*Carry-Ripple Adder Tree: $hc_{cra2}\times(2p^2+p\times(\lceil\log_2 p\rceil/2))$
$hc_{sd}=16$: Estimated hardware cost of a 1-digit decimal signed-digit adder [7], [23].
$hc_{cra1}=8$: Hardware cost of prior art 1-digit BCD carry-chain adder [22].
$hc_{cra2}=5$: Hardware cost of proposed 1-digit BCD carry-ripple adder [24].

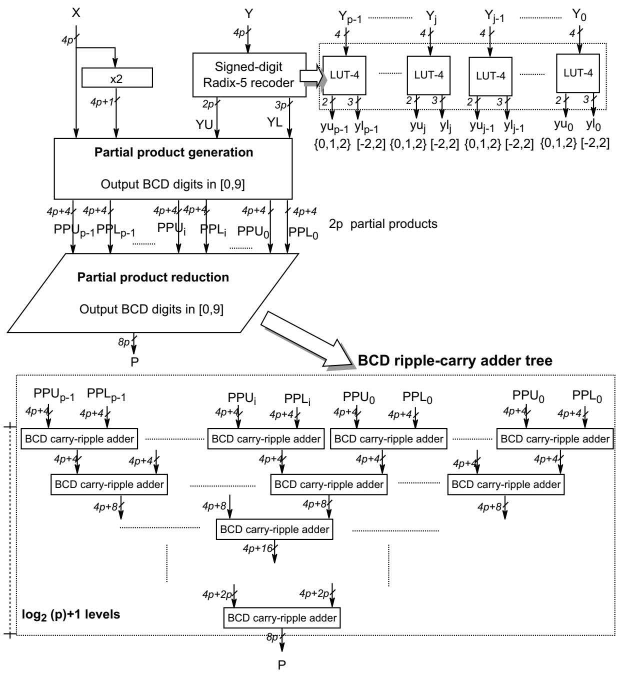

X
Y
4p
x2
4p+1
Signed-digit Radix-5 recoder
2p
3p
YU
YL
Y_{p-1} Y_j Y_{j-1} Y_0
LUT-4
yu_{p-1} yl_{p-1} yu_j yl_j yu_{j-1} yl_{j-1} yu_0 yl_0
{0,1,2} [-2,2]
Partial product generation
Output BCD digits in [0,9]
4p+4
PPU_{p-1} PPL_{p-1} PPU_i PPL_i PPU_0 PPL_0
2p partial products
Partial product reduction
Output BCD digits in [0,9]
8p
P
BCD ripple-carry adder tree
BCD carry-ripple adder
4p+8
4p+16
4p+2p
log2 (p)+1 levels

Figure 3. Proposed BCD parallel multiplier

III. PROPOSED METHOD

From the results of this survey, we opt for a parallel design with a signed-digit radix-5 recoding of the BCD multiplier Y and a BCD carry-ripple adder tree for partial product reduction. The general block diagram of the proposed decimal parallel multiplier is shown in Fig. 3. To improve the area and performance of the resultant implementations we propose a different organization of the basic scheme[10] that maps better in a Xilinx Virtex-5/6 device (see hardware cost in column 4 of Table I). In our case, only the $2X$ multiple is precomputed. The remaining BCD multiplicand multiples are computed on-the-fly in the BCD partial product generation block. The decimal partial product generation is detailed in Section III-A. We also have developed a new BCD carry-propagate addition algorithm [24], detailed in Section III-B, that leads to an efficient decimal carry-ripple adder tree implementation on a Xilinx Virtex-5/6 device (see hardware cost in row 4 of Table I with $hc_{cra}=5$). Moreover, since it presents many similarities with a binary carry-ripple adder tree, the partial product reduction tree can be pipelined using existing techniques for binary [25]. The combinational and pipelined BCD multipliers are presented in Section IV.

A. Partial product generation

We recode the BCD digits of Y using the signed-digit radix-5 recoding described by Equation (3) in Section II-B.

Each 4-bit BCD digit Y_k is recoded into an upper 2-bit digit $Y_k^U \in \{0,1,2\}$ and a lower 3-bit signed-digit $Y_k^L \in \{-2,-1,0,1,2\}$. This recoding generates $2p$ partial products aligned as shown in Fig. 2, that is, two BCD partial products $PPU[k]$ and $PPL[k]$ per BCD digit Y_k. The upper partial product $PPU[k]$ is computed from X and Y_k^U as

$$PPU[k] \;=\; \begin{cases} 10X & \text{if } Y_k^U == 2 \\ 10X/2 & \text{if } Y_k^U == 1 \\ 0X & \text{if } Y_k^U == 0 \end{cases} \tag{4}$$

We have mapped this equation into a Virtex-5/6 FPGA slice. Fig 4 shows the implementation of one BCD digit of $PPU[k]$. The LUTs implement the selection of digits $(10X)_i = X_{i-1}$,

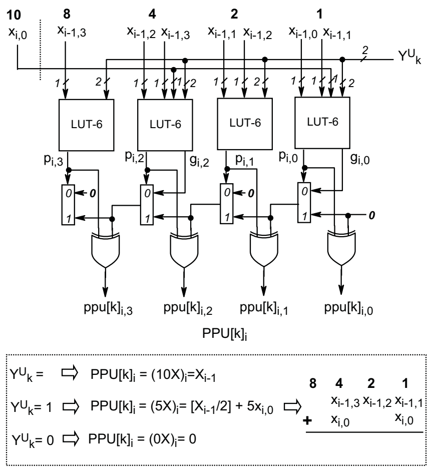


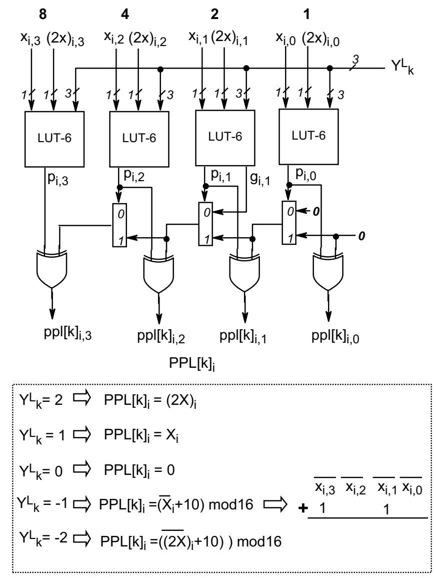
Figure 4. Generation of partial product $PPU[k]$ at digit i

$(5X)_i$ or 0. The BCD multiple $10X$ is obtained at no cost by shifting the multiplicand X 4 bits to the left. The multiple $5X$ is computed dividing $10X$ by 2, that is, each digit of $5X$ is given by $(5X)_i = \lfloor X_{i-1}/2 \rfloor + x_{i,0}\ 5$. This division is implemented as a 1-bit right shift of $10X$ followed by a conditional addition of 5 (BCD value 0101) at digit i when $x_{i,0}$ is one. This 4-bit addition is implemented using the fast carry logic of the Virtex-5/6 slice.

The $k-th$ lower partial product $PPL[k]$ is computed from $2X$, X and Y_k^L as

$$PPL[k] \;=\; \begin{cases} 2X & \text{if } Y_k^L == 2 \\ 1X & \text{if } Y_k^L == 1 \\ 0X & \text{if } Y_k^L == 0 \\ 10^p - X & \text{if } Y_k^L == -1 \\ 10^p - 2X & \text{if } Y_k^L == -2 \end{cases} \tag{5}$$

In Fig. 5 we show the implementation of this equation into the Virtex-5/6 slice for one digit. Only the multiples X and $2X$ are pre-computed. A negative partial product is obtained by computing on demand the 10's complement of the corresponding positive multiple (that is, by the 10's complement of X if $Y_K^L = -1$, and by the 10's complement of $2X$ if $Y_K^L = -2$).



8
4
2
1
$x_{i,3}$ $(2x)_{i,3}$
$x_{i,2}$ $(2x)_{i,2}$
$x_{i,1}$ $(2x)_{i,1}$
$x_{i,0}$ $(2x)_{i,0}$
Y^L_k
LUT-6
LUT-6
LUT-6
LUT-6
$p_{i,3}$
$p_{i,2}$
$p_{i,1}$
$g_{i,1}$
$p_{i,0}$
$ppl[k]_{i,3}$
$ppl[k]_{i,2}$
$ppl[k]_{i,1}$
$ppl[k]_{i,0}$
$PPL[k]_i$
$Y^L_k = 2$ ⇨ $PPL[k]_i = (2X)_i$
$Y^L_k = 1$ ⇨ $PPL[k]_i = X_i$
$Y^L_k = 0$ ⇨ $PPL[k]_i = 0$
$Y^L_k = -1$ ⇨ $PPL[k]_i = (\overline{X_i}+10) \bmod 16$ ⇨
$\overline{x_{i,3}}$ $\overline{x_{i,2}}$ $\overline{x_{i,1}}$ $\overline{x_{i,0}}$
+ 1 1
$Y^L_k = -2$ ⇨ $PPL[k]_i = (\overline{(2X)_i}+10)) \bmod 16$

Figure 5. Generation of partial product $PPL[k]$ at digit i

The 10's complement of X is obtained by computing the 9's complement of each BCD digit X_i as

$$9 - X_i = (15 - X_i + 10) - 16 = (\overline{X_i} + 10) \mod 16 \tag{6}$$

and adding one bit to the least significant digit of $PPL[k]$. To avoid a large carry propagation in the partial product generation, this bit is added in the partial product reduction tree. The 9's complement of X_i and $(2X)_i$ implies a 4-bit carry-propagate addition implemented using the fast carry logic of the Virtex-5/6 slice as shown in Fig. 5.

B. Partial product reduction

The $2p$ partial products generated are aligned and reduced using the tree of carry-ripple adders of Fig. 3. It has $\log_2(2p)$ levels of carry-ripple adders ranging from $4p+4$-bit to $6p$-bit length. The result of each two-operand BCD sum is passed to an adder a level down, and the carry ripples through the length of the adder. However, since the carry-ripple chains overlap, the signals are propagated as much as $8p$ bits in length plus $\log_2(2p)$ levels in depth. Therefore, the critical path delay of this tree is given by

$$t_{ppr} = t_{c-path} \times 2p + t_{s-path} \times (\log_2(2p) - 1) \tag{7}$$

where t_{c-path} is the delay of the carry chain of a 1-digit (4-bit) BCD adder and t_{s-path} the delay of the sum path of the 1-digit BCD adder. Bioul et al. [22] proposed several decimal adders that use the fast carry chain of the Virtex-6 slice to speedup a two-operand BCD addition. The implementation with the lowest hardware cost is shown in Fig. 6. In this design, the decimal carries are obtained by a direct implementation of a decimal carry-propagate recurrence $C_{i+1} = G_i \vee P_i\ C_i$. Boolean signals G_i and P_i indicate the conditions for the generation and propagation of decimal carries for each digit, and are true when the binary sum of the input digits $Z_i = X_i + Y_i$ is higher than 9 or equal to 9 respectively. When a

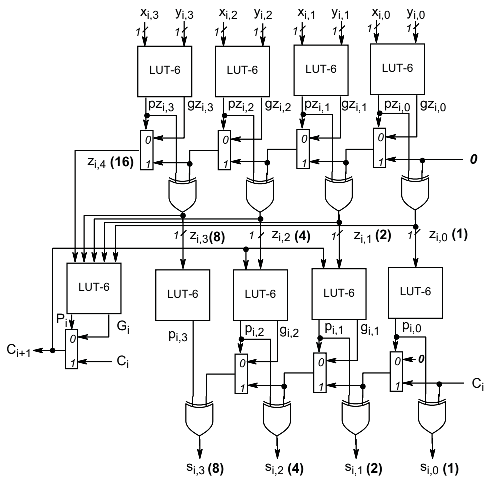


Figure 6. BCD Carry-Chain Adder (1-digit). Prior Art [22]

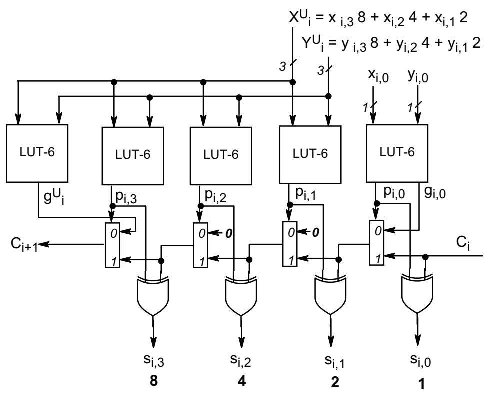


Figure 7. Proposed BCD Carry-Ripple Adder (1 digit)

decimal carry-out C_{i+1} is produced, the binary sum must be corrected by 6 (0110) to obtain the BCD sum digit S_i, that is $S_i = Z_i + 6\ C_{i+1} + C_i$. The problem of this design is its high hardware cost, equivalent to eight LUT-6, and that t_{s-path} is more than double the delay of a 4-bit binary carry-ripple adder, because the sum path goes through two successive levels of LUT-6 separated by the global routing.

We propose a different method for BCD carry-propagate addition [24] that lead to the implementation of Fig. 7. In this method, a correction factor of 6 (0110) is added to the BCD input digits X_i, Y_i when the sum $X_i^U + Y_i^U$ is equal or higher than 8. We denote by X_i^U, Y_i^U the 3 most significant bits of X_i and Y_i respectively. This conditional addition of 6 is merged with the binary addition of $X_i + Y_i + C_i$ using the Virtex-5/6 slice configuration of Fig. 7. Another boolean signal g_i^U, which indicates when $X_i^U + Y_i^U \geq 10$, is computed in a fifth LUT-6. In this way, the decimal carries C_{i+1} are obtained from a single binary carry-propagate addition of the modified BCD input operands. The BCD sum digit S_i is equal to the 4-bit binary sum

$$Z_i = \begin{cases} (X_i + Y_i + 6 + C_i) \mod 16 & \text{if } X_i^U + Y_i^U \geq 8 \\ (X_i + Y_i + C_i) \mod 16 & \text{else} \end{cases}$$

except when $Z_i \in \{14(1110), 15(1111)\}$. In this case, the +6 factor has been added in excess and the correct BCD sum digits are $S_i \in \{8(1110), 9(1001)\}$. To avoid corrections between levels of the adder tree, we allow the 4-bit combinations (1110), (1111) to be valid representations of decimal values 8 and 9. Only the output digits Z_i of the adder tree are corrected as

$$P_i = z_{i,3}\ 8 + (z_{i,2}\overline{z_{i,3}})\ 4 + (z_{i,1}\overline{z_{i,3}})\ 2 + z_{i,0} \qquad (8)$$

to obtain the digits P_i of the final BCD product using two 1-bit flip-flops of the Virtex-5/6 slice configured as two logical AND gates with an inverted input. With this method, the equivalent hardware cost of 1-digit BCD adder is reduced to 5 LUTs per digit and t_{c-path}, t_{s-path} are roughly the delays of the carry and sum paths of a 4-bit binary carry-ripple adder.

Table II
SYNTHESIS RESULTS FOR THE COMBINATIONAL IMPLEMENTATION

BCD Mult. size ($p_X \times p_Y$)	Hardware Cost (# LUT-6)	Delay (ns)
4×4	336	4.91
8×4	642	5.17
8×8	1268	6.41
16×8	2396	6.90
16×16	4880	8.38
34×16	10010	9.48
34×34	21611	14.39

IV. PARALLEL BCD MULTIPLIERS

Combinational and pipelined versions of the proposed parallel BCD multiplier were designed in VHDL for arbitrary $p_x \times p_y$-digit wide multiplications using the Xilinx ISE Design Suite 11.4. The basic building blocks of the multiplier were instantiated into components of the Xilinx Virtex-6 library to have more control over the mapping process. The architectures were synthesized in a Virtex-6 XC6VLX75T device with speed grade-3 using the XST compiler and simulated with Modelsim SE 6.5.

A. Combinational Architecture

We have synthesized the combinational BCD multiplier of Fig. 3 for different operand sizes. A 16×16-digit BCD multiplier is required for the multiplication of BCD significands in the IEEE Decimal64 format. For the IEEE Decimal128 format, the BCD multiplication size supported must be 34×34. Besides, we have included several rectangular multipliers in this list. The area and delay synthesis results are presented in Table II.

The hardware cost is expressed in terms of equivalent LUT-6 units and the delay in ns.

B. Pipelined Architecture

We consider a 2-stage pipeline for the multiplicand recoding and generation of partial product. We place each level of the adder tree in a different pipeline stage. Besides, each carry-ripple adder is pipelined in chunks of m bits at most. The total number of pipelined stages is equal to $2 + \lceil 8p/m \rceil + \log_2(p)$. In

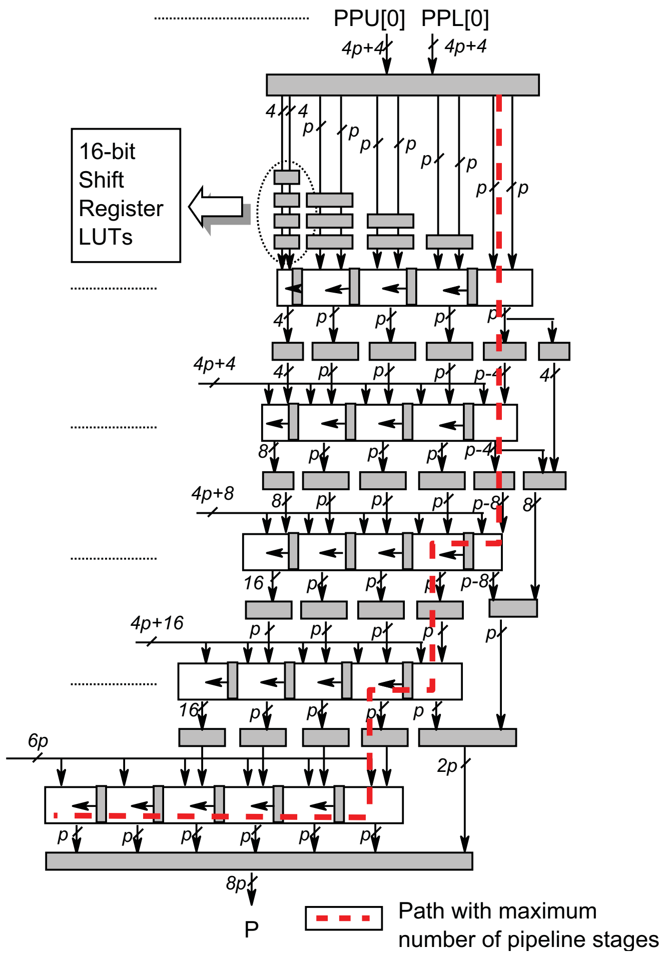

PPU[0]
PPL[0]
4p+4
4p+4
16-bit Shift Register LUTs
4p+4
4p+8
4p+16
6p
2p
8p
P
Path with maximum number of pipeline stages

Figure 8. Detail of a pipelined 16×16-digit BCD Multiplier ($m = p = 16$)

Fig. 8 we show this register placement for the least significant part of the partial product reduction tree of a 16×16-digit BCD multiplier and $m = p = 16$. A significant amount of registers is required for input synchronization [25]. To reduce the hardware cost, these synchronization registers are placed in the first pipeline level of the tree and packed together as 16-bit shift register LUTs.

We have pipelined in this way the 16×16 BCD parallel multiplier and implemented for different values of m. The synthesis results are presented in Table III.

V. Comparison

We compare our synthesis results with three different parallel implementations of decimal multiplication on FPGAs recently reported [14], [18], [19]. Vestias and Neto [19] present results for 8×8-digit and 16×16-digit multipliers in a Virtex-4 SX35-12 FPGA. These designs can make use of embedded binary multipliers to reduce the number of utilized LUTs. Their 16×16 multiplier uses 16 embedded DSP blocks and 3005 LUTs with a delay of 68 ns, or equivalently 6493 LUTs with a delay of 65 ns.

On the other hand, our combinational 16×16 multiplier has a cost of 4880 LUTs and a latency of 8.38 ns on Virtex-6 with speed grade -3, though these figures are difficult to extrapolate to Virtex-4 for a fair comparison with [19]. To give a notion of how Virtex-4 and Virtex-6 implementations could compare, we show in Table IV the area and latency figures of a 53×53-bit binary multiplier provided by the Xilinx Core generator. Virtex-6 performance is practically doubled compared to Virtex-4 due to feature size reduction of the underlying VLSI technology. These results are shown in Table IV.

Although in terms of area both implementations are comparable, our 16×16 BCD combinational multiplier is more than 7 times faster than any of the designs presented by Vestias and Neto [19].

Sutter et al. [18] reported results for combinational implementations of BCD multipliers up to 32×16 digits. The 16×16 BCD multiplier has a latency of 26.9 ns and occupies 22033 LUTs on a Virtex-4 with speed grade -11. Again, a direct comparison of this implementation with our Virtex-6 implementation is quite unfair. However, we observe that the reduction in latency is more than 3 times between the implementation of Sutter et al. [18] (Virtex-4) and our combinational multiplier (Virtex-6). For hardware cost, the reduction is around a factor of 4.5. Therefore, these significant improvements cannot be explained only by the use different implementation technologies. We have also included in this Table the synthesis results of a 16×16 BCD multiplier reported by Baesler et al. [14] in a Virtex-II Pro XC2VP30 with speed grade -6.

VI. Conclusion and Future Work

In this work we presented the design of a BCD multiplier and its implementation on a Virtex-5/6 FPGA. We performed a survey of prior techniques of BCD multiplication to find the most adequate for low-latency area-efficient FPGA implementations. The proposed architecture is based on a parallel generation of the decimal partial products using a signed-digit radix-5 recoding of the multiplier operand. The partial product reduction is performed in a tree of BCD carry-ripple adders. We developed new methods for an efficient implementation of both generation and reduction techniques on the Virtex-5/6 slice. Combinational and pipelined versions of the BCD multiplier were synthesized in a Virtex-6 speed grade-3 device and the results compared with those of three recent FPGA implementations of parallel BCD multipliers. We show that the proposed design is a very competitive option for high-performance low-latency implementations of BCD multiplication on Virtex-5/6 FPGAs at a moderate hardware cost.

Future work includes the extension of the proposed BCD multiplier architectures to decimal floating-point. Also, we are planning to integrate the proposed multiplier in a core of the FloPoCo project[1], a generator of arithmetic cores for FPGAs, and to support more FPGA targets than Virtex-5/6 families.

[1]http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

Table III
SYNTHESIS RESULTS FROM PIPELINING OF THE 16×16 BCD MULTIPLIER

m (bits)	#pipe. stages	Clock Freq. (Mhz)	Delay latency× #stages	Hardware Cost #LUT-6	#FF
64	8	453 Mhz	17.7 ns (2.21×8)	5008	6926
32	10	579 Mhz	17.2 ns (1.72×10)	6032	6957
16	14	672 MHz	20.9 ns (1.49×14)	6544	7020
8	22	731 Mhz	30.1 ns (1.37×22)	6800	7148

Table IV
COMPARISON OF 16×16 BCD MULTIPLIERS

Design	Delay (ns)	Hardware Cost (# components)
Combinational	8.4 ns	4880 LUT
Pipelined (8-stage)	17.7 ns (8@453 Mhz)	5008 LUT 6926FF
Virtex-6 53×53*-bit Binary Mult.*	*26.6 ns (12@450 Mhz)*	*150LUT-FF 133LUT 276FF 10DSP*
Vestias et al. w/DSP [19]	68 ns	3005 LUT 16 DSP
Vestias et al. wo/DSP [19]	65 ns	6493 LUT
Sutter et al. [18]	26.9 ns	22033 LUT
Virtex-4 53×53*-bit Binary Mult.*	*45 ns (18@400 Mhz)*	*279Slices 386LUT 437FF 16DSP*
Baesler et al. [14] (Virtex-II Pro)	54 ns (6@111 Mhz)	5289 LUT

ACKNOWLEDGMENT

The authors would like to thank Bogdan Pasca for his valuable help and many interesting suggestions throughout the preparation of this paper.

REFERENCES

[1] M. Cowlishaw, "Decimal floating-point: Algorism for computers," in *Proc. 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, Santiago de Compostela, Spain, Jul. 2003, pp. 104–111.

[2] L. Eisen *et al.*, "IBM POWER6 accelerators: VMX and DFU," *IBM Journal Research and Development*, vol. 51, no. 6, pp. 663–684, Nov. 2007.

[3] E. Schwarz, J. Kapernick, and M. Cowlishaw, "Decimal floating-point support on the IBM System z10 processor," *IBM Journal Research and Development*, vol. 51, no. 1, pp. 4:1–4:10, Jan./Feb. 2009.

[4] *IEEE Std 754(TM)-2008, IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std., Aug. 2008.

[5] L. Dadda and A. Nannarelli, "A variant of a radix-10 combinational multiplier," in *Proc. IEEE Int. Symposium in Circuits and Systems (ISCAS 2008)*, Santiago de Compostela, Spain, May 2008, pp. 3370–3373.

[6] M. A. Erle and M. J. Schulte, "Decimal multiplication via carry-save addition," in *Proc. of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2003)*, The Hague, The Netherlands, Jun. 2003, pp. 348–358.

[7] M. A. Erle, E. M. Schwarz, and M. J. Schulte, "Decimal multiplication with efficient partial product generation," in *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH17)*, Cape Cod, MA, USA, Jun. 2005, pp. 21–28.

[8] B. J. Hickman, A. Krioukov, M. A. Erle, and M. J. Schulte, "A parallel IEEE P754 decimal floating-point multiplier," in *Proc. 25th IEEE Conference on Computer Design (ICCD'07)*, Lake Tahoe, CA, USA, Oct. 2007, pp. 296–303.

[9] R. D. Kenney, M. J. Schulte, and M. A. Erle, "High-frequency decimal multiplier," in *Proc. 2004 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2004)*, San Jose, CA, USA, Oct. 2004, pp. 26–29.

[10] T. Lang and A. Nannarelli, "A radix-10 combinational multiplier," in *Proc. 40th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, Oct. 2006, pp. 313–317.

[11] G. Jaberipur and A. Kaivani, "Binary-coded decimal digit multipliers," *IET Comput. Digit. Tech.*, vol. 1, no. 4, pp. 377–381, Jul. 2007.

[12] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high-performance parallel decimal multipliers," in *Proc. 18th IEEE Symposium on Computer Arithmetic (ARITH18)*, Montpellier, France, Jun. 2007, pp. 195–204.

[13] ——, "Improved design of high-performance parallel decimal multipliers," *IEEE Trans. Comput.*, vol. 59, no. 5, pp. 679–693, May 2010.

[14] M. Baesler and T. Teufel, "Fpga implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier," in *Proc. IEEE International Conference on Reconfigurable Computing and FPGAs 2009 (ReConFig'09)*, Cancun, Mexico, Dec. 2009, pp. 6–11.

[15] R. James, K. Jacob, and S. Sasi, "Performance analysis of double digit decimal multiplier on various FPGA logic families," in *Proc. V Southern Programmable Logic Conference (SPL09)*, Sao Carlos, Brazil, Apr. 2009, pp. 165–170.

[16] R. McIlhenny and M. Ercegovac, "On the design of a radix-10 online floating-point multiplier," in *Proc. SPIE on Advanced Signal Processing Algorithms, Architectures, and Implementations*, San Diego, CA, USA, Aug. 2009.

[17] C. Minchola and G. Sutter, "A FPGA IEEE 754-2008 Decimal64 floating-point multiplier," in *Proc. IEEE International Conference on Reconfigurable Computing and FPGAs 2009 (ReConFig'09)*, Cancun, Mexico, Dec. 2009, pp. 59–64.

[18] G. Sutter, E. Todorovich, G. Bioul, M. Vazquez, and J.-P. Deschamps, "FPGA implementations of BCD multipliers," in *Proc. IEEE International Conference on Reconfigurable Computing and FPGAs 2009 (ReConFig'09)*, Cancun, Mexico, Dec. 2009, pp. 36–41.

[19] M. P. Vestias and H. C. Neto, "Parallel decimal multipliers using binary multipliers," in *VI Southern Programmable Logic Conference, (SPL 2010)*, Ipojuca, Brazil, Mar. 2010, pp. 73–78.

[20] "Virtex-6 configurable logic block user guide," http://www.xilinx.com/, Xilinx Inc., Sep. 2009.

[21] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev, "A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format," *IEEE Trans. Comput.*, vol. 58, no. 2, pp. 148–162, Feb. 2009.

[22] G. Bioul, M. Vazquez, J.-P. Deschamps, and G. Sutter, "Decimal addition in FPGA," in *Proc. V Southern Programmable Logic Conference(SPL09)*, Sao Carlos, Brazil, Apr. 2009, pp. 101–108.

[23] A. Svoboda, "Decimal adder with signed-digit arithmetic," *IEEE Trans. Comput.*, vol. C, pp. 212–215, Mar. 1969.

[24] A. Vazquez and F. de Dinechin, "Multi-operand decimal tree adders for FPGAs," INRIA, Research Report, Sep. 2010.

[25] F. de Dinechin, H. D. Nguyen, and B. Pasca, "Pipelined FPGA adders," École Normale Supérieure de Lyon, LIP Research Report ensl-00475780, Apr. 2010.