Multi-operand redundant adders on FPGAs

Javier Hormigo, Julio Villalba, and Emilio L. Zapata

Abstract—Although redundant addition is widely used to design parallel multi–operand adders for ASIC implementations, the use of redundant adders on Field Programmable Gate Arrays (FPGAs) has generally been avoided. The main reasons are the efficient implementation of carry propagate adders (CPAs) on these devices (due to their specialized carry–chain resources) as well as the area overhead of the redundant adders when they are implemented on FPGAs. This paper presents different approaches to the efficient implementation of generic carry–save compressor trees on FPGAs. They present a fast critical path, independent of bit–width, with practically no area overhead compared to CPA trees. Along with the classic carry-save compressor tree, we present a novel linear array structure which efficiently uses the fast carry–chain resources. This approach is defined in a parameterizable HDL code based on CPAs, which makes it compatible with any FPGA family or vendor. A detailed study is provided for a wide range of bit–widths and large number of operands. Compared to binary and ternary CPA trees, speed-ups of up to 2.29 and 2.14 are achieved for 16 bit–width and up to 3.81 and 3.11 for 64 bit–width.

Index Terms—Computer arithmetic, reconfigurable hardware, multi–operand addition, redundant representation, carry–save adders.

1 Introduction

The use of Field Programmable Gate Arrays (FPGAs) to implement digital circuits has been growing in recent years. In addition to their reconfiguration capabilities, modern FPGAs allow high parallel computing. FPGAs achieve speedups of two orders of magnitude over a general-purpose processor for arithmetic intensive algorithms [1]. Thus, these kinds of devices are increasingly selected as the target technology for many applications, especially in digital signal processing [2] [3] [4] [5], hardware accelerators [6] [7] [8], cryptography [9] [10] and much more. Therefore, the efficient implementation of generalized operators on FPGAs is of great relevance.

The typical structure of an FPGA device is a matrix of configurable logic elements (LEs), each one surrounded by interconnection resources. In general, each configurable element is basically composed of one or several n-input lookup tables (N–LUT) and flip–flops. However, in modern FPGA architectures, the array of LEs has been augmented by including specialized circuitry, such as dedicated multipliers, block RAM, etc. In [11], the authors demonstrate that the intensive use of these new elements reduces the performance GAP between FPGA and ASIC implementations.

One of these resources is the carry–chain system which is used to improve the implementation of carry propagate adders (CPAs). It mainly consists of additional specialized logic to deal with the carry signals, and specific fast routing lines between consecutive LEs, as shown in Fig. 1. This resource is presented in most current FPGA devices from low–cost ones to high-end families and it accelerates the carry propagation by more

• *J. Hormigo, J. Villalba and E. L. Zapata are based at the Department of Computer Architecture, University of Malaga, Malaga, Spain, E-29071. E-mail: fjhormigo@uma.es*

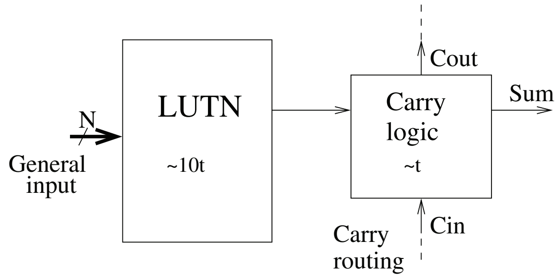


Fig. 1. General scheme of dedicated carry–chain resources included in modern FPGA devices

than one order of magnitude compared to its implementation using general resources. Apart from the CPA implementation, many studies have demonstrated the importance of using this resource to achieve designs with better performance and/or less area requirements, and even for implementing non–arithmetic circuits [12] [13].

Multi-operand addition appears in many algorithms, such as multiplication [14] [15], filters [16] [17], SAD [18] and others [1] [8] [19] [20] [21]. To achieve efficient implementations of this operation, redundant adders are extensively used [22] [23]. Redundant representation reduces the addition time by limiting the length of the carry–propagation chains. The most usual representations are carry–save (CS) and signed–digit (SD). A CS adder (CSA) adds three numbers using an array of Full–Adders (FAs), but without propagating the carries. In this case, the FA is usually known as a 3:2 counter. The result is a CS number, which is composed of a sum–word and a carry–word. Therefore, the CS result is obtained without any carry propagation in the time taken by only one full-adder. The addition of two CS numbers requires an array of 4:2 compressors, which can be implemented by two 3:2 counters. The conversion to non–redundant representation is achieved by adding the sum and carry word in a conventional CPA [24].

However, due to the efficient implementation of CPAs, the use of redundant adders has usually been rejected when targeting FPGA technology. A direct implementation of a 3:2 counter usually doubles the area requirements of its equivalent CPA and improved speed is only noticeable for long bit–widths. Nevertheless, several recent studies have demonstrated that redundant adders can be efficiently mapped on FPGA structures, reducing area overhead and improving speed, as described in Section 2. Despite the important advances represented by these previous studies, the solutions proposed require either (or sometimes both) the use of a sophisticated heuristic to generate each compressor tree or a low-level design. The latter impedes portability, since it is highly dependent on the inner structure. In addition, their area and speed could be improved, since the use of a specialized fast carry–chain is very limited.

In this paper, we study the efficient implementation of multi–operand redundant compressor trees in modern FPGAs by using their fast carry resources. Our approaches strongly reduce delay and they generally present no area overhead compared to a CPA tree. Moreover, they could be defined at a high level based on an array of standard CPAs. As a consequence, they are compatible with any FPGA family or brand, and any improvement in the CPA system of future FPGA families would also benefit from them. Furthermore, due to its simple structure, it is easy to design a parametric HDL core which allows synthesizing a compressor tree for any number of operands of any bit–width. Compared to previous approaches, our design presents better performance, is easier to implement, and offers direct portability.

The rest of the paper focuses on CS representation, since the extension to SD representation could be simply achieved by inverting certain input and output signals from and to the compressor tree, as was demonstrated in [25]. Since it is unnecessary to make any internal changes to the array structure, these small modifications do not significantly modify compressor tree performance.

The remainder of this paper is organized as follows: Section 2 reviews previous work on redundant addition on FPGAs. In Section 3, we present our proposals for implementing multi–operand redundant compressor trees on FPGAs and a theoretical analysis of their performance. In Section 4, we compare the results of implementation using different approaches. Finally, the conclusions are presented in Section 5.

2 Related works

The optimization of redundant addition on FPGAs has been addressed using different approaches: i) the efficient mapping of isolated redundant adders on an inner structure of FPGAs [26] [27] [28] [29]; ii) utilizing different heuristics to design compressor trees based on bit counters [30] [31] [32] [33]; iii) proposing hardware modifications to existing FPGA architectures [34] [35] [36] [37] [38]; and iv) specific applications [39] [14]. Next, we summarize the papers more closely related to our work.

In [26], the use of high-radix carry–save representation is proposed, i.e. converting long CPAs into the additions of short digits. However, this unconventional representation has important limitations; for example, bit shifting is not allowed. In [27], the implementation of a radix-4 signed digit adder on 6–LUT based FPGAs is addressed, but carry resources are not used and the area overhead is still very high (88% more logic resources). Low-level designs, i.e. using and directly configuring FPGA primitives, are used in [28] and [29] to map classic binary redundant compressors on 4–LUT based FPGAs using carry resources. Both studies conclude that a 4:2 compressor obtains the best performance and produces no area overhead on Xilinx FPGAs. All of these studies only focus on the optimization of isolated adders, but they do not address the issue of compressor tree design. Nevertheless, they could be used as basic building blocks to construct classic compressor trees, as shown in Section 3.1.

Parallel multi-operand addition is addressed for 6–LUT–based FPGAs by Parandeh–Afshar et al. in [30] [31] [32] and by Matsunaga et al. in [33]. All of these studies present compressor tree designs based on Generalized Parallel Counters (GPCs) [40] [41]. First, several GPC sizes are suitably selected and characterized to efficiently use the inner resources of the target FPGA. Second, each study proposes a different algorithm to build the specific compressor tree based on a network of GPCs in such a way that an attempt is made to minimize the critical path and/or the area of the tree. With the exception of [31], the GPCs are implemented using the general logic resource alone. In general, they report considerably reduced delays and a moderate increase in area compared to ternary CPA trees. Their main drawbacks are that they are not valid for 4–LUT-based FPGAs, they yield unpredictable results, and they require the use of software to design each specific compressor tree.

3 Carry–save compressor trees on FPGAs

In this section, we present different approaches to efficiently map carry–save compressor trees on FPGA devices. In addition, approximate area and delay analysis are conducted for the general case. A more accurate analysis for specific examples is provided in Section 4.

Let us consider a generic compressor tree of N_{op} input operands with N bit–width each. We also assume the same bit–width for input and output operands. Thus, input operands should have previously been zero or sign extended to guarantee that no overflow occurs. A detailed analysis of the number of leading guard bits required for multi–operand carry–save addition is provided in [42].

3.1 Regular carry–save compressor tree design

The classic design of a multi–operand carry–save compressor tree attempts to reduce the number of levels in its structure. The 3:2 counter or the 4:2 compressor are the most widely known building blocks to implement it [43]. We select a 4:2 compressor as the basic building block, since it could be efficiently implemented on Xilinx FPGAs [28].

The implementation of a generic carry–save compressor tree requires $\lceil N_{op}/2 \rceil - 1$ 4:2 compressors (since each one eliminates two signals), whereas a carry-propagate tree uses $N_{op} - 1$ CPAs (since each one eliminates one signal) [24]. If we bear in mind that a 4:2 compressor uses practically double the amount of resources as CPAs [28], both trees basically require the same area.

On the other hand, the speed of a compressor tree is determined by the number of levels required. In this case, since each level halves the number of input signals, the critical path delay (D) is approximately

$$L_{4:2} = \lceil \log_2(N_{op}) \rceil - 1 \tag{1}$$

$$D \approx L_{4:2} \cdot d_{4:2} \tag{2}$$

where $L_{4:2}$ is the number of levels of the compressor tree and $d_{4:2}$ is the delay of a 4:2 compressor level (including routing).

This structure is constructed assuming a similar delay for all paths inside each 4:2 compressor. Nevertheless, in FPGA devices with dedicated carry resources, the delay from the carry input to the carry output and the routing to the next carry input is usually more than one order of magnitude faster than the rest of the paths involved in connecting two FAs (see Fig. 1). Thus, the connection of FAs through the carry–chain should be preserved as much as possible to obtain fast circuits. In fact, this is the idea behind the structure of the 4:2 compressor presented in [28] and [29] for Xilinx FPGA. We now generalize this idea to compressors of any size by proposing a different approach based on linear arrays. This reduces the critical path of the compressor tree when it is implemented on FPGAs with specialized carry–chains.

3.2 Linear array structure

In the previous approach, specialized carry resources are only used in the design of a single 4:2 compressor, but these resources have not been considered in the design of the whole compressor tree structure. To optimize the use of the carry resources, we propose a compressor tree structure similar to the classic linear array of CSAs [24]. However, in our case, given the two output words of each adder (sum–word and carry–word), only the carry–word is connected from each CSA to the next, whereas the sum–words are connected to lower levels of the array.

Fig. 2 shows an example for a 9:2 compressor tree designed using the proposed linear structure, where all lines are N bit–width buses, and carry signal are

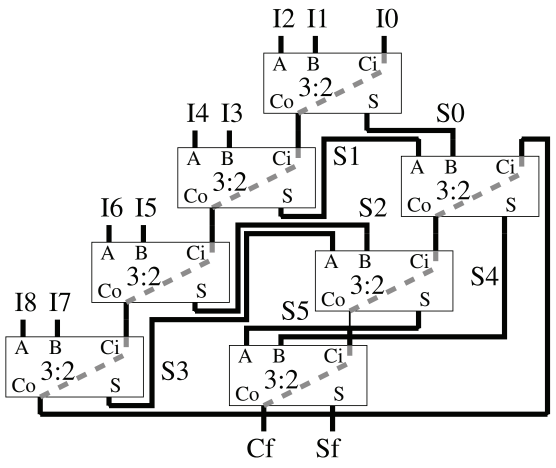


Fig. 2. N bit–width carry–save 9:2 compressor tree based on a linear array of CSAs

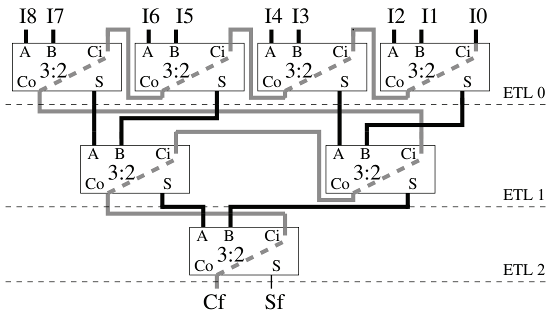


Fig. 3. Time model of the proposed carry-save 9:2 compressor tree

correctly shifted. For the CSA, we have to distinguish between the regular inputs (A and B) and the carry input (Ci in the figure), whereas the dashed line between the carry input and output represents the fast carry resources. With the exception of the first CSA, where Ci is used to introduce an input operand, on each CSA Ci is connected to the carry output (Co) of the previous CSA, as shown in Fig. 2. Thus, the whole carry–chain is preserved from the input to the output of the compressor tree (from I0 to Cf). First, the two regular inputs on each CSA are used to add all the input operands (Ii). When all the input operands have been introduced in the array, the partial sum–words (S_i) previously generated are then added in order (i.e. the first generated partial sums are added first) as shown in Fig. 2. In this way, we maximize the overlap between propagation through regular signals and carry–chains.

Regarding the area, the implementation of a generic compressor tree based on N bit–width CSAs requires $N_{op} - 2$ of these elements (since each CSA eliminates one input signal) [24]. Therefore, considering that a CSA could be implemented using the same number of resources as a binary CPA (as shown below), the proposed linear array, the 4:2 compressor tree, and the binary CPA tree have approximately the same hardware cost.

In relation to the delay analysis, from a classic point of view our compressor tree has $N_{op} - 2$ levels. This is much more than a classic Wallace tree structure and thus a longer critical path. Nevertheless, since we are targeting an FPGA implementation, we temporarily assume that there is no delay for the carry–chain path. Under this assumption, the carry signal connections could be eliminated from the critical path analysis and our linear array could be represented as a hypothetical tree, as shown in Fig. 3 (where the carry–chain is represented in gray). To compute the number of effective time levels (ETL) of this hypothetical tree, each CSA is considered a 2:1 adder, except for the first, which is considered a 3:1 adder. Thus, the first level of adders is formed by the first $\lfloor (N_{op} - 1)/2 \rfloor$ CSAs (which correspond to partial addition of the input operands). This first ETL produces $\lfloor (N_{op} - 1)/2 \rfloor$ partial sum–words which are added to a second level of CSAs (together with the last input operand if N_{op} is even) and so on, in such a way that each ETL of CSAs halves the number of inputs to the next level. Therefore, the total ETLs in this hypothetical tree are

$$L = \lceil \log_2(N_{op} - 1) \rceil \tag{3}$$

and the delay of this tree is approximately L times the delay of a single ETL.

However, the delay of the carry–chain is comparatively low, but not null. Let us consider just two global values for the delay: d_{carry}, which is the delay for the path between the carry inputs (Ci) of two consecutive CSAs (see Fig. 3); and d_{sum}, which is the delay from one general input of a CSA (A or B) to a general input of a directly connected CSA, i.e. the time taken by the data to go from an ETL to the next one (see Fig. 3). Even under this simplified scenario, it is unfeasible to obtain a general analytical expression for the delay of our compressor tree structure. On each ETL, the propagation through carry–chains and the general paths are overlapped and this overlap depends on multiple factors. First, it depends on the relative relationship between the values of d_{carry} and d_{sum} (which is associated with the FPGA family used). Second, it depends on the number of operands which affect both the delay of the carry–chain of each ETL and the internal structure of the hypothetical tree. Even though the former could be expressed as an analytical formula, the latter cannot be expressed in this way (especially when $N_{op} - 1$ is not a power of two). However, it is possible to bound the critical path delay by considering two extreme options.

One extreme situation occurs when the delay of the whole carry–chain corresponding to each ETL ($d_{carry} \times$ the number of CSAs of the ETL) is always greater than the delay from an ETL to the next one (d_{sum}). In this case, the timing behavior corresponds to a linear array and the critical path is represented in Fig. 4. Initially, the first carry out signal is generated from $I1$, $I2$, $I3$ in the first CSA and then the carry signal is propagated through the whole carry–chain until the output. Thus, the delay of the critical path has two components corresponding

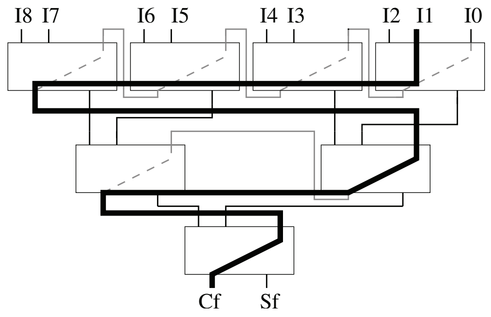


Fig. 4. Critical path of the proposed 9:2 compressor tree for linear array behavior

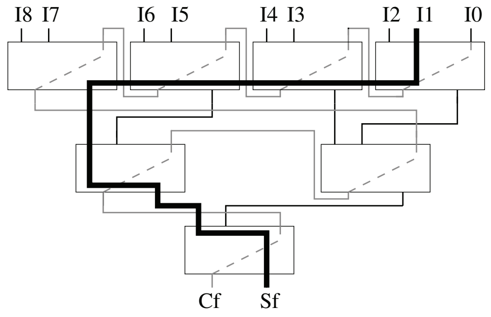


Fig. 5. Critical path of the proposed 9:2 compressor tree for tree behavior

to the generation of the first carry signal and the propagation through the carry–chain. If we characterize the delay from a general input to the carry output in the first CSA (including later routing) as d_{sum}, then the estimated lower bound for the delay of the compressor tree is

$$D_{low} \approx d_{sum} + (N_{op} - 3) \cdot d_{carry} \tag{4}$$

As mentioned, d_{sum} is usually one order of magnitude greater than d_{carry}. Thus, the initial condition could be partially fulfilled for compressor trees with high N_{op}, but only in the first ETLs, since the number of CSAs on each ETL is nearly halved compared to the previous one. In fact, since the last ETL has only one CSA (ETL0 in Fig. 2), this condition (i.e. in all ETLs) can only be completely fulfilled if $d_{sum} < d_{carry}$, which is not possible on FPGAs. Therefore, although equation 4 is an underestimated lower bound, it reflects the behavior of the first ETLs for high N_{op}, as described below.

The other extreme situation occurs when the delay of the carry–chain on each ETL ($d_{carry} \times$ the number of CSAs of the ETL) is always less than the delay from an ETL to the next one (d_{sum}). In this case, we obtain the hypothetical tree presented in Fig. 3. The critical path is shown in Fig 5. It begins as in the previous case: a first carry generation from the general inputs and carry propagation through the carry–chain of the first ETL, since all internal general paths of the CSAs corresponding to the first ETL are updated in parallel and they need the carry input to generate the output signals. However, due to the initial premise, and since sum signals arrive at the first CSAs of each ETL earlier than at the last ones, after the first ETL the critical path goes from one ETL to the next one through the general routing. Therefore, the delay of the critical path has three components corresponding to the generation of the first carry signal, propagation through the carry–chain of the first ETL, and propagation across the remaining of the ETLs. In this case, taking into account that the delay between the carry input and the sum output (including later routing) is approximately d_{carry}, the estimated upper bound for the delay of the compressor tree is $d_{sum} \times$ the number of ETLs $+ d_{carry} \times$ the number of CSAs of the first ETL, that is

$$D_{up} \approx \lceil \log_2(N_{op} - 1) \rceil \cdot d_{sum} + \frac{N_{op} - 1}{2} \cdot d_{carry} \quad (5)$$

This scenario is very frequent because if the delay through the carry–chain of the first ETL is less than d_{sum}, then the next ETLs hold this condition. Thus, only the following condition

$$d_{sum} > \frac{N_{op} - 1}{2} \cdot d_{carry} \quad (6)$$

needs to be fulfilled. Therefore, for values of N_{op} up to $2d_{sum}/d_{carry}$, the delay of the linear array compressor tree is very close to D_{up}. However, for greater values of N_{op}, the delay of the compressor tree is between D_{low} and D_{up}, since the hypothetical structure of the compressor tree is a mix of both situations: the first CSAs form a linear array until the delay of the carry–chain in an ETL is lower than d_{sum}, and then the remaining ones form a hypothetical tree.

In any case, the behavior of the delay related to the number of operands has two additive components: i) a discontinuous logarithmic variation due to the number of ETLs of the hypothetical tree; and ii) a linear variation related to the propagation through either the carry–chain of the first ETL (for low N_{op}) or the linear array part (for high N_{op}). In addition, increasing N_{op} causes the growth of the last term, which could occasionally reduce the value of the first term, since first ETLs of the tree part could become part of the linear structure. As a result, logarithmic behavior is dominant for low values of N_{op}, whereas linear behavior prevails for high values. On the other hand, due to the interaction of these two terms, the curve of the delay in relation to N_{op} is smooth, instead of the increment by steps produced in a classic tree, as shown in Section 4.

If we compare this approach to the one based on 4:2 compressors, the hypothetical tree of the former generally has one more level than the latter. In addition, the linear array compressor tree also has a delay associated with the linear component, whereas the 4:2 compressor tree does not. However, since $d_{4:2}$ is significantly greater than d_{sum}, the linear approach is faster than the 4:2 compressor based one (as shown in Section 4).

3.3 High-level implementation

A high-level description of an FPGA design using HDL presents some significant advantages, such as portability among different families (even from different brands), easier generalization, lower error production during the design process, etc. A disadvantage is that control over the final implementation is lost, which could produce unexpected results. Thus, the HDL code needs to be carefully selected, especially when specific inner resources of the FPGA are used. In fact, a low-level design of the basic building block, which is instantiated from the higher level circuit, is sometimes the only way to achieve efficient implementations. Thus, the design is anchored to a specific FPGA inner structure. This is the case of the classic 4:2 compressor tree proposed in Section 3.1 for 4–LUT-based FPGA families.

The linear array proposed in Section 3.2 requires the use of 3:2 counters as a basic building block. However, the commercial software tools do not support high-level CSA implementation and in any case they do not use the carry logic for the implementation of this element. In addition, no efficient low-level implementation of a 3:2 counter has been reported in the literature. Thus, a different approach awaits discovery.

As an example, a more detailed description of the linear array structure for an N bit–width 5:2 compressor tree is shown in Fig. 6. This circuit could be implemented by connecting the FAs (which have to be designed at a low level) as shown in Fig. 6(b), while carefully choosing the routing nets to guarantee the preservation of the carry–chains. However, at a high level, the only way to guarantee the utilization of the specialized carry logic and the preservation of the carry–chains is by using standard CPAs as basic building blocks. In addition, this allows portability between different FPGA families.

A careful examination of Fig. 6(b) shows that the linear array of three CSAs (see Fig. 6(a)) can be interpreted as an array of 3 bit–width CPAs diagonally deployed (see Fig. 6(c)[1]), as highlighted in Fig. 6(b) for one of them. Hence, the proposed linear compressor tree of N_{op} inputs could be implemented by using an array of $N_{op} - 2$ bit–width CPAs. Although both circuits are exactly the same, we have simply provided a new interpretation at the semantic level. However, this new interpretation allows the implementation of the proposed circuit by means of standard CPAs instantiation.

In this way, the most significant sum–bit of each CPA comprises the final sum–word of the compressor tree, whereas the last carry–out of each CPA comprises the final carry–word. With the exception of the CPAs at ending positions, each CPA processes one bit of each input operand and partial sum, varying the bit weight depending on its relative position. Let us number each CPA according to the position of its final sum bit; for example, CPA 0 is the adder which produces the least significant bit of the final sum–word, and CPA $(N-1)$ is the most significant one. We also number the operands added in the linear array following their order of use, in-

1. Note that, for clarity, the input signal connections to CPAs have not been shown in (c). These connections exactly coincide with the ones in (b)

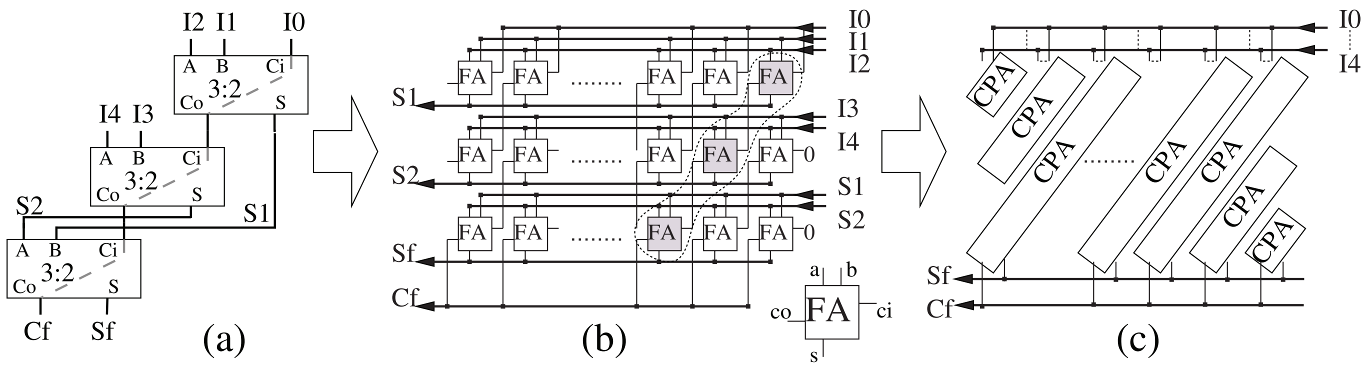


Fig. 6. Transformation of a N bit–width 5:2 linear array compressor tree into a CPA array

cluding the partial sum–words. Thus, the i–th bit of each input of the CPA j adds the bits at position $(j+i-N_{op}+3)$ of the operands $2i+1$ and $(2i+2)$ (note that the operand 0 is connected to the carry input of each CPA). For extreme adders, these previous equations could produce negative bit positions, which should be set to zero as shown next. The use of the ”generate” statement in HDL code (VHDL or Verilog) to implement these connections makes it easy to design a parameterizable compressor tree.

The CPAs from 0 to $(N_{op}-4)$ should be shorter than the regular ones, specifically from one to $(N_{op}-3)$ bit–width, respectively, since the least significant part of these adders is out of range (see Fig. 6). Nevertheless, to deal with these CPAs, we keep the size of the $(N_{op}-2)$ bit–width constant by using zero padding, as the software implementation tool eliminates the superfluous hardware. Similarly, in the most significant part, we have to use CPAs whose final sum and carry bits are out of range (from N to $(N+N_{op}-4)$), although they are required to generate intermediate results. Thus, although the implementation of an N bit–width linear array of $(N_{op}-2)$ CSAs requires the instantiation of $(N+N_{op}-3)$ CPAs of $(N_{op}-2)$ bit–width, the occupied area, after eliminating superfluous hardware, is really the same in both approaches. For instance, to implement a CSA compressor tree of 16 bit–width for 10 input operands $(N_{op}=10)$, 23 CPAs of 8 bit–width are instantiated, but after eliminating the superfluous hardware, they occupy an area equivalent to 16 CPAs of 8 bit–width or 8 CSAs of 16 bit–width.

We should also mention that if $N < N_{op}-2$ is fulfilled, then all CPAs comprising the compressor tree are at ending positions (i.e. any of them will have one or more FA out of range). In practice, this fact limits the maximum delay produced due to carry propagation up to $N \cdot d_{carry}$. This could have a marked effect on the delay for $N_{op} >> N$ since it limits the linear dependence of the delay in relation to N_{op}, as shown in Section 4.

By using this scheme based on standard CPAs, we have created a generic VHDL entity which allows us to implement a compressor tree unit with any number of inputs and any bit–width. This entity could be implemented by using any typical synthesis tool and targeting any FPGA. In addition, it will take advantage of all the resources available to accelerate carry propagate addition, including those which will be developed in the future for new FPGA devices.

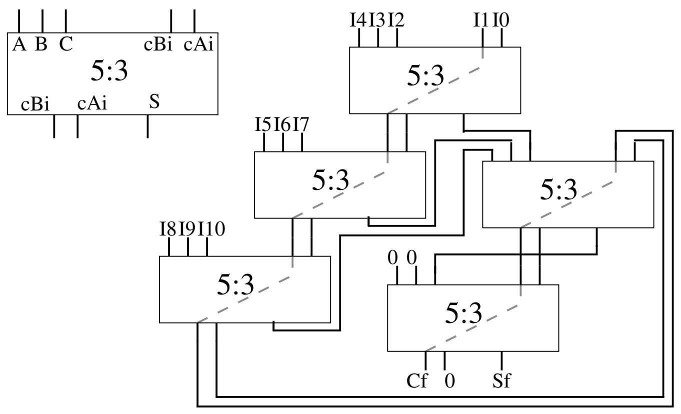


Fig. 7. Carry-save 11:2 compressor tree based on a linear array of 5:3 compressors

3.4 Improvement for ternary adders

To improve the performance of multi–operand addition, the newest FPGA families, such as Virtex-5/7 or Stratix-II/V, can efficiently implement ternary addition $(A+B+C=D)$ [44] [45] [46]. On these FPGAs, a ternary adder requires the same amount of resources as a simple 2-input adder while showing a similar speed. Since each ternary adder eliminates two operands, the number of adders required for a compressor tree is $\lceil (N_{op}-1)/2 \rceil$, which is almost half the amount needed in the binary case. On the other hand, the number of levels is

$$L_{3:1} = \lceil \log_3(N_{op}) \rceil \tag{7}$$

which is considerably faster than the one based on binary adders. Therefore, the ternary adder is preferred to implement multi–operand parallel addition when targeting these devices. We now present how our linear array compressor tree design is adapted to take advantage of this new resource.

The ternary adder structure is based on the integration of an initial 3:2 CSA together with a binary CPA in the same LEs [44] [45] [46]. Thus, a 1–bit element of the ternary adder has three operand input signals, one sum output signal and two carry signals having two inputs and two outputs. One carry signal (*cA*), which is related to the CSA, has limited delay, i.e., the generation of this carry signal does not depend on the previous value of that carry. The other (*cB*), which is related to the CPA, forms a standard carry–chain. Similar to how a carry–ripple adder is converted into a CSA, the ternary adder becomes a 5:3 compressor array (CA) if the carry signals are disconnected.

Hence, as shown in Fig. 7, an efficient compressor tree could be created by constructing a linear array of 5:3 CAs. Similar to the previous approach (see Section 3.2), and with the exception of the first CA where two additional operands are introduced, the two carry input signals are connected to the corresponding carry outputs of the previous adder (preserving the carry–chain constituted by *cB* signal), whereas the partial sum–words generated are added in the same order as they were produced after adding all the input operands.

The number of operands N_{op} should be an odd number and one input zero–word must be used when necessary. In addition, the last partial sum requires a final addition with two zero–words to guarantee a zero output in the last *cA* signals (see Fig. 7), since only the last *cB* signals make up the final carry–word. Taking all these considerations into account, along with the fact that the first 5:3 CA adds five operands, the number of 5:3 CAs required to implement a compressor tree is

$$N_{Add} = \left\lceil \frac{N_{op}-1}{2} \right\rceil \tag{8}$$

since two operands are eliminated by each CA. Given that a classic carry-propagate compressor tree requires the same number of ternary adders, and each N bit–width 5:3 CA uses the same FPGA resources as a ternary adder, the proposed carry–save compressor tree has the same area cost as its equivalent carry-propagate compressor tree.

Regarding delays, a study parallel to the one performed for the 3:2 CSA array in Section 3.2 leads to a similar conclusion, i.e., the proposed array has, in general, a linear structure followed by a hypothetical tree structure. The approximate bounds of the delay for the extreme cases (i.e., a whole linear array or a tree) are

$$D_{low} \approx d_{5:3} + (\left\lceil \frac{N_{op}}{2} \right\rceil - 1) \cdot d_{cB}) \tag{9}$$

$$D_{up} \approx (\lceil \log_3(N_{op}-2) \rceil + 1) \cdot d_{5:3} + \frac{N_{op}-1}{2} d_{cB} \tag{10}$$

where d_{cB} is the delay corresponding to the path between the *cB* inputs of two consecutive 5:3 CAs and $d_{5:3}$ is the delay from one ETL to the next (including the delay of signals *cA*), i.e. from a general input of a CA to the general input of the CA which is connected to the sum–word of the previous CA. Once again, the delay corresponds to the addition of a logarithmic component and a linear one. The logarithmic behavior is dominant for low values of N_{op} whereas the linear behavior dominates for high values of N_{op}. The curve of the delay in relation to N_{op} is also smooth, instead of an increment by steps. These conclusions are borne out in Section 4.

This new compressor tree design (see example in Fig. 7) could also be implemented at a high-level description using ternary CPAs as a basic building block (see Fig. 6). n ternary adders of N_{Add} bit–width diagonally arranged are required to implement it (simplifying the extreme cases). Once again, the most significant sum–bit of each ternary adder comprises the sum–word of the compressor tree, whereas the last *cB* out is the final carry–word. Except for the ending adders, each ternary adder sums one bit of each operand and partial sum, varying the bit weight depending on its relative position. If adders and bits are numbered as for the binary case (see Section 3.3), the i–th bit of the ternary adder j adds the bit $j+i-N_{Add}+1$ of the operands $3i+2$, $3i+3$, and $3i+4$. Note that the operands 0 and 1 are connected to the two carry inputs of each ternary adder and negative bit positions are set to zero.

This linear structure is efficiently mapped on any FPGA device suitable to implement ternary adders. These devices are the newer FPGAs based on 6–LUT or 8–LUT, such as Stratix-II/V for Altera or Virtex-5/7 for Xilinx. The high-level description of this compressor tree could be used by any software tool capable of targeting ternary adders as possible basic elements. In other cases a generic ternary adder should be previously defined.

3.5 Pipelining

One of the main advantages of carry–save compressor trees is that they are very suitable for pipelining. An M–stage pipeline compressor tree of L levels of compressors is implemented simply by introducing one level of registers for each L/M levels of compressors. However, the proposed designs are defined using an array of CPAs, which complicates the introduction of registers in the linear array compressor tree.

However, the proposed approach is still easy to apply in pipeline designs. The pipeline compressor tree is designed by utilizing smaller linear array compressor trees as basic building blocks, which are registered in the input or output. For instance, a two-stage 18:2 compressor tree is built by using three 6:2 linear array compressor trees.

The best size for the linear array blocks depends on the number of operands (N_{op}) and the number of stages (S) required for our design, but the best size could be found using

$$X = \left\lceil 2 \sqrt[S]{\frac{N_{op}}{2}} \right\rceil \tag{11}$$

In the case that the time requirements are known instead of the number of stages, the graphs shown in Section 4 could be used to find the size that best meets these requirements.

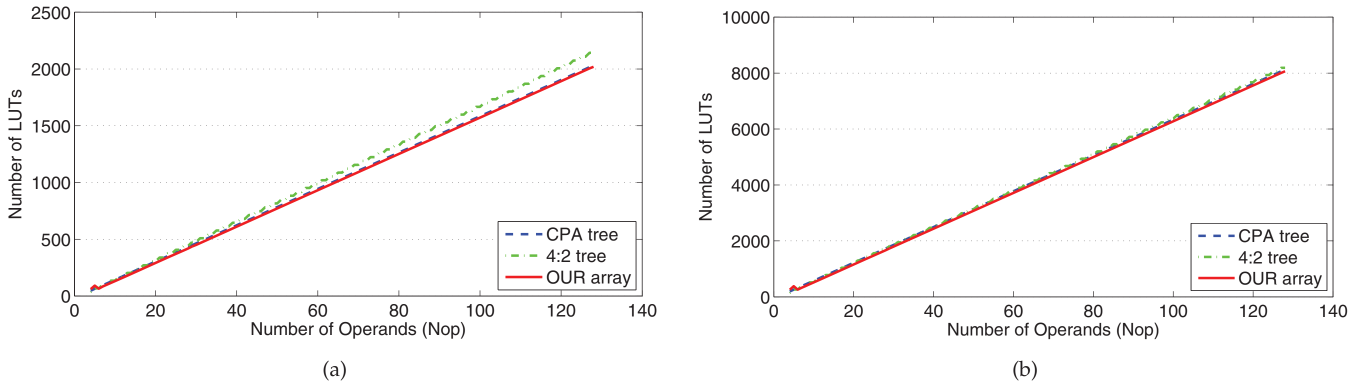


Fig. 8. Area (LUTs) of the different compressor tree approaches implemented on Virtex-4 when varying the number of input operands (N_{op} for (a) 16 bit–width and (b) 64 bit–width

4 Implementation results and comparison

To measure the effectiveness of the designs presented in this paper, we have developed two generic VHDL modules implementing the proposed compressor tree structures: first, the linear array implemented by using CPAs (binary and ternary) and, second, the 4:2 compressor tree using the design of the compressor presented in [28]. Both modules provide the output result in carry–save format and allow the selection of different parameters such as: the number of operands (N_{op}), the number of bits per operand (N), and the basic building blocks (i.e. binary or ternary adder) for the linear array. For the purposes of comparison, similar modules, which implement classic adder tree structures based on binary CPAs and ternary CPAs, have also been developed. All these modules were simulated using Modelsim SE 6.3f and they were synthesized using Xilinx ISE 9.2, targeting Spartan-3A, Virtex-4, and Virtex-5 devices. A generic ternary adder module was designed following the recommendations of Xilinx [46], since this adder is not automatically supported by ISE 9.2. Furthermore, to investigate their portability, compressor trees based on ternary CPAs were also synthesized to target the Altera Stratix-II family. In this case, the ternary adders are directly instantiated at a high level. We now summarize the main results obtained in this study.

4.1 Results on FPGA families with support for binary CPAs

For the sake of simplicity, of the two FPGA families tested (Spartan-3A and Virtex-4), only the results corresponding to the Virtex-4 family are presented, since the results are very similar for both families. On these FPGAs, the compressor trees based on ternary adders are not efficiently implemented, and thus we have only tested the ones based on binary adders. For purposes of clarity, let us denote as *CPA tree* the classic tree structure based on CPAs , *OUR array* the proposed linear array structure based on 3:2 CSAs, and *4:2 tree* the classic tree structure based on a 4:2 compressors.

Regarding the area, Fig. 8 shows the number of LUTs required by the different compressor tree structures when varying N_{op} from 4 to 128 operands, for 16 and 64 bit–widths. With the exception of 4- and 5-operand compressor trees, which we consider separately, the area used for the three compressor trees is very similar and varies linearly with the number of operands and the bit–width, as expected. Specifically, the area of *CPA tree* and *OUR array* is practically identical, whereas the *4:2 tree* requires a little more area (up to 6% for a 16 bit–width and up to 2% for a 64 bit–width), due to the implementation of boundary bits on the 4:2 CA. Let us now consider the cases of 4 and 5 operands. The CPAs involved in the implementation of *OUR array* are only 2 and 3 bit–width for all operand sizes. Given this small size, the synthesis tool implements these CPAs by exclusively using LUTs, and not the specialized carry–chain, since this produces faster circuits. As a consequence, there is an increase in area for these particular cases (as shown in Fig. 8), which could be eliminated by manually designing low-level CPAs or by changing the synthesis tool. On the other hand, this faster CPA implementation leads to more significant speed–ups for these cases, as shown in the following.

Regarding speed, Fig. 9 shows the speed–up achieved when using *OUR array* instead of *CPA tree*, for different numbers of operands and varying the number of bits from 16 to 96 bit–width. As can be seen, *OUR array* is always faster than *CPA tree*, and the speed–up practically grow linearly in relation to number of bits. This is due to the linear dependency of the delay on the CPAs, whereas the delay remains constant for CSAs. Thus, although the speed–up achieved for 16 bit–width is moderate, i.e., 12% to 50% faster (in the range of values selected for N_{op} and excluding 4- and 5-operand compressor trees), for 64 bit–width the speed–up ranges from 44% to 104% faster. As mentioned above, 4- and 5-operand compressor trees achieve high speed–up due to the small CPAs required.

The speed–up achieved is very dependent on the

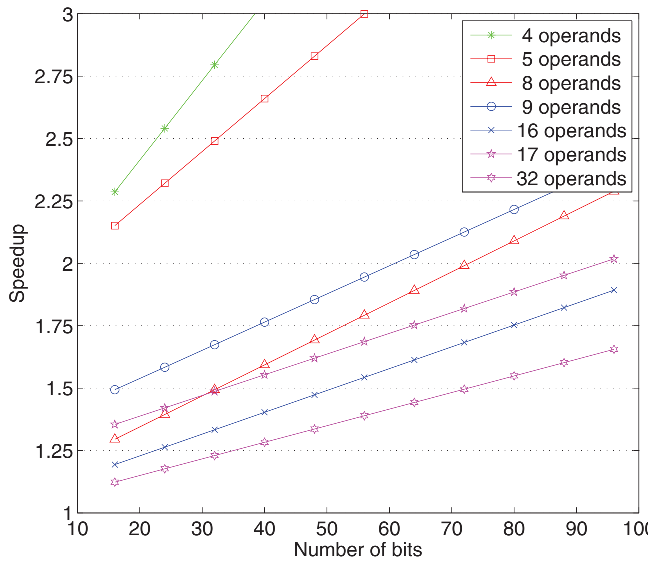


Fig. 9. Speed-up achieved on Virtex-4 by using *OUR array* instead of *CPA tree* for different number of operand (N_{op}) when varying the number of bits (N)

number of operands N_{op}. On the one hand, the growth of the speed–up in relation to N (i.e. the slope) increases when N_{op} decreases. On the other hand, the starting point of the speed–up line generally decreases when N_{op} increases, although it presents a strong increase for certain values of N_{op}. For this reason, and due to compressing the maximum amount of information while keeping the image simple, we have used values of N_{op} which can be used as bounds of monotonic intervals. In this way, for values of N_{op} between 5 and 8 operands, the line of the speed–up is between these bound lines; for values of N_{op} between 9 and 16 operands, the line is between these bound lines, and so on. In addition, inside these ranges, and for a fixed number of bits, the speed–up always decreases when the number of operands increases.

Fig. 10 represents the delay changes in relation N_{op} from 4 to 128 operands for N equal to 16 and 64 bits. For all cases, *OUR array* is the fastest solution, whereas *CPA tree* is faster than *4:2 tree* when N is 16 (except for N_{op} lower than 9); we obtain the opposite result when N is 64 bits. The graphs corresponding to *CPA tree* and *4:2 tree* are arranged in steps. They present an almost imperceptible growth in delay when N_{op} increases, but a very high growth occurs when N_{op} is a power of two. This behavior is due to the introduction of a new level of adders at these points. Nevertheless, the behavior of *OUR array* is smoother since the delay of the proposed structure depends on both the number of levels and the number of adders on each level (see Section 3.2). This figure makes clear the behavior shown in Fig. 9. Similar behavior in relation to N_{op} is observed when comparing *OUR array* with *4:2 tree* . However, in this case, the speed–up achieved is practically independent of the number of bits N.

TABLE 1
Bounds and averages of the speed–up achieved on Virtex-4 by using *OUR array* and *4:2 tree* instead of *CPA tree*

N_{op}	4-16	17-32	33-64	65-128
N	*OUR array* vs *CPA tree* (min/mean/max)			
16	1.19/1.46/2.29	1.12/1.21/1.35	1.06/1.16/1.27	1.04/1.11/1.20
32	1.33/1.66/2.80	1.23/1.33/1.49	1.14/1.25/1.38	1.05/1.15/1.26
64	1.61/2.08/3.81	1.44/1.56/1.75	1.31/1.43/1.59	1.14/1.27/1.43
128	2.17/2.92/5.85	1.87/2.03/2.28	1.65/1.81/2.01	1.36/1.55/1.76
	4:2 tree vs *CPA tree* (min/mean/max)			
16	-0.97/1.06/1.52	0.90/0.91/0.91	0.86/0.87/0.87	0.83/0.84/0.84
32	1.08/1.21/1.86	0.99/0.99/1	0.93/0.93/0.94	0.89/0.90/0.90
64	1.31/1.51/2.54	1.16/1.17/1.17	1.07/1.07/1.08	1.01/1.01/1.01
128	1.77/2.11/3.90	1.50/1.52/1.52	1.34/1.35/1.36	1.24/1.24/1.25

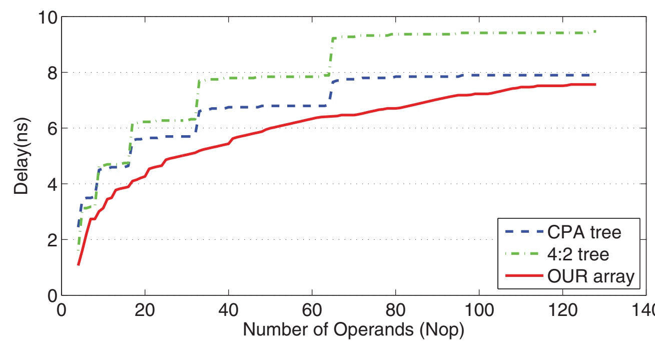
We should also note that when the bit–width N is very low compared to N_{op}, the maximum delay of *OUR array* is limited due to the effect of the ending CPAs, as stated in Section 3.3. For this reason, in Fig. 10, the delay of *OUR array* for more than 64 operands is smaller for 16 bit–width (see Fig. 10(a)) than for 64 bit–width (see Fig. 10(b)).

Table 1 presents the bounds and the average of the speed–ups obtained by using *OUR array* or *4:2 tree* instead of *CPA tree*, for different ranges of N_{op}. The *OUR array* approach is clearly superior to the classic *4:2 tree*, and it could achieve strongly reduced delays, even for short bit–widths.

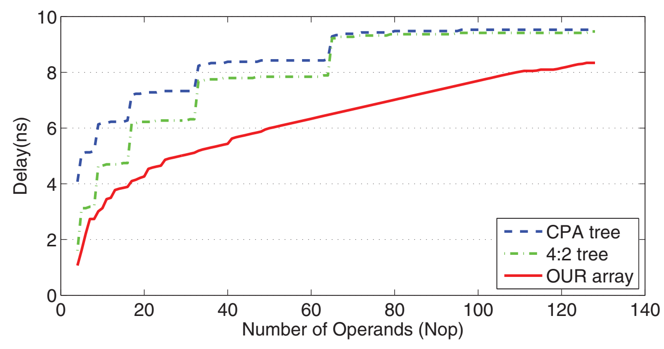
4.2 Results on FPGA families with support for ternary CPAs

We have synthesized all the compressor tree structures, varying N_{op} from 4 to 128 operands for N equal to 16 and 64 bits, targeting the Virtex-5 family. Along with the compressor trees used in the previous comparison, we have also studied the compressor trees constructed using ternary adders. These are classic tree structures based on ternary CPAs (*CPA3 tree*) [44] and the linear array structure based on the 5:3 CA proposed in Section 3.4 (*OUR3 array*). As expected, Fig. 11 and Fig. 12 show that the compressor trees based on ternary adders clearly outperform the others.

Regarding the area, Fig. 11 shows the number of LUTs required to implement the different compressor trees on Virtex-5 for 16 bit–width when varying N_{op}. As stated in Section 3.4, *CPA3 tree* and *OUR3 array* use the same amount of hardware and practically halve the number of LUTs required to implement the other ones (see Fig. 11). As expected, the results obtained for *CPA tree*, *4:2 tree*, and *OUR array* are approximately the same on Virtex-4 and Virtex-5. Note that, although not shown in the figure, the variation in the area corresponding to all compressor trees is also linear in relation to the bit–width.

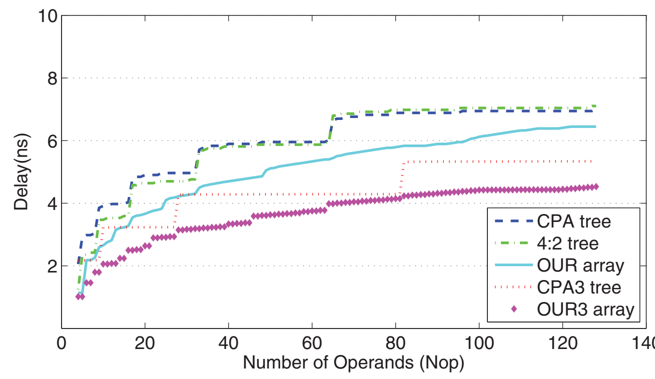


(a)

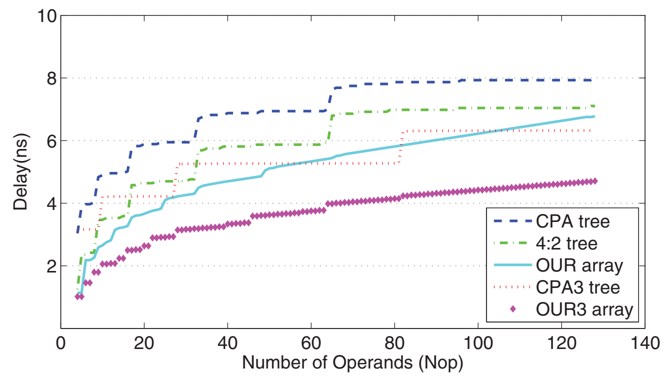


(b)

Fig. 10. Delay (ns) of the different compressor tree approaches implemented on Virtex-4 when varying the number of input operands (N_{op}) for (a) 16 bit–width and (b) 64 bit–width



(a)



(b)

Fig. 12. Delay (ns) of the different compressor tree approaches implemented on Virtex-5 when varying the number of input operands (N_{op}) for (a) 16 bit–width and (b) 64 bit–width

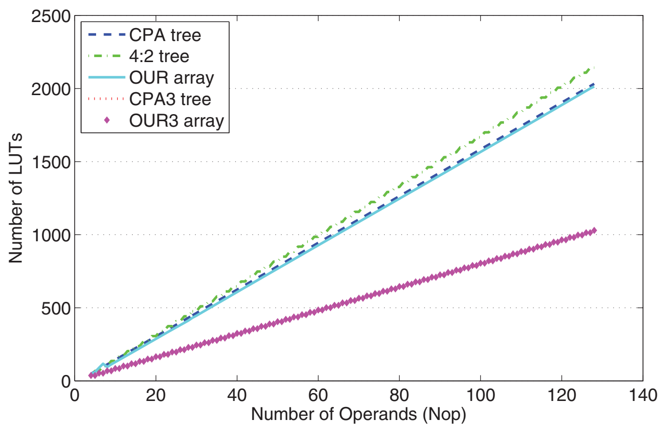


Fig. 11. Area (LUTs) of the different compressor tree approaches implemented on Virtex-5 when varying the number of input operands (N_{op}) for 16 bit–width

Regarding speed, Fig. 12 shows the delay in relation to N_{op} for 16 and 64 bit–width when these compressor trees are implemented on Virtex-5. Again, ternary adder-based compressor trees are generally superior to the others, although *OUR array* is sometimes faster than *CPA3 tree*, especially for low N_{op} and high N. As expected, the relative behavior of the compressor trees previously implemented on Virtex-4 is practically the same on Virtex-5 (see Fig. 10 and Fig. 12).*OUR3 array* is always the fastest approach and it achieves a considerable speed–up in relation to the second one, *CPA3 tree*. The behavior of these two compressor trees in relation to N_{op} is similar to the one described in the previous section for *OUR array* and *CPA tree*, respectively, but the new curves are adjusted to log3 instead of to log2. Thus, a similar variation in speed–up is achieved by *OUR3 array* compared to *CPA3 tree*. That is, there is a slight monotonically decrease in speed–up when N_{op} increases, but it presents high increments for values of N_{op} corresponding to a power of three, as shown in Fig. 13. This figure shows speed–up compared to bit–width for values of N_{op} which are the bounds of monotonically decreasing intervals (values around the power of three). It looks very similar to Fig. 9. Once again, the speed–up linearly increases with the number of bits, whereas it generally decreases by steps in relation to the number of operands.

To give a more accurate idea of the improvements achieved using *OUR3 array*, Table 2 shows the bounds and the averages of the speed–up for different ranges of

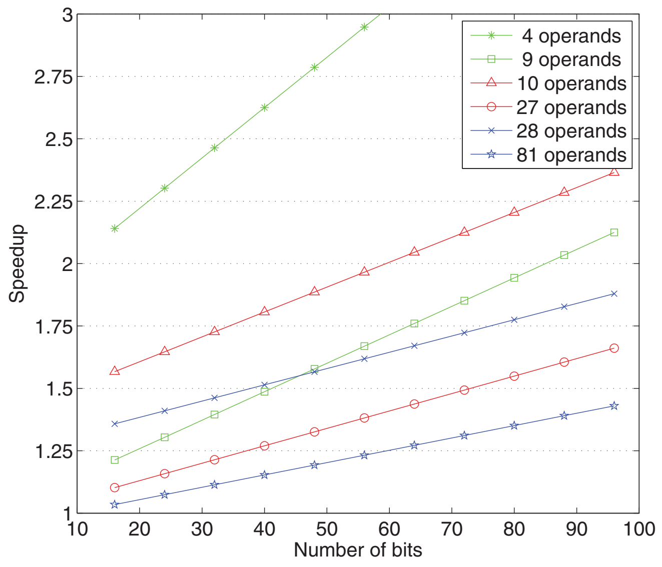


Fig. 13. Speed-up achieved on Virtex-5 by using *OUR3 array* instead of *CPA3 tree* for different number of operand (N_{op}) when varying the number of bits (N)

TABLE 2
Bounds and averages of the speed–up achieved on Virtex-5 by using *OUR3 array* instead of *CPA3 tree*

N_{op}	4-9	10-27	28-81	82-128
N	*OUR3 array* vs *CPA3 tree* (min/main/max)			
16	1.21/1.62/2.14	1.10/1.30/1.57	1.03/1.18/1.36	1.18/1.21/1.26
32	1.40/1.86/2.46	1.21/1.43/1.73	1.11/1.27/1.46	1.20/1.27/1.33
64	1.76/2.35/3.11	1.44/1.70/2.05	1.27/1.45/1.67	1.34/1.42/1.49
128	2.49/3.32/4.40	1.88/2.22/2.68	1.59/1.81/2.09	1.62/1.71/1.80

N_{op}.

4.3 Portability to a different brand

We used the Altera Stratix-II family to validate the proposed approaches on a different brand devices. We tested the implementation of compressor trees based on ternary adders which are the best option on these devices. The same VHDL cores described in the previous section have been synthesized using Synplify Pro D-2010.03 targeting the Stratix-II family. In this case, the basic ternary adders have been instantiated by a high-level VHDL, since it is directly supported by this tool. This has a serious drawback, since the inferred ternary adder does not support carry input and output signals. Thus, the adders have to be extended by one bit at each extreme to control the carry signals. Consequently, this introduces additional area and delay which affects the results.

Fig. 14 shows the area results obtained for *OUR3 array* and *CPA3 tree* for 16 and 64 bit–widths. We can see that *OUR3 array* presents a considerable area overhead, a mean of 10%, due to the use of oversized adders. This excess could be eliminated by implemented a ternary adder core at low level.

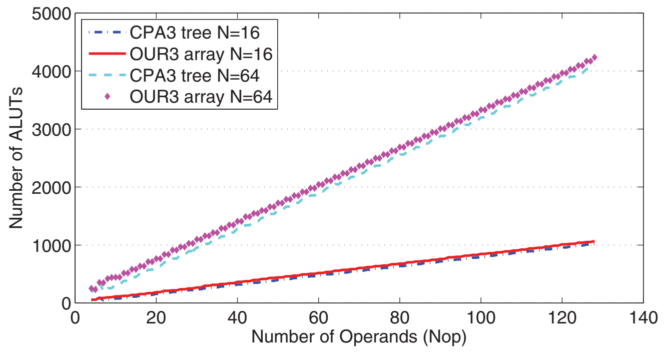


Fig. 14. Area (ALUTs) of the ternary adder-based compressor tree approaches implemented on Stratix-II when varying the number of input operands (N_{op}) for 16 and 64 bit–width

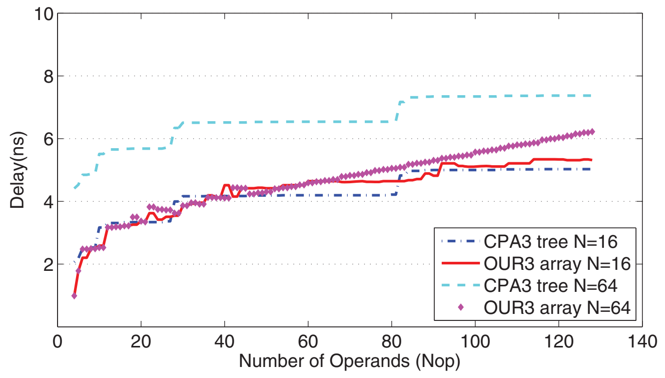


Fig. 15. Delay (ns) of the ternary adder-based compressor tree approaches implemented on Stratix-II when varying the number of input operands (N_{op}) for 16 and 64 bit–widths

Regarding speed, Fig. 15 shows the delay for both types of compressor trees using 16 and 64 bit–widths when varying N_{op}. As expected, the behavior is similar to that on Virtex-5 (see Fig. 12). However, in this case, *CPA3 tree* is faster than *OUR3 array* in most of the compressor trees for 16 bit-width, which is also explained for the wider ternary adders used. For 64 bit-width, again *OUR3 array* strongly reduces the delay. This is clearly presented in Fig. 16, where the speed–up achieved by *OUR3 array* related to N is shown for several values of N_{op} (the bounds of monotonically decreasing intervals). In this case, although *OUR3 array* achieves less improvement than on Virtex-5 for lower N (see Fig. 13), the increase in speed-up is much greater when N increases (greater slope), producing better results for $N > 64$, as shown in Table 3.

4.4 Compressor trees with irregularly shaped data

In the previous experiments we have only considered compressor trees with a regular shape, i.e. they have the same number of bits in all columns and rows. However,

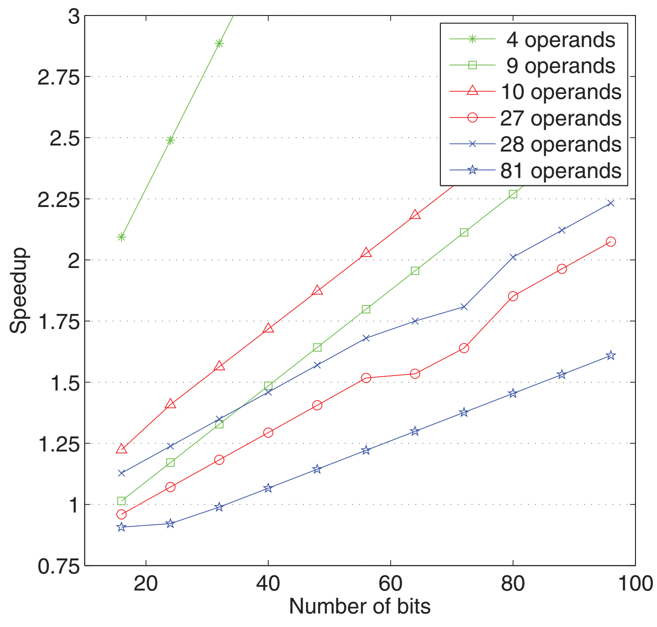


Fig. 16. Speed-up achieved on Stratix-II by using *OUR3 array* instead of *CPA3 tree* for different number of operand (N_{op}) when varying the number of bits (N)

TABLE 3
Bounds and averages of the speed–up achieved on Stratix-II by using *OUR3 array* instead of *CPA3 tree*

N_{op}	4-9	10-27	28-81	82-128
N	*OUR3 array* vs *CPA3 tree* (min/main/max)			
16	1.01/1.27/2.09	0.92/1.02/1.22	0.9 /0.95/1.13	0.94/0.98/1.06
32	1.33/1.64/2.89	1.08/1.26/1.56	0.99/1.12/1.35	1.04/1.07/1.11
64	1.96/2.47/4.47	1.49/1.71/2.18	1.3 /1.48/1.75	1.18/ 1.3 /1.41
128	3.21/4.13/7.64	2.52/2.74/3.42	1.92/2.17/2.67	1.69/1.86/2.03

our approaches should be extended to implement non-regular compressor trees, since there are many applications with this characteristic. The detailed optimization of these kinds of trees would require further research and is beyond the scope of this paper. Thus, in the following we present a non-optimal solution to these cases; we will attempt to improve this in a future study.

We consider that the direct use of our linear array approach by zero padding of the input operands should perform well in relation to delay, but it could well produce a noticeable area overhead due to the incomplete use of the instantiated CPA. Nevertheless, if the input operands are ordered from small to large bit–widths in the CSA array, the area overhead will be very limited since modern software tools are able to remove unnecessary hardware. To verify this, we have implemented the partial product reduction tree (PPRT) of a generic multiplier for $N \times N$ bits, using our approach and a standard CPA tree. The results obtained are summarized in Fig. 17 and Fig. 18 for the area and delay, respectively. We can see that the speed–up using *OUR array* grows along with the size of the multiplier, whereas the area

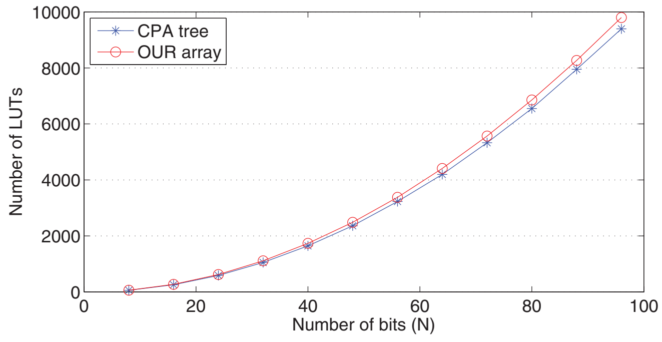


Fig. 17. Area (LUTs) of PPRT implemented on Virtex-4 by using different compressor tree approaches when varying the number of bits (N)

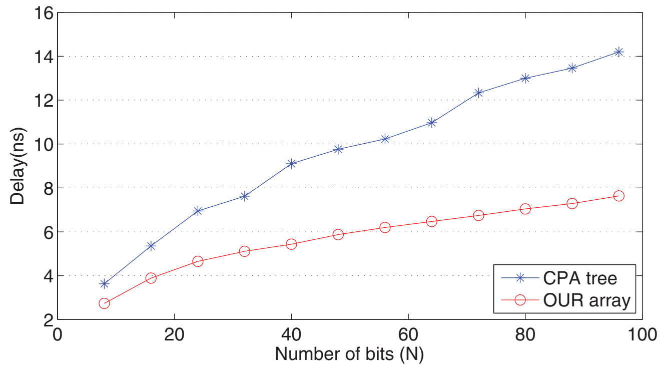


Fig. 18. Delay (ns) of PPRT implemented on Virtex-4 by using different compressor tree approaches when varying the number of bits (N)

overhead is about 5%.

4.5 Comparison with trees based on GPCs

As summarized in Section 2, the implementation of carry–save compressor trees on FPGA has been previously proposed based on the use of GPCs [30] [31] [32] [33]. A fair comparison between our results and theirs is not possible, since they do not describe the data in sufficient detail. Their results correspond to a very limited number of sizes (i.e. the combinations of N_{op} and N); specifically, only 3 sizes are reported by Matsunaga et al. [33] and only 11 cases by Parandeh–Afshar et al. [30] [31] [32]. In addition, the latter study does not provide direct information about the sizes used. Furthermore, it does not seem possible to predict any behavior based on their results, since they have a high variability. Thus, it makes it impossible to extrapolate them to other sizes. Despite these problems, if we still wish to perform a quantitative comparison, we could assume that they use a sufficient number of experiments, and make a quantitative comparison of the speed based on the relative improvement achieved for each approach in relation to the ternary CPA tree approach. Regarding area, we do not perform a quantitative comparison since the data reported is insufficient and even contradictory.

According to [32], an average speed–up of 27% and 32% is achieved by [30] and [32], respectively, for the tested cases, which seems to be for $N_{op} < 17$, whereas [31] reports an average speed improvement of 23%. Our linear approach (*OUR3 array*) achieves an average of 55% on Virtex-5 and 17% on Stratix-II for $N_{op} < 17$ and 16 bit–width (these values are much higher for longer bit–width). On the other hand, the results provided in [33] for $N_{op} = 16$ size are 32%, 29%, and 36% speed–ups for 16-, 24- and 32 bit–widths, respectively, whereas *OUR3 array* achieves 29%, 36% and 42% speed–ups on Virtex-5, but only 3%,16%, and 26% on Stratix-II. Note that the results for the GPC-based tree cannot be extrapolated to different sizes.

On the other hand, although the sizes of the GPCs could be selected to fit on N-LUT-based FPGAs, they are not effective on 4–LUT-based FPGAs and they do not effectively use the specialized carry resources. In addition, GPC-based approaches require the use of a software algorithm to design each specific compressor tree, whereas our approaches can be modeled by a small parameterizable HDL code.

In summary, if the use of an heuristic is not a disadvantage and 6–LUT–based FPGA is the target technology, both solutions should be considered to obtain the best performance in relation to the speed/area ratio. The results will depend on the specific size, the shape of the compressor tree and the targeted FPGA family.

5 Conclusions

Efficiently implementing carry–save compressor trees on FPGA, in terms of area and speed, is made possible by using the specialized carry–chains of these devices in a novel way. Similar to what happens when using ASIC technology, the proposed carry–save linear array compressor trees lead to marked improvements in speed compared to CPA approaches and, in general, with no additional hardware cost. Furthermore, the proposed high-level definition of CSA arrays based on CPAs facilitates ease-of-use and portability, even in relation to future FPGA architectures, since CPAs will probably remain a key element in the next generations of FPGA. We have compared our architectures, implemented on different FPGA families, to several designs and have provided a qualitative and quantitative study of the benefits of our proposals.

Acknowledgments

This work was supported in part by the Junta of Andalucía under the project P07-TIC-02630, and the Ministry of Education and Science of Spain under contract TIN2006-01078.

References

[1] B. Cope, P. Cheung, W. Luk, and L. Howes, "Performance comparison of graphics processors to reconfigurable logic: A case study," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 433–448, Apr. 2010.

[2] S. Dikmese, A. Kavak, K. Kucuk, S. Sahin, A. Tangel, and H. Dincer, "Digital signal processor against field programmable gate array implementations of space-code correlator beamformer for smart antennas," *IET Microwaves, Antennas Propagation*, vol. 4, no. 5, pp. 593–599, May 2010.

[3] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 886–896, Jul. 2005.

[4] F. Schneider, A. Agarwal, Y. M. Yoo, T. Fukuoka, and Y. Kim, "A fully programmable computing architecture for medical ultrasound machines," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 2, pp. 538–540, Mar. 2010.

[5] J. Hill, "The soft-core discrete-time signal processor peripheral [applications corner]," *IEEE Signal Processing Magazine*, vol. 26, no. 2, pp. 112–115, Mar. 2009.

[6] J. S. Kim, L. Deng, P. Mangalagiri, K. Irick, K. Sobti, M. Kandemir, V. Narayanan, C. Chakrabarti, N. Pitsianis, and X. Sun, "An automated framework for accelerating numerical algorithms on reconfigurable platforms using algorithmic/architectural optimization," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1654–1667, Dec. 2009.

[7] H. Lange and A. Koch, "Architectures and execution models for hardware/software compilation and their system-level realization," *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1363–1377, Oct. 2010.

[8] L. Zhuo and V. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Transactions on Computers*, vol. 57, no. 8, pp. 1057–1071, Aug. 2008.

[9] C. Mancillas-Lopez, D. Chakraborty, and F. Rodriguez Henriquez, "Reconfigurable hardware implementations of tweakable enciphering schemes," *IEEE Transactions on Computers,*, vol. 59, no. 11, pp. 1547–1561, Nov. 2010.

[10] T. Guneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1498–1513, Nov. 2008.

[11] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[12] M. Frederick and A. Somani, "Beyond the arithmetic constraint: Depth-optimal mapping of logic chains in LUT-based FPGAs," *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, pp. 37–46, 2008.

[13] T. PreuBer and R. Spallek, "Enhancing FPGA device capabilities by the automatic logic mapping to additive carry chains," *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pp. 318–325, 2010.

[14] S. Gao, D. Al-Khalili, and N. Chabini, "Implementation of large size multipliers using ternary adders and higher order compressors," *International Conference on Microelectronics (ICM)*, pp. 118–121, 2009.

[15] S. Gao, D. Al-Khalili, and N. Chabini, "Fpga realization of high performance large size computational functions: multipliers and applications," *Analog Integrated Circuits and Signal Processing*, pp. 1–15, 2011; Article in Press.

[16] K. Macpherson and R. Stewart, "Rapid prototyping - area efficient FIR filters for high speed FPGA implementation," *Vision, Image and Signal Processing, IEE Proceedings -*, vol. 153, no. 6, pp. 711 –720, dec. 2006.

[17] P. Meher, S. Chandrasekaran, and A. Amira, "FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic," *Signal Processing, IEEE Transactions on*, vol. 56, no. 7, pp. 3009 –3017, july 2008.

[18] Z. Kincses, Z. Nagy, L. Orzo, P. Szolgay, and G. Mezo, "Implementation of a parallel SAD based wavefront sensor architecture on FPGA," in *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*, aug. 2009, pp. 823 –826.

[19] F. Bensaali, A. Amira, and A. Bouridane, "Accelerating matrix product on reconfigurable hardware for image processing applications," *Circuits, Devices and Systems, IEE Proceedings -*, vol. 152, no. 3, pp. 236 – 246, june 2005.

[20] Y. F. Chan, M. Moallem, and W. Wang, "Design and implementation of modular FPGA-based PID controllers," *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 4, pp. 1898 –1906, aug. 2007.

[21] C. Villalpando, A. Morfopolous, L. Matthies, and S. Goldberg, "FPGA implementation of stereo disparity with high throughput

for mobility applications," in *Aerospace Conference, 2011 IEEE*, march 2011, pp. 1 –10.
[22] C. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 2, pp. 14–17, 1964.
[23] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, no. 5, pp. 349–356, 1965.
[24] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.
[25] P. Kornerup, "Reviewing 4-to-2 adders for multi-operand addition," *The Journal of VLSI Signal Processing*, vol. 40, pp. 143–152, 2005.
[26] J.-L. Beuchat and J.-M. Muller, "Automatic generation of modular multipliers for FPGA applications," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1600–1613, Dec 2008.
[27] G. 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, 2008.
[28] M. Ortiz, F. Quiles, J. Hormigo, F. Jaime, J. Villalba, and E. Zapata, "Efficient implementation of carry-save adders in FPGAs," *20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 207–210, 2009.
[29] W. Kamp, A. Bainbridge-Smith, and M. Hayes, "Efficient implementation of fast redundant number adders for long wordlengths in FPGAs," *International Conference on Field-Programmable Technology (FPT'09)*, pp. 239–246, 2009.
[30] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient synthesis of compressor trees on FPGAs," *Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 138–143, 2008.
[31] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting fast carry-chains of FPGAs for designing compressor trees," *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 242–249, aug. 2009.
[32] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Improving synthesis of compressor trees on FPGAs via integer linear programming," *International Conference on Design, Automation and Test in Europe (DATE'08)*, pp. 1256–1261, 2008.
[33] T. Matsunaga, S. Kimura, and Y. Matsunaga, "Multi-operand adder synthesis on FPGAs using generalized parallel counters," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 337–342, 2010.
[34] M. Frederick and A. Somani, "Multi-bit carry chains for high-performance reconfigurable fabrics," *Int. Conf. on Field Programmable Logic and Applications*, pp. 1–6, 2006.
[35] P. Brisk, A. Verma, P. Ienne, and H. Parandeh-Afshar, "Enhancing FPGA performance for arithmetic circuits," *44th ACM/IEEE Design Automation Conference (DAC'07)*, pp. 334–337, 2007.
[36] H. Parandeh-Afshar, A. Verma, P. Brisk, and P. Ienne, "Improving FPGA performance for carry-save arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 4, pp. 578–590, apr. 2010.
[37] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. Verma, P. Brisk, F. Gurkaynak, Y. Leblebici, and P. Ienne, "Architectural improvements for field programmable counter arrays: Enabling efficient synthesis of fast compressor trees on FPGAs," *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, pp. 181–190, 2008.
[38] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. K. Verma, H. S. A. Niaki, C. Nicopoulos, F. K. Gurkaynak, P. Brisk, Y. Leblebici, and P. Ienne, "Field programmable compressor trees: Acceleration of multi-input addition on FPGAs," *ACM Trans. on Reconfigurable Technology Systems*, vol. 2, pp. 13:1–13:36, Jun 2009.
[39] R. Gutierrez, J. Valls, and A. Perez-Pascual, "FPGA-implementation of time-multiplexed multiple constant multiplication based on carry-save arithmetic," *19th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 609–612, 2009.
[40] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, "Compact high-speed parallel multiplication scheme," *IEEE Transactions on Computers*, vol. C-26, no. 10, pp. 948–957, 1977.
[41] S. Dormido and M. Canto, "Synthesis of generalized parallel counters," *IEEE Transactions on Computers*, vol. C-30, no. 9, pp. 699–703, Sep. 1981.
[42] P. Kornerup and J.-M. Muller, "Leading guard digits in finite precision redundant representations," *IEEE Transactions on Computers*, vol. 55, no. 5, pp. 541–548, May 2006.
[43] S.-F. Hsiao, M.-R. Jiang, and J.-S. Yeh, "Design of high-speed low-power 3-2 counter and 4-2 compressor for fast multipliers," *IEE Electronics Letters*, vol. 34, no. 4, pp. 341–343, Feb. 1998.
[44] Altera, "Stratix ii vs. Virtex-4 performance comparison, WP-S2052505-2.0," *www.altera.com/literature*, 2006.
[45] Xilinx, "Achieving higher system performance with the virtex-5 family of FPGAs, WP245," *www.xilinx.com/support/documentation*, 2006.
[46] Xilinx, "Ug369 virtex-6 FPGA dsp48e1 slice, user guide," *www.xilinx.com/support/documentation*, 2009.

PLACE PHOTO HERE

Javier Hormigo received an M.Sc and a Ph.D., both in Telecommunication Engineering, from the University of Mlaga, Spain, in 1996 and 2000, respectively. He was a member of the Image and Vision Department of the Instituto de ptica, Madrid, Spain, in 1996. He joined the University of Mlaga in 1997 and is currently Associate Professor in the Computer Architecture Department. His research interests include computer arithmetic, specific application architectures, and FPGA.

PLACE PHOTO HERE

Julio Villalba received a B.Sc degree in Physics in 1986 (University of Granada, Spain) and a Ph.D in Computer Engineering in 1995 (University of Malaga, Spain). During 1986-1991, he worked in the R&D Department of Fujitsu Spain and was an assistant professor. Since 2007, he has been a Full Professor in the Department of Computer Architecture at the University of Malaga. Currently he is an Associate Editor of IEEE Trans. on Computers. His research interests include computer arithmetic and specific application architectures.

PLACE PHOTO HERE

Emilio L. Zapata obtained a degree in Physics from the University of Granada in 1978, and a Ph.D. in Physics from the University of Santiago de Compostela in 1983. From 1978 to 1982 he was assistant professor at the University of Granada. In 1982 he joined the University of Santiago de Compostela where he became full professor in 1990. Since 1991, he has been full professor at the University of Malaga. Currently, he is head of the Computer Architecture Department at the University of Malaga, Spain.

Dr. Zapata has published over 90 journal and 200 conference papers on the parallel computing field (applications, compilers and architectures). His main research topics include: application fields, compilers, computer arithmetic and application-specific array processors. Dr. Zapata is a member of the Editorial Board of the Journal of Parallel Computing and Journal of Systems Architecture. He has also been a guest editor of special issues of the Journal of Parallel Computing.