Multi-operand Decimal Adder Trees for FPGAs

Alvaro Vazquez, Florent De Dinechin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Multi-operand Decimal Adder Trees for FPGAs

Álvaro Vázquez — Florent de Dinechin

N° 7420

October 2010

ISSN 0249-6399 ISRN INRIA/RR--7420--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Multi-operand Decimal Adder Trees for FPGAs

Álvaro Vázquez *, Florent de Dinechin †

Thème : Algorithms, Certification, and Cryptography
Équipe-Projet Arénaire

Rapport de recherche n° 7420 — October 2010 — 20 pages

Abstract: The research and development of hardware designs for decimal arithmetic is currently going under an intense activity. For most part, the methods proposed to implement fixed and floating point operations are intended for ASIC designs. Thus, a direct mapping or adaptation of these techniques into a FPGA could be far from an optimal solution. Only a few studies have considered new methods more suitable for FPGA implementations. A basic operation that has not received enough attention in this context is multi-operand BCD addition. For example, it is of interest for low latency implementations of decimal fixed and floating point multipliers and decimal fused multiply-add units. We have explored the most representative proposals for multi-operand BCD addition and found that the resultant implementations in FPGAs are still very inefficient in terms of both area and latency when compared to their binary counterparts. In this paper we present a new method for fast and efficient implementation of multi-operand BCD addition in current FPGA devices. In particular, our proposal 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 implementations halve the area and latency of previous proposals, presenting area and delay figures close to those of optimal binary adder trees.

Key-words: Decimal arithmetic, IEEE 754-2008, BCD multi-operand addition, carry-ripple adder, pipelined adder trees, FPGA implementation

* INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon, Alvaro.Vazquez-Alvarez@inrialpes.fr
† ENS de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon, Florent.de.Dinechin@ens-lyon.fr

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Addition décimale multi-opérande sur FPGA

Résumé : La recherche sur l'implantation en matériel de l'arithmétique décimale est actuellement très active, la plupart des travaux portant sur des opérateurs pour les processeurs, en virgule fixe ou flottante. Mais les techniques développées pour un circuit intégré n'aboutissent pas forcément à une implémentation optimale dans un FPGA. Il n'y a que peu d'études ciblant explicitement les FPGA. Cet article s'intéresse dans ce contexte, à l'addition BCD multi-opérande, au cœur de multiplieurs et de multiplieurs-accumulateurs à faible latence.

Nous étudions les architectures proposées pour cette opération décimale, et nous observons que, sur FPGA, leur performance (surface et latence) est très inférieure à celle des opérations binaire à précision comparable. Nous présentons donc dans cet article une nouvelle technique d'addition BCD multi-opérandes qui s'avère plus efficace que les propositions précédentes sur les FPGA actuels. Elle s'adapte particulièrement bien à la structure fine des FPGA Xilinx Virtex-5/Virtex-6, et se prête bien au pipeline. Les résultats de synthèse montrent que notre implémentation divise par deux la surface et la latence par rapport aux propositions précédentes, les ramenant à des valeurs comparables à celles des meilleurs additionneurs multi-opérandes binaires.

Mots-clés : Arithmétique décimale, IEEE 754-2008, addition BCD multi-opérande, additionneur à propagation de retenue, arbres d'addition pipelinés, implémentation sur FPGA

1 Introduction

For years, the use of dedicated decimal hardware has been limited to mainframe business computers [25, 6, 4] and handheld calculators at the low end [7]. Very recently a renewed interest in providing hardware acceleration for decimal arithmetic has emerged [9]. It has been boosted by the numerically intensive computing requirements of new commercial, financial and Internet applications, such as e-commerce and e-banking. Because of the perspectives of a more widespread use of decimal processing, the revised IEEE 754-2008 Standard for Floating-Point [19] incorporates a specification for decimal arithmetic.

Besides, the advances in FPGA technology have opened up new opportunities to implement efficient floating-point coprocessors for specialized tasks [28] for a fraction of the cost of an ASIC implementation. Thus, even though most of the current research on decimal arithmetic is targeted at high-performance VLSI design [10, 12, 14, 20, 21, 26, 31], a few works present implementations of decimal arithmetic units for FPGAs [3].

The design of decimal units for FPGAs faces several challenges: first, the inherent inefficiency of decimal representations in systems based on two-state logic and a complex mapping of the decimal arithmetic rules into boolean logic. On the other hand, the special built-in characteristics of FPGA architectures make it difficult to use many well-known methods to speedup computations (for example, carry-save and signed-digit arithmetics). Therefore, it may be preferable to develop specific decimal algorithms more suitable for FPGAs rather than adapting existing ones targeted for ASIC platforms.

In this context, we present the algorithm, architecture and FPGA implementation of a novel unit to perform fast addition of a large amount of decimal (BCD) fixed-point or integer operands. This operator is also of key importance for other arithmetic operations such as decimal multiplication and division. Among the novel contributions of this work are the following:

- A new decimal carry-propagate addition algorithm.
- A decimal digit adder based on this algorithm that makes a full use of the fast carry chain and slice logic of a state-of-the-art FPGA device.
- A highly pipelineable, minimum logic depth tree structure of decimal adders, which can be configured to sum any arbitrary number of BCD operands running at clock rates up to 765 MHz in a Xilinx Virtex-6 device.

The structure of the paper is as follows. In Section 2 we present a survey of prior work on multi-operand addition. In Section 3 we introduce a new method for BCD carry-propagate addition. Section 4 presents the combinational and pipelined architectures of the proposed BCD multi-operand adders and their implementation in a Xilinx Virtex-5/6 FPGA [32]. Estimations for area and delay and a comparison are shown in Section 5. Finally, the conclusions are summarized in Section 6.

2 Survey of Multi-Operand Addition Methods

In this Section we present prior work on binary and decimal multi-operand addition and analyse different representative FPGA implementations, discussing their associated advantages and costs. Based on the conclusions extracted from this survey, in Section 3 we present a proposal that leads to more efficient implementations of decimal multi-operand adders in FPGA platforms.

2.1 Multi-operand adders for FPGAs

Binary multi-operand adders are generally arranged in two ways: in an array of rows or in a tree-like structure. In an array configuration each row of adders reduces one further operand, so that m levels of adders are required to reduce m operands into a final one. On the other hand, in a m-operand adder tree the number of logic levels is $\log_2(m)$ (or $\log_2(m) - 1$ levels for a signed-digit or a carry-save adder tree, but a final carry-propagate adder is needed in this case). Furthermore, the hardware cost of both configurations is similar, so tree configurations are usually preferred, though the array has a more regular routing.

Concerning the type of adder, the delay of each carry-propagate adder (carry-ripple, carry-lookahead,...) depends on the length of its input operands (delay proportional to n for an n-bit binary carry-ripple adder, $O(\log_2 n)$ for a carry lookahead adder). To reduce the latency of addition on FPGAs, these devices incorporate dedicated fast carry chain paths [32]. This favours simple carry-ripple implementations in FPGAs over other carry-propagate adder topologies, except in the case of non-pipelined implementations for large size operands, but at the expense of a high hardware cost [33]. Moreover, pipelining techniques in FPGAs can be applied more effectively to carry-ripple adders [11]. On the other hand, signed-digit and carry-save adders have a constant computation time, delaying the computation of the carry propagation until the end. However, a straightforward implementation on FPGAs [22] roughly requires double hardware than a carry-ripple adder, and does not exploit the fast carry chain to improve speed. Several authors [18, 23] have recently proposed efficient mappings of carry-save adders on FPGAs but only for the binary case.

The delays of both carry-ripple and carry-save adder trees are proportional to $n + \log_2(m) - 1$ [1], because the horizontal delays through the $\log_2(m)$ levels of carry-ripple chains overlap. In ASIC implementations the advantage of the carry-save adder tree lies on the flexibility of routing, so the critical path delay can be reduced by an optimization of interconnections between full adders. In FPGAs, a conventional carry-save adder tree of full adders [22] is slower than a carry-ripple adder tree due to the complex routing, so the carry-ripple adder tree configuration is generally preferred. This can be partially solved by mapping efficiently compressors that perform operand reductions larger than 3:2 [18, 23].

Decimal multi-operand adders can also be implemented as carry-ripple, signed-digit or carry-save adder trees but they require additional logic for decimal correction. Therefore, delay and hardware cost are larger than in the binary case. Next, we describe the most representative methods proposed for decimal fixed-

[1] We consider that the final carry propagation in the case of the signed-digit or carry-save adder tree is implemented in a ripple-carry way

point/integer carry-propagate (carry-ripple, carry lookahead,...) and carry-free (signed-digit, carry-save...) addition.

2.2 BCD addition/subtraction methods

The IEEE 754-2008 Standard [19] specifies three basic formats (Decimal32, Decimal64, and Decimal128) and two different encodings (binary or BID and decimal or DPD) for 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 [8]. On the other hand, the decimal encoding is more oriented to hardware implementations, and uses BCD to represent the coefficient as a p-digit decimal 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

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

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. Note that only ten out of the sixteen possible 4-bit combinations are used (0000_2 to 1001_2, where the subscript $_2$ indicates that the number is represented in binary).

We want to obtain a non-redundant operand S that represents the decimal addition of m integer operands $Z[k]$ of p digits coded in BCD, that is

$$S = \sum_{k=0}^{m-1} Z[k] = \sum_{k=0}^{m-1} (\sum_{i=0}^{p-1} Z[k]_i \ 10^i) \tag{3}$$

with $Z[k]_i = \sum_{j=0}^{3} z[k]_{i,j} \ 2^j$.

To reduce all the operands $Z[k]$ into a BCD sum S, we can perform recursively two-operand BCD carry-propagate additions or use decimal carry-free additions with a final conversion to obtain the result in the conventional non-redundant BCD format. This conversion is usually implemented as a BCD carry-propagate addition. Several decimal carry-propagate, carry-save and signed-digit addition methods have been proposed:

2.2.1 BCD carry-propagate addition methods

An straightforward approach to add two BCD operands is to perform a binary carry-propagate addition to form an intermediate result [24]. To generate the correct BCD sum, the 6 unused combinations (1010_2 to 1111_2) need to be skipped from the digits of the intermediate result. The usual method is to add correction factors of 6 (0110_2). This may produce subsequent carry propagation between digits, which would require the computation of a new intermediate

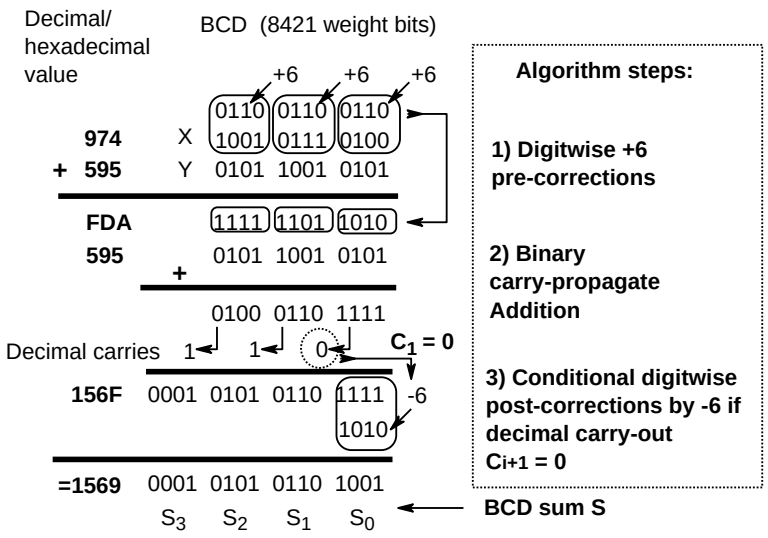


Figure 1: Example of BCD addition.

sum. BCD subtraction is performed by adding the ten's complement of the subtrahend and the minuend [24]. The ten's complement of a BCD operand Y is obtained by the 9's complement of each 4-bit BCD digit Y_i in two possible ways as

$$9 - Y_i = \begin{cases} 15 - 6 - Y_i = 15 - (Y_i + 6) = \overline{Y_i + 6} \\ -16 + (15 - Y_i) + 10 = (\overline{Y_i} + 10) mod 16 \end{cases} \quad (4)$$

followed by an addition of one bit at the least significant digit (usually introduced as a carry input in a BCD addition). A line over a digit or an operand denotes a bit complementing operation.

Many techniques have been proposed to reduce the delay of the basic method, which can be included in one of these two groups:

1. **Methods based on pre-correction and post-correction of the binary sum [2, 15, 16, 17, 30].** An example of BCD addition is shown in Fig. 1. To speed up the evaluation, the correction factors of 6 are added digit-wise[2] to a BCD input operand. In this way, all the decimal carries are obtained from a single binary carry-propagate addition of the modified operands. The decimal carries correspond to the carries generated each 4 binary positions. For BCD subtraction, the bits in the subtrahend are inverted (9's complement plus 6 digit addition). The intermediate binary sum is computed next. In a basic implementation, the intermediate result is finally corrected by subtracting six (digit-wise operation) from each digit position with a decimal carry-out of zero. These methods are specially suited for combined binary/BCD implementations. BCD adders of this type were implemented in the IBM z/System processors [4].

2. **Direct decimal addition [25] and variants [1, 3, 14, 21].** In these methods, the decimal carries are obtained by a direct implementation of

[2]This correction does not produce a carry propagation between digits, since each incremented digit is bounded by $1001_2 + 0110_2 = 1111_2$.

a decimal carry-propagate recurrence, such as:

$$C_{i+1} = G_i \vee P_i \; C_i \tag{5}$$

where $\vee$ is the logical OR and the logical AND is represented by a blank space between variables. Signals G_i and P_i indicate the conditions for the generation and propagation of decimal carries for each digit, and only depend on the value of the input digit sums $Z_i = X_i + Y_i \in \{0, \ldots, 18\}$, that is:

$$G_i = \begin{cases} 1 & \texttt{If } Z_i > 9 \\ 0 & \texttt{Else} \end{cases} \quad P_i = \begin{cases} 1 & \texttt{If } Z_i = 9 \\ 0 & \texttt{Else} \end{cases} \tag{6}$$

The decimal carries can be evaluated by means of any conventional carry-propagate technique (carry-ripple, carry lookahead, carry-skip...). When a decimal carry-out C_{i+1} is produced at a decimal position i, the corresponding 4-bit binary sum Z_i must be corrected by 6 (0110_2) to obtain the BCD sum digit S_i, that is $S_i = Z_i + 6 \; C_{i+1} + C_i$. S_i can be implemented using either a direct evaluation of the boolean expressions [13, 25] or a carry-select mechanism with +6 biased conditional binary sums [3, 21]. The first approach was used in the IBM S/390 machines [6].

2.2.2 BCD carry-free addition methods

Decimal carry-save addition methods use a two BCD word to represent sum and carry [10, 20] or a BCD sum word and a carry bit per digit [13, 21]. The first group implements decimal addition mixing binary 3:2 carry-save adders (made of full adders) or binary counters with combinational logic for decimal correction. This logic consists on digit additions of +6 or +12 (Modulo 16) to correct the out of range 4-bit combinations of the intermediate binary sum and carry words. In [31] is proposed the use of decimal codes different from BCD to implement decimal carry-save addition using a binary 3:2 carry-save adder and a simple module that multiplies by 2 the carry operand. The second group of decimal carry-save methods [13, 21] uses different topologies of 4-bit BCD carry-propagate adders to implement decimal carry-save addition. Each digit adder takes two BCD digits and a 1-bit carry input and generates a 1-bit carry output and the BCD sum digit.

On the other hand, decimal signed-digit adders [14, 27, 29] use a signed-digit redundant representation (digits in $\{-a, \ldots, 0, \ldots, a\}$, $a \in \{6, 7, 8, 9\}$) of the decimal operands. The carry propagation is limited to the next significant position, although the resultant VLSI implementations are complex and present irregular layouts. Moreover, there is no special feature in current FPGAs to simplify the implementation of signed-digit arithmetic [5].

2.3 Estimated cost of FPGA implementations

We have performed a comparative study of several representative BCD multi-operand addition methods [3, 14, 21, 31]. We modeled tree structures of these decimal adders parametrized by p (operand length in digits) and m (number of input operands) and estimated the delay and hardware cost of a state-of-the-art FPGA implementation, more specifically for Virtex-5/6 devices. We use this first order evaluation model to compare the complexity of the different

Type of BCD Adder Tree	Delay (ns) $\{p, m\} = 16$
Signed-Digit [14]	t_{ccap}+$(4t_s + 2t_c) \times \lceil \log_2(m) \rceil$	17.91
Carry-Save [21]	t_{ccap}+$t_{cca1}(\lceil \log_2(m) \rceil$+$\lceil (m\text{-}1)/8 \rceil)$	15.35
Carry-Save [31]	t_{ccap}+$2t_s(\lceil \log_{3/2}(m) \rceil - 1)$	13.67
Carry-Chain (Low-cost) [3]	$t_c \times p/4$+$t_{cca1} \lceil \log_2(m) \rceil$	8.87
Carry-Chain (Fast version) [3]	$t_c \times p/4$+$t_{cca2} \lceil \log_2(m) \rceil$	8.12
Carry-Ripple (proposed)	$t_c \times p$+$t_s \lceil \log_2(m) \rceil$	4.72

$t_c = 0.06ns$: Delay of a 4-bit slice fast carry-path in Virtex-6 speed grade -3.
$t_s = 0.94ns$: Delay of a 1-bit sum-path in Virtex-6 speed grade -3.
$t_{cca1} \approx 2.2t_s + 1.5t_c = 2.16ns$: Delay of 1-digit carry-chain adder (Low-cost) [3].
$t_{cca2} \approx 2t_s + 1.5t_c = 1.97ns$: Delay of 1-digit carry-chain adder (Fast)[3].
$t_{ccap} \approx t_{cca1} + (t_c \times (p-1)/4)$: Delay of p-digit carry-chain adder [3].

Type of BCD Adder Tree	Hardware Cost (# LUT-6) $\{p, m\} = 16$
Signed-Digit [14]	$hc_{sd} \times p \times (m-1) + hc_{cca1} \times p$	3968
Carry-Save [21]	$hc_{cca1} \times p \times (m + \lceil (m-1)/8 \rceil)$	2304
Carry-Save [31]	$hc_{csa} \times p \times (m-2) + hc_{cca1} \times p$	1472
Carry-Chain (Low-cost) [3]	$hc_{cca1} \times p \times (m-1)$	1920
Carry-Chain (Fast version) [3]	$hc_{cca2} \times p \times (m-1)$	2400
Carry-Ripple (proposed)	$hc_{cra2} \times p \times (m-1)$	1200

$hc_{sd} = 16$: Hardware cost of a 1-digit decimal signed-digit adder [29].
$hc_{csa} = 6$: Hardware cost of a 1-digit decimal 3:2 carry-save adder [31].
$hc_{cca1} = 8$: Hardware cost of 1-digit BCD carry-chain adder (Low-cost)[3].
$hc_{cca2} = 10$: Hardware cost of 1-digit BCD carry-chain adder (Fast-version) [3].
$hc_{cra} = 5$: Hardware cost of proposed 1-digit BCD carry-ripple adder.

Table 1: Estimated delay and area figures for decimal adder trees (m BCD operands of p-digit length).

adder tree structures, not to obtain highly accurate area and delay figures. We present more accurate area and delay estimations from synthesis in Section 5. To simplify the calculations, we consider that the sum does not overflow and can be represented in p digits. Table 1 shows area and delay estimations as a function of p and m, and particularized for the case $p = 16$, $m = 16$.

We have mapped the different 1-digit adders into the available hardware resources of Virtex-5/6 to obtain numerical area and delay estimations. We use the configurable logic blocks (CLBs) to implement different logic and arithmetic circuits. Each CLB element contains a pair of independent slices with fast carry chains and is connected to a switch matrix for routing. The architecture of a Virtex-5/6 generic slice is shown in Fig. 2. A slice contains four LUTs (look-up tables), four storage elements, wide-function multiplexers, and carry logic. A Virtex-5/6 LUT can implement any boolean function of 6 inputs (output O6) or two boolean functions up to 5 shared inputs (outputs O5, O6). The storage elements can be configured as edge-triggered D-type flip-flops, level-sensitive latches or even as simple logic gates. The hardware cost of a design is given as the number of equivalent 6-input LUTs (LUT-6) used in its implementation. We express the total delay of a design as a function of two elementary slice delays: the delay of a 4-bit carry chain (t_c, horizontal path in Fig. 2) and the delay of a 1-bit sum (t_s, vertical path in Fig. 2), with $t_s >> t_c$. In Table 2 we

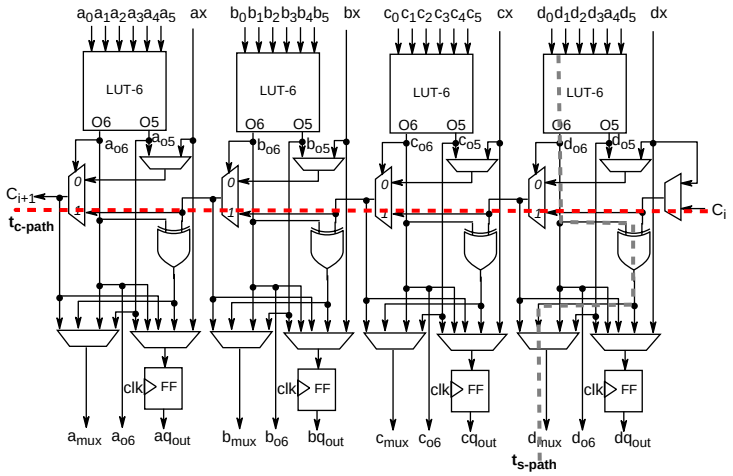


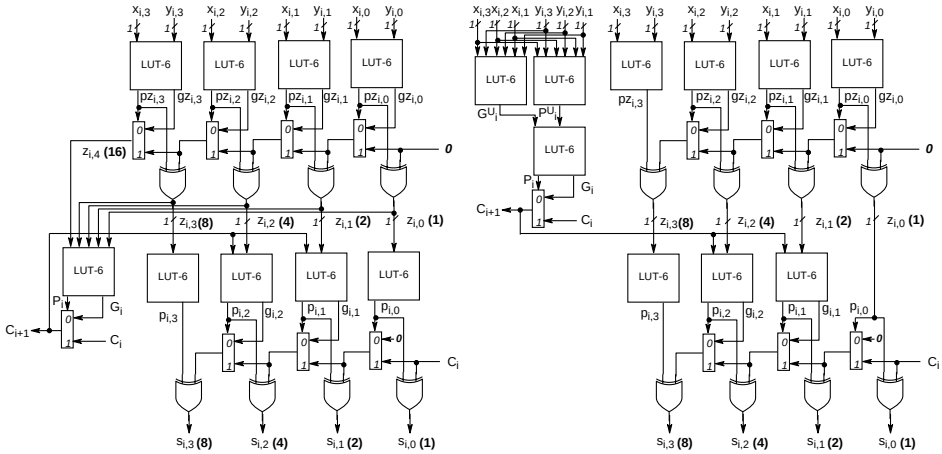
Figure 2: Block diagram of a Virtex-5/6 Slice.

have used values $t_s = 0.94ns$ and $t_c = 0.06ns$, obtained for Virtex-6 with speed grade -3. We have not considered neither storage element delays nor routing delays in this evaluation model.

The multi-operand adder tree implemented by Erle et al. in a decimal multiplier [14] uses the decimal signed-digit adder proposed by Svoboda [29]. The resulting implementation of this digit adder in a Virtex-5/6 is complex and uses more logic (16 LUTs) than the BCD carry-ripple or carry-save digit adders. The decimal carry-save adder trees analyzed [21, 31] require less logic, but still have many interconnections which complicate the FPGA implementation and routing. An alternative is to implement a BCD carry-propagate adder tree that can make use of the fast carry chain. In Fig. 3 we show the Virtex-6 implementation of the BCD carry-chain adders proposed by Bioul et al. [3], based on direct decimal addition methods. The low-cost ($hc_{cca1} = 8$) 1-digit adder is shown in Fig 3a. We also implemented the final BCD carry-propagate adders in the carry-free adder trees [14, 21, 31] using this cell. A faster configuration but with a larger hardware cost ($hc_{cca2} = 10$) is shown in Fig. 3b. However, the $p-digit$ BCD carry-chain adders require more than double the hardware ($8p$ or $10p$ LUTs) of an equivalent binary carry-ripple adder ($3.32p$ LUTs) [3]. 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 mitigate the two previous drawbacks we developed a BCD carry-propagate addition algorithm that can be implemented in a binary carry-ripple fashion. We present this algorithm in the next Section. It leads to a more efficient implementation of BCD addition in FPGAs. Besides, the resultant tree implementations in Virtex-5/6, shown in Section 4, present many similarities with a binary carry-ripple adder tree, so they can be pipelined in a similar way using existing techniques [11].

[3] A p-digit BCD adder has the same input range as a binary $\lceil \log_2(10) \rceil \times p$-bit adder, with $\lceil \log_2(10) \rceil \approx 3.32$.



(a) Low-cost version (b) Fast version

Figure 3: BCD Carry-Chain Adders (1-digit). Prior Art [3]

3 Proposed BCD addition method

Unlike direct decimal addition methods, the methods based on pre- and post-corrections of the binary sum (see Fig. 1) can make full use of any binary adder, for example a binary carry-ripple adder. However, the area and latency overhead due to these decimal corrections is not negligible and the resulting implementations are as complex as the BCD carry-chain adders proposed by Bioul et al. [3]. The method proposed in this work also obtains the decimal sum from the binary addition of the BCD input operands, but the post-correction stage is simplified or removed at the expense of a slightly complex pre-correction stage. Besides, another advantage of our proposal is that the pre-correction stage can be integrated with the binary carry-ripple addition logic in Virtex-5/6 FPGAs at the cost of an extra 6-input LUT per digit. As we show later in Section 4, each 1-digit (4-bit) adder occupies the area of 4-bit binary carry-ripple adder (one Virtex-5/6 slice) and an additional 6-input LUT.

The algorithm is defined for two BCD input operands but can be applied recursively to more BCD addends. An example of addition for three BCD operands X, Y, W is shown in Fig. 4. First, BCD operands X, Y are added as $Z = X + Y$ in three steps. The two first are pre-correction steps and the third is the subsequent binary carry-propagate addition. The pre-correction consists on conditional addition of $+6$ (0110_2) factors to the input BCD operands in a digit-wise fashion. These $+6$ corrections are needed to get the correct decimal carry outputs at each digit position since the BCD addition is computed as a binary carry-propagate addition. In the previous methods [2, 15, 16, 17], the $+6$ factors were added independently of the value of the input operands, such that the binary sum had to be corrected at each decimal position when the corresponding carry-out was zero.

In our method, a correction factor of 6 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 sum of the 3 most significant bits of X_i and Y_i respectively. To check this

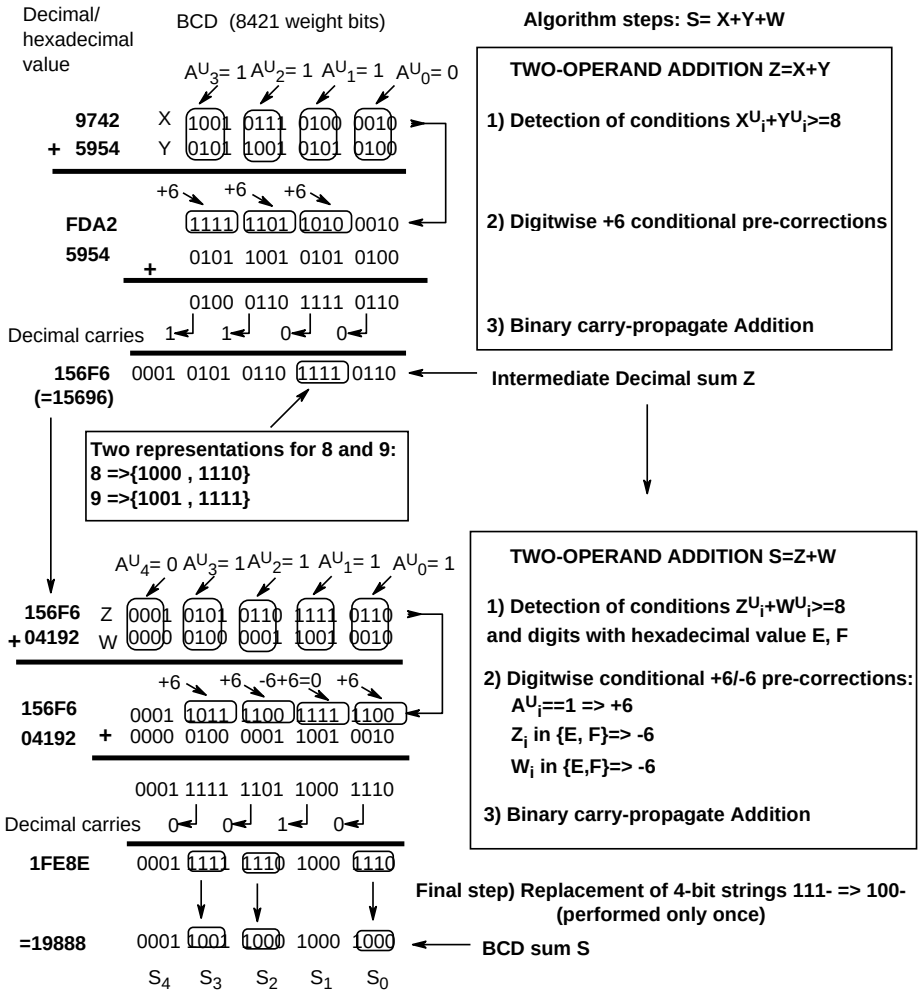


Figure 4: Example of BCD addition using the proposed algorithm.

condition we implement the boolean function A_i^U, defined as

$$A_i^U = \begin{cases} 1 & \texttt{If } X_i^U + Y_i^U \geq 8 \\ 0 & \texttt{Else} \end{cases} \tag{7}$$

Finally, each decimal sum digit as the 4-bit binary sum

$$Z_i = \begin{cases} (X_i + Y_i + 6 + C_i) mod 16 & \texttt{If } A_i^U == 1 \\ (X_i + Y_i + C_i) mod 16 & \texttt{Else} \end{cases} \tag{8}$$

The different values of Z_i and C_{i+1} as functions of X_i, Y_i and C_i are shown in Table 2.

If $X_i^U + Y_i^U > 8$, the decimal carry output C_{i+1} is equal to 1, so in this case the $+6$ corrective factor produces the correct BCD sum digit at the output of the binary addition Z_i. If $X_i^U + Y_i^U < 8$, the decimal carry output C_{i+1} is always zero and the pre-corrective factor $+6$ is not added to position i. The

Z_i	C_{i+1}	$X_i + Y_i$	$X_i^U + Y_i^U$	A_i^U	C_i
0 to 9	0	0 to 8	0 to 6	0	$\{0, 1\}$
0 to 9	1	10 to 18	10 to 16	1	$\{0, 1\}$
$\{14, 15\}$	0	$\{8, 9\}$	8	1	0
15	0	8	8	1	1
0	1	9	8	1	1
$\{0, 1\}$	1	10	8	1	$\{0, 1\}$

Table 2: Different cases for Z_i and C_{i+1}.

correct BCD sum digit is also given by Z_i. However, the BCD sum digit i does not correspond with Z_i when $X_i^U + Y_i^U == 8$ and C_{i+1} is zero. In this case a corrective factor +6 was added in excess since the carry output C_{i+1} is zero, so $Z_i \in \{14\ (1110)_2, 15\ (1111)_2\}$ instead of the correct BCD sum digits $\{8\ (1110)_2, 9\ (1001)_2\}$.

To avoid post-corrections between the different levels of two-operand adders, we allow 4-bit combinations $\{1110_2, 1111_2\}$ to be valid representations of decimal values 8 and 9 respectively. Thus, intermediate decimal sums use 4-bit vectors $\{1000_2, 1110_2\}$ to represent 8, and 4-bit vectors $\{1001_2, 1111_2\}$ to represent 9. The subsequent two-operand BCD addition $S = Z + W$ is modified to detect both 4-bit vectors values for 8 and 9 in parallel with conditions $Z_i^U + W_i^U \geq 8$ as we show in the bottom half of Fig. 4. As before, the decimal sum digits are computed as the 4-bit binary sums

$$S_i^* = \begin{cases} (Z_i + W_i + 6 + C_i) mod 16 & \texttt{If } Z_i^U + W_i^U \geq 8 \\ (Z_i + W_i + C_i) mod 16 & \texttt{Else} \end{cases} \tag{9}$$

Once all the BCD inputs operands have been reduced to one decimal operand S^*, the 4-bit binary strings $S_i^* \in \{1110_2, 1111_2\}$ must be replaced by the BCD digits $\{1000_2, 1001_2\}$ respectively. The BCD final sum digits S_i are obtained from the 4-bit vectors S_i^* in a single final step as

$$S_i = s_{i,3}^*\ 8 + (s_{i,2}^* \overline{s_{i,3}^*})\ 4 + (s_{i,1}^* \overline{s_{i,3}^*})\ 2 + s_{i,0}^* \tag{10}$$

As we show in Section 4, to implement this transformation we configure two storage elements of the Virtex-5/6 slice as logical AND gates with an inverted input for bit $s_{i,3}^*$.

4 Virtex-5/6 implementations

In this Section we detail the Virtex-5/6 implementation of the proposed BCD carry-ripple adder tree for multioperand decimal addition. We present the combinational architecture in Section 4.1 and a fully pipelined design in Section 4.2.

4.1 Combinational architecture

The block diagram of the combinational architecture that implements equation (3) is shown in Fig. 5. It consists of a tree of $\lceil \log_2(m) \rceil$ levels of BCD carry-

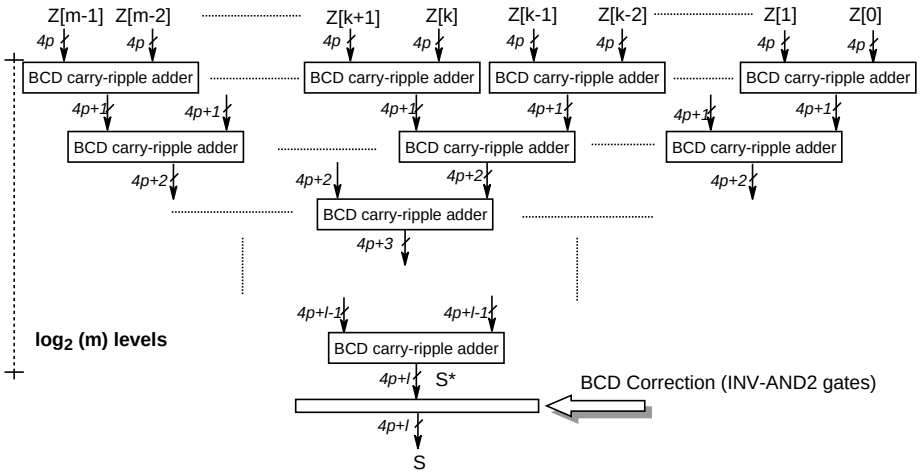


Figure 5: Block Diagram of a Carry-Ripple Adder Tree.

ripple adders that reduces m BCD operands $Z[k]$ into a final BCD sum S. The width of each BCD adder in the first row of the tree is p digits (or $4p$ bits). To avoid sum overflows, we need to extend the width of each BCD adder an appropriate amount. For example, the width of the final adder requires to be incremented in at least $\lceil \log_{10} m \rceil$ digits or l bits, with l given by

$$l = 4\lfloor \log_{10} m \rfloor + \lceil \log_2(m/10^{\lfloor \log_{10} m \rfloor}) \rceil \tag{11}$$

The result of each two-operand BCD sum is passed to an adder a level down and the carry is ripple through length of the adder. However, since the carry-ripple chains overlap, the signals are propagated as much as $4p + l$ bits in length plus $\lceil \log_2(m) \rceil$ levels in depth.

Each BCD carry-ripple adder adds two BCD operands and produce an intermediate decimal sum as described in Section 3. To implement efficiently the sum digit Equation (8), we merged the computation of the pre-correction steps (computation of A_i^U and conditional $+6$ addition) with the 4-bit binary carry-ripple sum. The resultant Virtex-5/6 implementation of 1-digit BCD carry-ripple adder is shown in Fig. 6. This sum cell adds two decimal digits X_i, Y_i, and a carry input C_i, and produces a decimal digit Z_i and a carry output C_{i+1}. The decimal digits are represented in the extended BCD code of Table 3, though in practice digits 8 and 9 only have the following two representations: $\{1000_2, 1110_2\}$ for 8, and $\{1001_2, 1111_2\}$ for 9.

The hardware cost of this 1-digit BCD adder is of five 6-input LUTs (1 slice and a LUT). The 6-input LUTs compute the binary carry-generate ($g_{i,j}$) and carry-propagate ($p_{i,j}$) signals required to implement Z_i (Equation (8)). The boolean expressions for the binary carry-generate and carry-propagate signals are given by:

$$\begin{aligned}
g_{i,3} &= x_{i,3}y_{i,3} \vee x_{i,3}(y_{i,2} \vee y_{i,1}) \vee y_{i,3}(x_{i,2} \vee x_{i,1}) \vee x_{i,2}y_{i,2}(x_{i,1} \vee y_{i,1}) \\
g_{i,2} &= 0 \\
g_{i,1} &= 0 \\
g_{i,0} &= x_{i,0}y_{i,0}
\end{aligned}$$

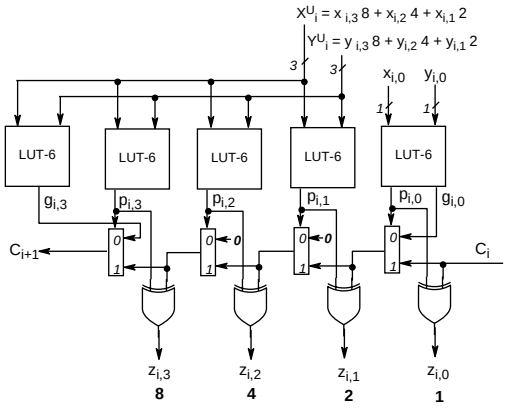


Figure 6: Proposed BCD Carry-Ripple Adder (1 digit).

Decimal	BCD	Extended BCD
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1**0
9	1001	1**1

* can be either 0 or 1.

Table 3: BCD codings

$$
\begin{aligned}
p_{i,3} &= (x_{i,3} \oplus y_{i,3})\overline{[x_{i,2} \vee y_{i,2} \vee x_{i,1} \vee y_{i,1}]} \vee \overline{x_{i,3} y_{i,3}}[x_{i,2} y_{i,2} \overline{x_{i,1} y_{i,1}} \\
&\quad \vee (x_{i,2} \oplus y_{i,2}) x_{i,1} y_{i,1}] \vee x_{i,3} \overline{y_{i,3} y_{i,2} y_{i,1}} \vee y_{i,3} \overline{x_{i,3} x_{i,2} x_{i,1}} \\
p_{i,2} &= (\overline{y_{i,3}} y_{i,2}) \oplus (\overline{x_{i,3}} x_{i,2}) \oplus [x_{i,1} y_{i,1} \overline{A_i^U} \vee (x_{i,3} \vee \overline{x_{i,1}})(y_{i,3} \vee \overline{y_{i,1}}) A_i^U] \\
p_{i,1} &= (\overline{y_{i,3}} y_{i,1}) \oplus (\overline{x_{i,3}} x_{i,1}) \oplus A_i^U \\
p_{i,0} &= x_{i,0} \oplus y_{i,0}
\end{aligned} \tag{12}
$$

with

$$A_i^U = x_{i,3} \vee y_{i,3} \vee x_{i,2} y_{i,2} \vee (x_{i,2} \vee y_{i,2}) x_{i,1} y_{i,1} \tag{13}$$

The logic complexity and logic depth of the proposed BCD sum cell are close to the values of the 4-bit binary carry-ripple adder for Virtex-5/6.

The final BCD correction block, which implements Expression (10), is detailed in Fig. 7 (1-digit slice).

We have estimated the area and delay of the proposed BCD adder tree using the model for the Virtex-5/6 slice described in Section (2.2). We have not included in this rough estimation the hardware required to avoid sum overflows. The critical path delay is given by

$$t_{tree} = t_c \times p + t_s \times \lceil \log_2(m) \rceil \tag{14}$$

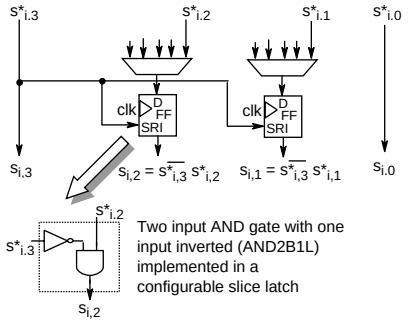


Figure 7: Block Diagram of 1-digit correction.

where t_c is the delay of the carry chain of a 1-digit (4-bit) BCD adder and t_s the delay of the sum path of the 1-digit BCD adder. The equivalent hardware cost of each adder is 5 LUTs per digit. The total hardware cost of the BCD adder tree is $5 \times p \times (m-1)$ LUTs. The figures for $p = m = 16$ are shown in Table 1. The delay of the Virtex-6 implementation with speed grade -3 is estimated in 4.72 ns and requires about 1200 6-input LUTs. When compared to a adder tree implemented with the BCD Carry-Chain Adder cells [3] of Fig. 3a the area and speed are improved a factor 1.6 and 1.9 respectively.

4.2 Pipelined architecture

We have fully pipelined the combinational architecture as follows: each level of the adder tree is placed in a pipeline stage. Besides, each carry-ripple adder is pipelined in chunks of k bits at most. The total number of pipelined stages is equal to $\lceil (4p+l)/k \rceil + \lceil \log_2(m) \rceil$. In Fig. 8 we show a BCD adder tree with this register placement for chunks of $k = p$ bits.

A significant amount of registers is required for input synchronization [11]. 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.

5 Synthesis Results and Comparison

Combinational and pipelined versions of the BCD carry-ripple adder tree presented in Section 4 were designed in VHDL for arbitrary values m, p and k using the Xilinx ISE Design Suite 11.4. To have more control over the mapping process the different adder cells were directly instantiated into components of the Xilinx Virtex-6 library. 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.

From the survey presented in Section 2 we determined that the most suitable structures for fast BCD multi-operand addition in FPGAs are trees built of carry-ripple or carry-chain adders. To extract more precise conclusions from comparison, we coded and synthesized two different adder trees: a binary carry-ripple adder tree and a tree built of the BCD carry-chain adders shown in Fig.3 (Bioul et al. [3]).

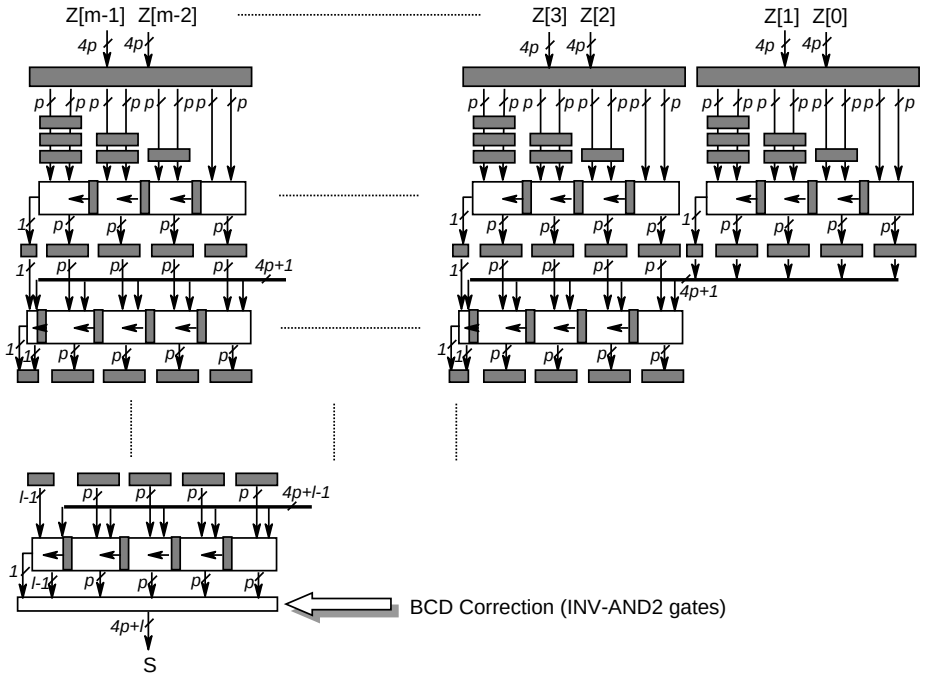


Figure 8: Detail of a pipelined BCD Multioperand Adder Tree ($k = p$).

	$p = 16$, $n = 54$		$p = 34$, $n = 113$	
Adder Type	Cost (# LUT-6)	Delay (ns)	Cost (# LUT-6)	Delay (ns)
BCD Carry-Chain (Low area) [3]	128	2.66	272	2.93
BCD Carry-Chain (Fast version) [3]	160	2.51	340	2.78
BCD Carry-Ripple (proposed)	80	2.17	170	3.25
Binary Carry-Ripple (n)	54	1.75	113	2.63
Binary Carry-Ripple ($4p$)	64	1.90	136	2.98

Table 4: Synthesis results for two-operand carry-ripple or carry-chain adders.

In Table 4 we show the area and delay obtained from the synthesis of different two operand ($m = 2$) carry-propagate adders. The hardware cost is expressed in terms of equivalent LUT-6 units and the delay in ns. We obtained results for the basic operand sizes specified in the IEEE 754-2008 standard: $p = 16$ and $p = 34$. In the case of binary operands, we considered n-bit operands with the same input range as in the decimal case: $n = \lceil 3.32 \times p \rceil$ bits, that is, $n = 54$ and $n = 113$. We have also included in the comparison binary operands of $4p$ bit length, more specifically of 64 and 136 bits length.

The proposed BCD carry-ripple adder practically halves the hardware cost of the BCD carry-chain adder and is still very competitive in terms of speed. The BCD carry-chain adders have speed advantages for large carry chains at expense of high hardware costs. However, as we show next, this advantage is not observed in a multi-operand adder tree, due to the overlap of the carry chain propagations. Table 5 shows the area and delay results from the synthesis

(p,m,n)	(16,16,54)		(34,34,113)	
Adder Type	Cost (# LUT-6)	Delay (ns)	Cost (# LUT-6)	Delay (ns)
BCD Carry-Chain (Low area) [3]	1976	9.10	9136	13.75
BCD Carry-Chain (Fast version) [3]	2470	8.28	11420	12.50
BCD Carry-Ripple (proposed)	1235	5.22	5710	8.62
Binary Carry-Ripple (n)	821	3.90	3769	6.23
Binary Carry-Ripple ($4p$)	971	4.05	4519	6.57

Table 5: Synthesis results for m-operand carry-ripple or carry-chain adders.

k (bits)	#pipe. stages	Clock Freq. (Mhz)	Delay latency× #stages	Hardware Cost #LUT-6	#FF
32	6	560 Mhz	10.7 ns (1.79 × 6)	2259	2042
24	7	621 Mhz	11.3 ns (1.61 × 7)	2259	2057
16	8	646 MHz	12.4 ns (1.54 × 8)	2259	2072
12	10	700 Mhz	14.3 ns (1.43 × 10)	2259	2102

Table 6: Synthesis results for the pipelined BCD adder tree ($m = p = 16$).

k (bits)	#pipe. stages	Clock Freq. (Mhz)	Delay latency× #stages	Hardware Cost #LUT-6	#FF
27	6	712 Mhz	8.4 ns (1.40 × 6)	1685	1715
18	7	797 Mhz	8.8 ns (1.25 × 7)	1685	1730
14	8	837 MHz	9.6 ns (1.20 × 8)	1685	1745
11	9	850 Mhz	10.6 ns (1.18 × 9)	1685	1760

Table 7: Synthesis results for the pipelined binary adder tree ($m = 16$,$n = 54$).

of different combinational carry-ripple and carry-chain adder trees. We have particularized the results for $m = 16$ and $m = 34$ input operands. Apart form a reduced hardware cost, the proposed BCD carry-ripple adder trees also have significant speed advantages with respect to the BCD carry-chain adder trees.

We have pipelined the combinational BCD carry-ripple adder tree as described in Section for $p = m = 16$ and different values of k. The synthesis results are presented in Table 6. We also pipelined a multioperand binary carry-ripple adder tree for $n = 54$ and $p = 16$ in the same way. The synthesis results are presented in Table 7. The BCD design has 30% more area and about 20% more latency than the binary carry-ripple tree.

6 Conclusions

In this work we presented the design of a BCD multi-operand adder and its implementation on a Virtex-5/6 FPGA. We performed a survey of prior techniques of BCD muti-operand addition to find the most area-efficient low-latency FPGA implementation. From this survey we found that tree structures build of

BCD carry-ripple or carry-chain adders are suitable for state-of-the-art FPGA implementations.

To improve the efficiency of Virtex-5/6 implementations we have developed a new algorithm for BCD carry-propagate addition. Combinational and pipelined versions of the BCD multi-operand adder were synthesized in a Virtex-6 speed grade-3 device and the results compared with a binary carry-ripple adder tree and a BCD multi-operand adder tree build of BCD carry-chain adders. 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.

As future work we plan to integrate the proposed multi-operand BCD adder in a core of the FloPoCo project[4], a generator of arithmetic cores for FPGAs, and to support more FPGA targets than Virtex-5/6 families.

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. J. Adiletta and V. C. Lamere, *BCD Adder Circuit*, US patent 4,805,131, Jul 1989.

[2] D. P. Agrawal, *Fast BCD/Binary Adder/Subtractor*, Electronics Letters, vol. 10, no. 8, pp. 121–122, Apr. 1974.

[3] G. Bioul, M. Vazquez, J.-P. Deschamps, G. Sutter, *Decimal Addition in FPGA*, V Southern Conference on Programmable Logic (SPL09), pp. 101–108, Apr. 2009.

[4] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz and S. R. Carlough, *The IBM z900 Decimal Arithmetic Unit*, Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1335–1339, Nov. 2001.

[5] G.C. Cardarilli, S. Pontarelli, M. Re and A. Salsano, *On the use of Signed Digit Arithmetic for the new 6-Inputs LUT based FPGAs*, 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 602–605, Sep. 2008.

[6] M. A. Check, T. J. Slegel, *Custom S/390 G5 and G6 Microprocessors*, IBM Journal Research and Development, vol.43, no.5/6, pp. 671–680, Sept./Nov. 1999.

[7] D.S. Cochran, *Algorithms and Accuracy in the HP-35*, Hewlett-Packard Journal, pp. 10–11, 1972.

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

[8] 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 Transactions on Computers, vol. 58, no. 2, pp. 148–162, Feb. 2009.

[9] M. Cowlishaw, *Decimal Floating-Point: Algorism for Computers*, 16th IEEE Symposium on Computer Arithmetic, pp. 104–111, July 2003.

[10] L. Dadda, *Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach*, IEEE Transactions on Computers, vol. 56, no. 10, pp. 1320–1328, Oct. 2007.

[11] F. de Dinechin, H. D. Nguyen and B. Pasca, *Pipelined FPGA Adders*, LIP Research Report no. ensl-00475780, Apr. 2010.

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

[13] M. A. Erle and M. J. Schulte, *Decimal Multiplication Via Carry-Save Addition*, Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 348–358, Jun. 2003.

[14] M. A. Erle, E. M. Schwarz and M. J. Schulte, *Decimal Multiplication With Efficient Partial Product Generation*, 17th IEEE Symposium on Computer Arithmetic, pp. 21–28, Jun. 2005.

[15] H. Fischer and W. Rohsaint, *Circuit Arrangement for Adding or Subtracting Operands in BCD-Code or Binary-Code*, US Patent 5,146,423, Sep. 1992.

[16] U. Grupe. *Decimal Adder*, US Patent 3,935,438, Jan. 1976.

[17] W. Haller, U. Krauch, and H. Wetter, *Combined Binary/Decimal Adder Unit*, US patent 5,928,319, Jul. 1999.

[18] J. Hormigo, M. Ortiz, F. Quiles, F. J. Jaime, J. Villalba and E.L. Zapata, *Efficient Implementation of Carry-Save Adders in FPGAs*, 20th IEEE international Conference on Application-Specific Systems, Architectures and Processors, pp. 207–210, Jul. 2009.

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

[20] R. D. Kenney and M. J. Schulte, *High-Speed Multioperand Decimal Adders*, IEEE Transactions on Computers, vol. 54, no. 8, pp. 953–963, Aug 2005.

[21] T. Lang and A. Nannarelli, *A Radix-10 Combinational Multiplier*, 40th Asilomar Conference on Signals, Systems, and Computers, pp. 313–317, Oct. 2006.

[22] P. M. Martinez, V. Javier, and B. Eduardo, *On the design of FPGA-based Multioperand Pipeline Adders*, XII Design of Circuits and Integrated System Conference, 1997.

[23] H. Parandeh-Afshar, P. Brisk, and P. Ienne, *Efficient Synthesis of Compressor Trees on FPGAs*, ASPDAC'08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference, pp. 138-143, 2008.

[24] R. K. Richards, *Arithmetic Operations in Digital Computers*, D. Van Nostrand Company. New Jersey, 1955.

[25] M. Schmookler and A. Weinberger, *High Speed Decimal Addition*, IEEE Transactions on Computers, vol. c-20, no. 8, pp. 862–866, Aug. 1971.

[26] E. Schwarz, J. Kapernick and M. Cowlishaw, *Decimal Floating-Point Support on the IBM System z10 processor*, IBM Journal of Research and Development, vol. 51, no. 1, Jan/Feb. 2009

[27] B. Shirazi, D.Y.Y. Yun and C. N. Zhang, *RBCD: Redundant Binary Coded Decimal Adder*, IEE Proceedings - Computers and Digital Techniques", vol. 136, pp. 156–160, Mar. 1989.

[28] D. Strenski, J. Simkins, R. Walke, and R. Wittig, *Evaluating FPGAs for Floating-Point Performance*, Intl. Workshop on High-Performance Reconfigurable Computing Technology and Applications, pp. 1–6, Nov. 2008.

[29] A. Svoboda, *Decimal Adder with Signed-Digit Arithmetic*, IEEE Transactions on Computers, vol. C, pp. 212–215, Mar. 1969.

[30] A. Vazquez and E. Antelo, *Conditional Speculative Decimal Addition*, 7th Conference on Real Numbers and Computers (RNC 7), pp. 47–57, Jul. 2006.

[31] A. Vazquez, E. Antelo and P. Montuschi, *Improved Design of High-Performance Parallel Decimal Multipliers*, IEEE Transactions on Computers, Vol. 59, No. 5, pp. 679–693, May 2010.

[32] Xilinx Inc., *Virtex-6 User Guide*, 2009, http://www.xilinx.com/.

[33] S. Xing and W. H. Yu, *FPGA Adders:Performance Evaluation and Optimal Design*, IEEE Design and Test of Computers, vol. 15, no. 1, pp. 24–29, Jan.-Mar. 1998.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399