INTERNATIONAL JOURNAL OF RESEARCH
In Advanced Engineering Technologies

Volume 2, Issue 2 OCT 2014

VLSI IMPLEMENTATION OF MULTI OPERAND DECIMAL ADDERS USING CSA

[1] **Mithikela Varun Kumar**, **PG Scholar in VLSI,**
[2] **K.Venkateswarlu, Assoc. Professor, ECE Department,**
[1]mvarun92@gmail.com ,
[2]ecehodjnit2@gmail.com.
Jawaharlal Nehru Institute of Technology, Ibrahimpatnam, Hyd, Telangana

ABSTRACT: Multi-operand adders, which are also found in parallel multipliers, usually consist of the compression trees which reduce the number of operands per a bit to two, and the carry propagate adder for the two operands in ASIC implementation. The former part is usually realized using full adders or(3;2) counters like Wallace-trees in ASIC, though adder trees or dedicated hardware are used in FPGA. In this paper, an approach to realize compression trees on FPGAs is proposed. In case of FPGA with m-input LUT, any larger or generalized parallel counters with up to m inputs can be realized with one LUT per an output. Our approach utilizes generalized parallel counters with up to m inputs and synthesizes the compression trees to implement high-performance multi-operand adders by setting some intermediate height limits in the compression process like Dadda multipliers. The goal of this research is to design arithmetic circuits that meet the challenges faced by computer architects during the design of high performance embedded systems.The focus is narrowed down to addition algorithms and the design of high speed adder architectures. Addition is one of the most basic operations performed in all computing units, including microprocessors and digital signal processors. It is also a basic unit utilized in various complicated algorithms of multiplication and division.
Keywords: Computer arithmetic, reconfigurable hardware, multi operand addition, redundant representation, carry-save adders

I. INTRODUCTION

Multi-operand addition, which is often found in partial product reduction of multipliers, or some combinations of addition and multiplication, is a fundamental and frequently used arithmetic operation. Though it can be realized with carry-propagate adder (CPA) trees, fast multi-operand addition usually consists of two phases, where the number of addends is compressed to 2 such as a Wallace tree and a Dadda tree, and then the final CPA generates the result of multiplication for ASIC implementation. Such trees are often constructed using 3-input 2-output counters (also called carry-save adder or full adder) and 2-input 2-output counters (half adder) as basic components.

In this paper we prove that there is possibility to implement carry-save adders on FPGA devices with a similar hardware cost to that of carry-propagate adders, while keeping a constant computation time, in such a way that considering operands with number of bits greater or equal to 16, the speed gain is notorious, this process is similar to an ASIC-based design.

II. CARRY SAVE ADDERS ON FPGA

This paper focuses mainly on the inner architecture of FPGAs with specialized carry-logic like Virtex 2, 4and Spartan 2, 3 of Xilinx and 4-input Look up tables. In spite of new generation Field programmable gate arrays which are having new inner architecture, FPGAs with four-input LUTs are widely used for medium complex applications due to low cost and low power consumption

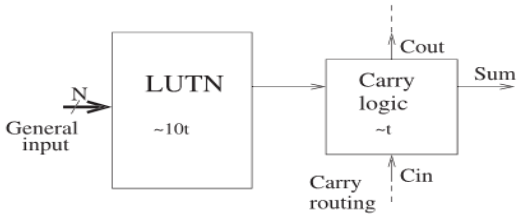

Fig.1. General scheme of dedicated carry-chain resources included in modern FPGA devices.

In carry save addition (CSA) implementation on FPGA, the carry-out bit and the sum bit are generated using two LUTs whereas a carry propagate addition (CPA) we need only one LUT. Thus, the hardware required for a Carry save adder is double than that for a CPA. Besides, the CSA implementation does not take advantage of the carry propagation logic.

In an attempt to use the available carry-logic while keeping an adder maximum delay bounded regardless of the word length, authors from [1] present a solution making use of a high radix carry-save representation. Due to this high radix representation, initially introduced to reduce the number of wires and registers required to store a value, the sum word from a carry-save number is represented in radix- r (i. e. log2r bits per digit) and the carry word requires one only bit per radix- r digit.

This representation allows the use of standard CPAs to add each of the sum word radix- r digits, connecting the carry word to the Carry propagate adder carry-in inputs, hence obtaining the final carry word at the CPA carry-out outputs. When this adder is implemented in an FPGA, we use the whole slice resources, including the carry logic, while increasing the addition delay. However, due to the great optimization of FPGAs carry logic, this delay increase is not very significative if the radix r is not high.

The main drawback in high radix carries save representation is that, the numbers shifts are not an easy task. In this case, complete shifts are only available for radix- r digits, i.e., shifts are only allowed for multiple of r numbers. This restriction comes from the carry word processing, since it is only available at some specific positions within the addition operation. This limitation becomes an important obstacle when applying the high radix carry save representation to many shift and add based algorithms, and even the work presented in [1] has to deal with this problem. For this reason, it is interesting to look for some other ways of using the carry logic when implementing carry save adders.

II. Efficient Mapping Of Carry - Save Adder in FPGA

Two different solutions to obtain a more efficient implementation of carry-save adders on FPGAs than the one presented are shown in this section.

A. Using half of a slice for a 3:2 counter

The first proposed solution makes use of only half of a slice for a 1-bit 3:2 carry-save adder implementation. However, the remaining half of slice cannot be fully used, since the carry bit produced by 3:2 counter computation is feeded into it, disabling a possible use for the rest of the carry propagation logic. In this solution it is not possible to implement two 1-bit
3:2 CSAs within a single FPGA slice. Nevertheless, the free semi-slice resources can still be used by some other type of logic computation which does not need to take advantage of the carry logic. Fig. 2 depicts how this solution is mapped into a slice.

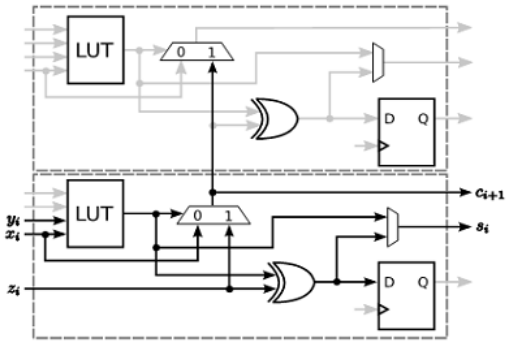

Fig 2. Efficient Slice mapping for 1-bit 3:2 CSA

The main drawback in this case is that the upper semi-slice (the one left free) often remains unused within their application. As a consequence, the area requirements for this approach is higher than the one obtained by the solution described by them. Some other example applications, such as a constant

multiplier and an additive range reduction are developed. Where we have successfully taken advantage of the upper semi-slice using it as a table look-up. From the results obtained, we can conclude that this solution is convenient for those applications where the upper semi-slice can be used.

B. Implementing a 4:2 compressor

To overcome the drawback shown in Section III-A, i.e. we cannot always guarantee a successful use of the upper semi-slice, for example for the commonly used multi operand addition. For this reason, here we propose a new type of mapping where we fully use a whole slice hardware resources. The new approach lies in a 4:2 compressor implementation instead of a single 3:2 counter. Fig. 3 depicts a typical 4:2 compressor scheme based on 3:2 counters, and Fig. 4 shows how this 4:2 compressor can be efficiently mapped into an FPGA slice. In order to achieve this goal, we have to map some parts from the addition of different weighted bits within the same slice. Specifically, the piece of hardwarehighlighted in Fig. 3 is implemented into asingle slice.

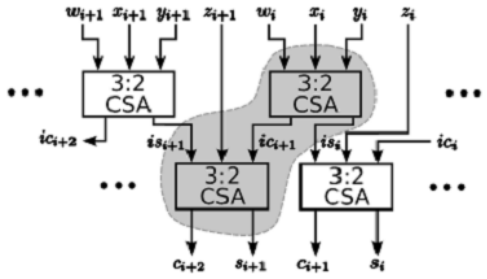

Fig. 3. 4:2 compressor implementation using 3:2 counters

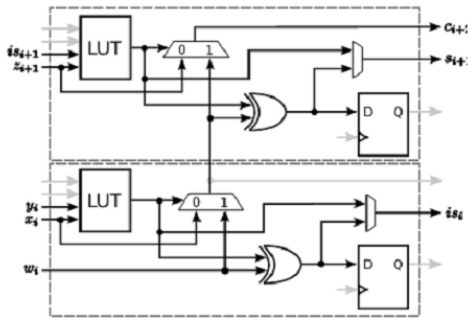

Fig. 4. Mapping of a 4:2 compressor into a slice

The upper semi-slice implements a second level 3:2CSA,whereas the bottom semi-slice is in charge of implementing a first level 3:2 CSA In order to take advantage of the carry propagation logic, a single slice implements the first level addition for bits with weight 2^i and the second level addition for bits with weight 2^i+1. In this way, all the slice resources are used.

III. 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. 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.

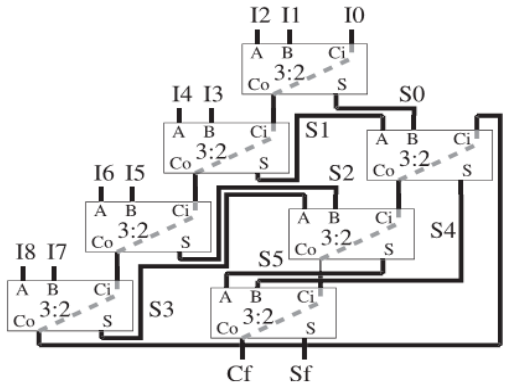

Fig.5. N-bit width CS 9:2 compressor tree based on a linear array of CSAs.

Fig. 5 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 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. 5.

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 (Si) previously generated are then added in order (i.e., the first generated partial sums are added first) as shown in Fig.5.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 Nop-2 of these elements (because each CSA eliminates one input signal).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 Nop-2 levels. This is much more than a classic Wallace tree structure and, thus, a longer critical path. Nevertheless, because 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. 6 (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 [Nop-1]/2 CSAs (which correspond to partial addition of the input operands). This first ETL produces [Nop-1]/2 partial sum-words that are added to a second level of CSAs (together with the last input operand if Nop 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 and the delay of this tree is approximately L times the delay of a single ETL.

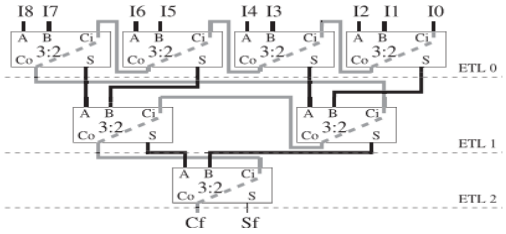

Fig.6. Time model of the proposed CS 9:2 compressor tree.

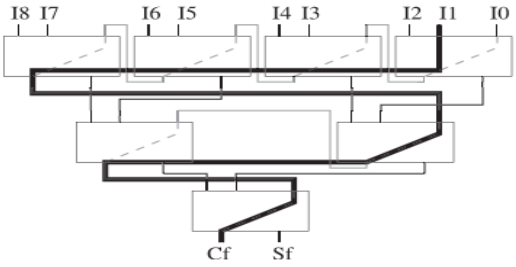

Fig.7. Critical path of the proposed 9:2 compressor tree for linear array behavior.

IV.Simulation Results

To evaluate the performance of the proposed compressing elements, we designed and synthesized pipelined compression trees with eight 10-bit inputs using
different techniques. Even for this relatively low word size, the ternary adder and the 4:2 compressor lead to the best efficiency of Ek =1.8 using k = 10 BLEs. The ternary adder tree requires two stages with four ternary adders in total while the compressor tree with 4:2 compressors requires three stages with three 4:2 compressors in total plus one common two-input adder to merge the result.

V. CONCLUSIONS AND FUTURE WORK

Prefix adder architectures capable of three – operand addition for cell based design and their synthesis have been designed and investigates in this thesis. Binary adders capable of constant addition have also been presented and their performance investigated. The design is possible due to the generation of a new set of intermediate outputs called “flag” bits. This design can be used

as a replacement to carry-save adders with the possibility of having the third operand as a constant or a variable binary number. The hardware will be optimized by gate sizing in order to achieve better performance results.

REFERENCES

[1] J.-L. Beuchat and J.-M. Muller, "Automatic generation of modular mul-tipliers for fpga applications," IEEE Transactions on Computers, vol. 57, no. 12, pp. 1600–1613, December 2008.

[2] J. Detrey, F. de Dinechin, and X. Pujol, "Return of the hardware floating-point elementary function," in Proceedingsof the 18th IEEE Symposium on Computer Arithmetic (Montpellier, France), Kornerup and Muller, Eds. Los Alamitos, CA: IEEE Computer Society Press, June 2007, pp. 161–168.

[3] H. Eberle, G. N., S. Shantz, V. G upta, L. Rarick, and S. Sundaram, "A public-key cryptographic processor for RSA and ECC," in Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP2004), September 2004.

[4] H. R. Ismail, R.C., "High performance complex number multiplier using booth-wallace algorithm," in IEEE International Conference on Semiconductor Electronics ICSE, November 2006.

[5] K. Manochehri and S. Pourmozafari, "Modified radix-2 montgomery modular multiplication to make it faster and simpler," in IEEE International Conference on Information Technology: Coding and Computing, ITCC 2005 , April 2005.

[6]M.D.ErcegovacandT.Lang, Digital Arithmetic. Morgan Kaufmann Publishers, 2004