Convolution Computation in FPGA Based on Carry-Save Adders and Circular Buffers

Carlos D. Moreno[1,4], Pilar Martínez[2,4], Francisco J. Bellido[1], Javier Hormigo[3,5], Manuel A. Ortiz[1], and Francisco J. Quiles[1]

[1] Computer Architecture, Electronics and Electronic Technology Department, University of Córdoba, Spain
[2] Applied Physics Department, EPS, University of Córdoba, Spain
[3] Computer Architecture Department, University of Málaga, Spain
[4] Campus Universitario de Rabanales. 14071 Córdoba, Spain
[5] Campus Universitario de Teatinos. 29080 Málaga, Spain
el1momoc@uco.es

Abstract. In this article, we present some architectures to carry out the convolution computation based on carry–save adders and circular buffers implemented on FPGAs. Carry-save adders are not frequent in the implementation in FPGA devices, since these have a fast carry propagation path. We make use of the specific structure of the FPGA to design an optimized accumulator which is able to deal with carry–save additions as well as carry–propagate additions using the same hardware. On the other hand, this structure of circular buffers allows the convolution computation of two signals with two algorithms of calculation: the input side algorithm and the output side algorithm, in a more efficient way.

Keywords: Convolution, FPGA, carry-save adders, circular buffers.

1 Introduction

There are several methods to describe the relation between the input and output of the linear time invariant systems (LTI), when both are represented according to time. One of the methods to describe it is by means of differential linear equations (in continuous time) or equations in differences of constant coefficients (in discrete time). Another way of representing this relation would be by means of a block diagram which represents the system as an interconnection of three elementary operations: multiplication, addition, and displacement in the time for systems in discreet time, or integration for systems in continuous time. The third form is the description by means of variables of state, which corresponds to a series of differential equations or in differences of the first order connected that represent the behavior of the 'state' of the system and an equation that relates the state to the output.

However, the most widely used method to represent this relation is related to its response to impulse. The response to impulse is the output of the system associated with the input of the impulse. Considering the response to the impulse, we determine

M.L. Reyes et al. (Eds.): IT Revolutions 2011, LNICST 82, pp. 237–248, 2012.

the output due to an arbitrary input, expressing the entry as an exaggerated superposition of impulses displaced in the time. For the linearity and invariance regarding time, the outputs must be an exaggerated superposition of responses to the impulse displaced in time. The term "convolution" is used to describe the procedure in the determination of the output from the input and the response to the impulse.

The response to the impulse is the output of an LTI system due to an input of an impulse applied in time t = 0 or n = 0. The response to the impulse characterizes completely the behavior of any LTI system. This can seem surprising, but it is a basic property of all the LTI systems. The response to the impulse is often determined by the knowledge of the configuration and dynamics of the system, or in case of an unknown system, it can be measured by applying an approximated impulse near the input of the system. The generation of a sequence of impulses in discreet time to prove an unknown system is direct. As regards the case in continuous time, a real impulse of zero width and infinite amplitude cannot be actually generated and is usually physically approximated as a great amplitude and narrow breadth pulse. This way, the response to the impulse can be approximated as the behaviour of the system in response at an entry of a high amplitude and extremely short duration.

If the input for a linear system expresses itself as a weighted superposition of impulses displaced in time, then the output is a weighed superposition of the response of the system to each impulse displaced in time. If the system is also invariant regarding time, then its response to an impulse displaced in time is a version displaced in the time of the response of the system to an impulse. Consequently, the output of an LTI system is given by a weighted superposition of responses to the impulse displaced in the time. This weighted superposition receives the name convolution sum in discrete time systems and of convolution integral in continuous time systems.

2 Convolution Sum

Let us consider the case of discrete time systems. First of all, we express an arbitrary signal as a weighted superposition of impulses displaced in time. The convolution sum is then obtained by applying to an LTI system a signal represented this way.

Let us consider the product of signal x [n] and the sequence of impulses δ [n], expressed in the following way:

$$x[n] \cdot \delta[n] = x[0] \cdot \delta[n] \tag{1}$$

If we generalize this relation for the product of x [n] and the sequence of impulses displaced in time, we obtain the following:

$$x[n] \cdot \delta[n-k] = x[k] \cdot \delta[n-k] \tag{2}$$

In this expression, n represents the index of time, therefore x [n] denotes a signal, while x [k] represents the value of signal x [n] in moment k. We see that the multiplication of a signal by the impulse displaced in time produces an impulse displaced in time with an amplitude given by the value of the signal at the moment when the impulse happens.

This property allows us to represent x [n] as a weighted sum of impulses displaced in time:

$$x[n] = \ldots + x[-2] \cdot \delta[n+2] + x[-1] \cdot \delta[n+1] + x[0] \cdot \delta[n] + x[1] \cdot \delta[n-1] + x[2] \cdot \delta[n-2] + \cdots \quad (3)$$

Which can be briefly written as follows:

$$x[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot \delta[n-k] \quad (4)$$

If we call the impulse response of the system h [n], the output y [n] of the system to an input x [n] is the convolution of both signals:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k] \quad (5)$$

This expression is called the *convolution sum*. This formula indicates that for calculating the convolution, we need four steps: reflection of h [k] in order to obtain h [-k], shift to obtain h [n–k], multiplication, and addition. The reflection operation is performed only once, however, the other three are repeated for all possible values of the displacement.

3 Algorithms for Calculating Convolution

Convolution is a formal mathematical operation that takes two signals and produces a third signal. In linear systems, the output signal y [n] of the system is obtained from the convolution of the input signal x [n] and from the response h [n] to the impulse function δ [n] of the system.

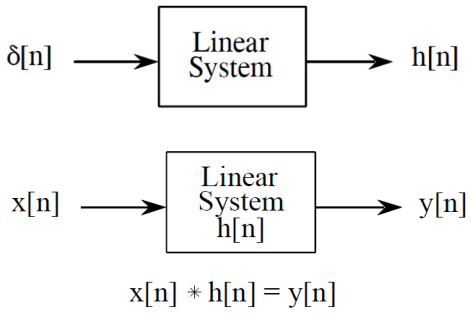

Fig. 1. Physical direction of convolution: the output of a linear system is obtained from the convolution of the input sign and the response to the impulse of the system

In most digital signal processing applications, the input signal is around some hundreds, thousands, or a few million samples in length. However, the impulse response is much shorter, around a few hundred samples. The mathematical operation of

convolution does not restrict the length of these signals, however, it determines the length of the output signal, which is the sum of the length of the two input signals minus one.

In digital signal processing, convolution can be interpreted in two ways: from the point of view of the input signal and from the point of view of the output signal [4]. From the point of view of the input signal implies the analysis of how each sample of the input signal contributes in many values of the output signal, and it provides a conceptual understanding on how convolution affects the signal digital processing. From the point of view of the output signal, it evaluates how each sample in the output signal receives information from many samples of the input signal. Let us consider for example the convolution of two signals x (n) and h (n) with a length of M = 5 and N = 4

$$x(n) = \{x_4, x_3, x_2, x_1, x_0\} \quad \text{length } M = 5$$
$$h(n) = \{h_3, h_2, h_1, h_0\} \quad \text{length } N = 4$$

For definition of discrete convolution:

$$y[i] = x[n] * h[n] = \sum_{i=0}^{M+N-1} x[j] \cdot h[i-j] \tag{6}$$

with length M + N – 1 = 5 + 4 – 1 = 8.

Each of the samples of the output signal y (n) = {y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0}, would be calculated as follows:

$$\begin{aligned}
y_0 &= x_0 \cdot h_0 \\
y_1 &= x_0 \cdot h_1 + x_1 \cdot h_0 \\
y_2 &= x_0 \cdot h_2 + x_1 \cdot h_1 + x_2 \cdot h_0 \\
y_3 &= x_0 \cdot h_3 + x_1 \cdot h_2 + x_2 \cdot h_1 + x_3 \cdot h_0 \\
y_4 &= x_1 \cdot h_3 + x_2 \cdot h_2 + x_3 \cdot h_1 + x_4 \cdot h_0 \\
y_5 &= x_2 \cdot h_3 + x_3 \cdot h_2 + x_4 \cdot h_1 \\
y_6 &= x_3 \cdot h_3 + x_4 \cdot h_2 \\
y_7 &= x_4 \cdot h_3
\end{aligned} \tag{7}$$

3.1 The Input Side Algorithm

The first point of view (the input side algorithm) performs the calculation of the convolution of Equations (7) vertically (by columns). It analyzes how each sample in the input signal affects several samples of the output signal, i.e., it takes x_0 and multiplies it by all samples of the response to the impulse h_3, h_2, h_1, h_0. In the first cycle, it would calculate $y_0 = x_0 \bullet h_0$, then it calculates the product terms of each sample output: $x_0 \bullet h_1$, $x_0 \bullet h_2$, $x_0 \bullet h_3$, etc. Subsequently, it takes x_1 and carries out the product for all h entries, i.e., it calculates $x_1 \bullet h_0$, which when added to its first already calculated term product, obtains the second sample of the output signal y_1, and then, it calculates $x_1 \bullet h_1$, $x_1 \bullet h_2$, $x_1 \bullet h_3$, etc. Then, it continues the operations with all samples of the input signal.

This point of view of convolution is based on the fundamental concept of signal digital processing: it decomposes the input, passes the components through the system, and synthesizes the output.

3.2 The Output Side Algorithm

The second point of view (the output side algorithm) performs the calculation of the convolution of Equations (7) horizontally (by rows). It examines how each sample of the output signal receives information from many points of the input signal. This viewpoint describes the mathematics of the convolution operation. The value of each sample of the output signal can be calculated regardless of the value of the other samples.

4 Architecture MAC/CSA

The convolution function is the key to many signal digital processing applications (filtering, FFT, correlation, neural networks, etc.). It is based on multiplication and accumulation (MAC), and this is why many high-performance FPGA devices have added special hardware to perform these operations. However, this is not to be found in low-cost FPGA devices.

To reduce the cost of the multiplications involved in applications with constant terms, a constant multiplier is usually implemented or distributed arithmetic is used. However, in other applications, multiplication cannot be avoided, due to which the FPGA device manufacturers use embedded multipliers to efficiently perform this operation.

The convolution involves many multiplications and accumulations. If an FPGA device has few multipliers, it is common to use them in an iterative way to implement a function. To increase performance, it is necessary to reduce cycle time, especially when many operations need to be performed. As the size of the multiplier is fixed in the FPGA and the accumulator is defined by the user, a suitable design of the hub will lead to further optimization of the final operation.

Many authors have implemented different designs for the output carrying of the convolution operation. Current FPGA devices include fast carry logic, which allows the implementation of adders with carry spread (CPA) fast. More specifically, the path for the propagation of the carry has been specially optimized so that it is a basic element for the next stage, and so that together with the carry logic, and spread the value of the carry. For this reason, we prefer the carry propagated adders (CPA) against the stored carry adders (carry save adders, CSA). For not very long word lengths CSAs and CPAs have similar delays, but CPA uses twice the amount of logical resources.

Current FPGAs are primarily designed for signal digital processing applications which involve quite smaller operands (16 to 32 bits). FPGA manufacturers decided to include additional carry logic for the implementation of fast carry-ripple adders for these operation sizes. Let us consider, for example, FPGA Xilinx Spartan-3 (low cost). Figure 2 describes the simplified architecture of a slice of this device [4], which is the main logical resource to implement combinational and sequential circuits.

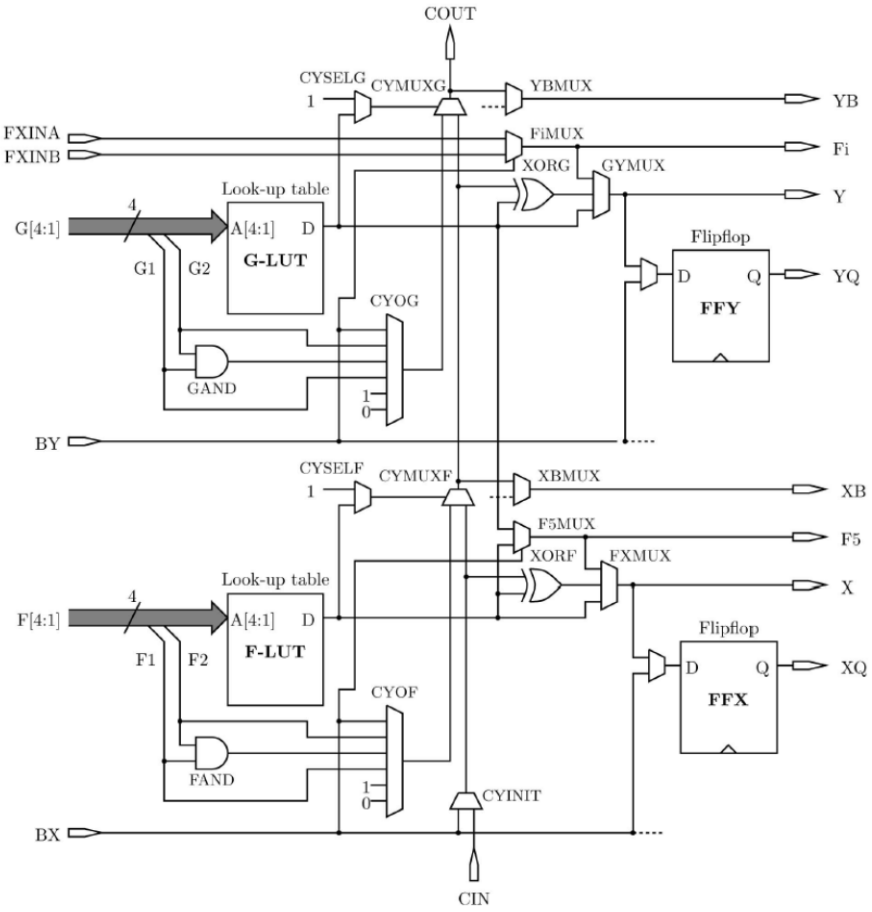

Fig. 2. Simplified diagram of a slice of a Spartan-3 FPGA device [4]

Each slice includes two four-input function generators (lookup table, G-LUT, and F-LUT), two flip-flops (FFY and FFX), carry logic (CYSELG, CYMUXG, CYSELF, CYMUXF, and CYINIT), logic gates (GAND, FAND, XORG AND XORF), and multiplexers with multiple functions. Each function generator is implemented using a programmable lookup table (LUT).

It is necessary to remember that a full adder (FA) is a combinational circuit with three input bits (the bits to be added x_i and y_i, and the input carry c_{in}) and two output bits (the sum bit s_i and the carry output c_{out}).We have that $s_i = x_i \oplus y_i \oplus c_{in}$, and output carry $c_{out} = x_i$, si $x_i = y_i$, and $c_{out} = c_{in}$ in any other case. Let us suppose that F-LUT calculates $x_i \oplus y_i$, then XORF door obtains the s_i sum bit, whereas the c_{out} output carry calculation involves three multiplexers (CYOF, CYSELF, and CYMUX). The s_i sum bit is sent to another slice (X output) or stored in the FFX flip-flop. The G-LUT could implement a second adder within the same slice, thus integrating two bits of a CRA adder. Some authors [1] state that in a conventional CSA adder, it is impossible

to implement two full adders with independent input carries in the same slice, since each slice has only one carry input, which would require then double of the hardware resources. Therefore, the hardware design tools assign two slices when a VHDL description of de s_i and c_{out} is carried out, without using a G-LUT.

Figure 3 describes the simplified architecture of a slice implementing a Carry Propagate adder (CPA).

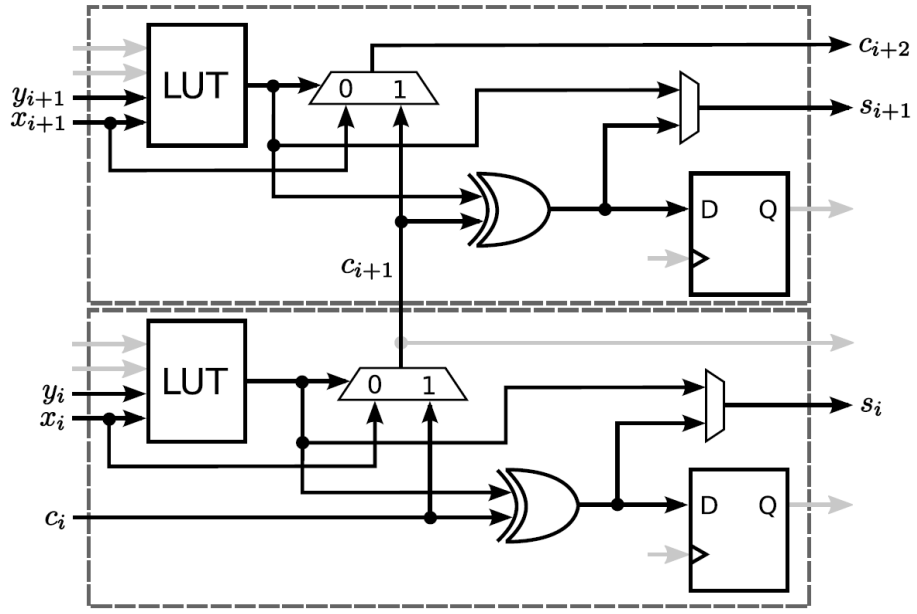

Fig. 3. Simplified diagram of a slice implementing a CPA

In summary, for a 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) only needs one LUT. Thus, the hardware required for a CSA 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. log2 *r* bits per digit) and the carry word requires one only bit per radix-*r* digit, as shown in [6].

Using this representation allows the use of standard CPAs to add each of the sum word radix-*r* digit, connecting the carry word to the CPA carry-in inputs, hence obtaining the final carry word at the CPA carry-out outputs. Thanks to this, when this adder is implemented in an FPGA, we make use of the whole slice resources, including the carry logic, at the expense of increasing the addition delay. However, due to the great optimization of FPGAs carry logic, this delay increase is not very significant if the radix *r* is not high.

However, the high radix carry save representation shows an important drawback, i. e., numbers shifts are not an easy task. In this case, complete shifts are only available for radix-*r* digits (groups of log2 *r* bits), 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.

The authors in [2] show two different solutions for a more efficient implementation of FPGA-CSA than those presented in [1]. This article shows that it is possible to use half of the slice to implement other functions such as a 3:2 counter or 4:2 compressor.

5 Circular Buffer Architecture for Convolution Algorithms

The architecture proposed in [6] presents an iterative architecture for convolution computation which is faster than previous implementations. It is achieved by using redundant arithmetic. In spite of redundant arithmetic, which involves an increase of hardware, we have developed a technique which allows to reuse the hardware resources to obtain the final result in conventional arithmetic. The architecture that we propose for convolution is based on an iterative use of the embedded multipliers presented in most of modern FPGA devices.

In this article, a combined CSA-CPA accumulator was proposed using the carry logic when implementing carry save adders. This combined CSA–CPA accumulator requires the hardware of one CSA accumulator only, without adding additional penalty time. However, the way in which the records where the signals which were going to be convolved were stored was not described.

Now we propose a new architecture based on the previous one [6], which includes circular buffers to record the samples of the input or output signals depending on the algorithm that we use for the calculation: from the view point of the input (input side algorithm) or from the point of view of the output (output side algorithm).

The output side algorithm examines how each sample of the output signal receives information from many points of the input signal. This viewpoint describes the mathematics of the convolution operation. You can calculate the value of each sample of the output signal regardless of the value of the other samples. The code for this algorithm is as follows:

```
For i=0 to M+N-1
        y[i] = 0
        For j=0 to N
        If (i-j ≥ 0)
        If (i-j < M)
                y[i] = y[i] + x[j] * h[i-j]
        End for
End for
```

The architecture which we propose would be the one shown in Figure 4.

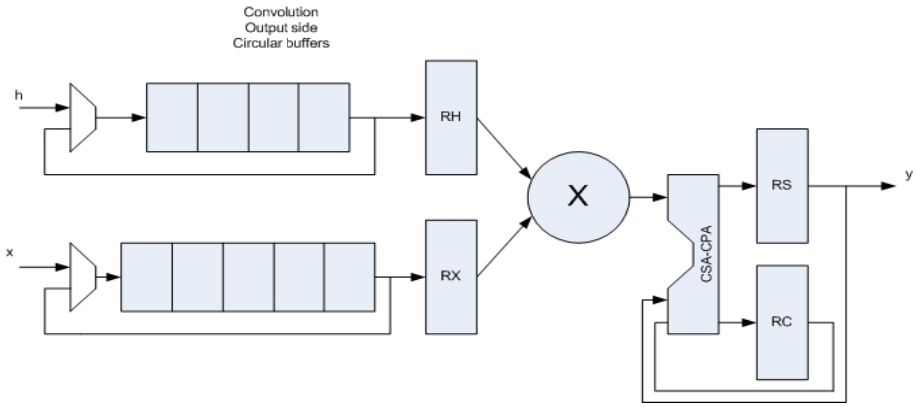

Fig. 4. Proposed architecture from the output side algorithm

In this architecture, we have been introduced two circular buffers for the samples of each input signal x [n] h [n] that are stored in registers RX and RH, a multiplier carries out the product and in order to execute the sum, we used the combined CSA-CPA accumulator as described below in Figure 5.

Here, the functionality of the combined CSA–CPA accumulator (box entitled CSA&CPA of Fig. 4) appears, where the gray lines represent the signals that do not participate in the operation. The left figure shows the accumulator working in CPA mode, whereas the right figure shows CSA mode. X_i represents a bit of the output of the multiplier, whereas C_i and S_i represent a bit of the carry and sum words, respectively. The input Sel selects either the CSA mode (Sel=0) or CPA mode (Sel=1).

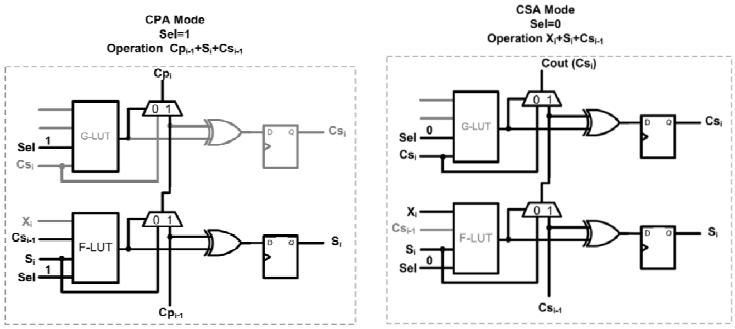

Fig. 5. Combined CSA-CPA accumulator

In CSA mode, the F-LUT is in charge of performing the addition of X_i and S_i, and the lower XOR gate operates to have the final sum bit (stored in the corresponding flip-flop). The carry bit is transmitted through the lower multiplexor and stored in the upper register, since Sel=0 forces the output of the G-LUT to zero. Also, Cs_i is transmitted using the upper mux. In CPA mode, the sum bit is generated from Cs_{i+1},

S_i, and Cp_{i-1} by means of the F-LUT and the associated XOR gate, whereas the carry is transmitted to the next slice by crossing both multiplexors (the output of the G-LUT is forced to one if Sel=1).

The input side algorithm analyzes how each sample in the input signal affects several samples of the output signal. The code for the implementation of this algorithm would be as follows:

```
For i=0 to M+N-1
For j=0 to M
        y[i+j] = y[i+j] + x[i] * h[j]
End for
End for
```

The architecture which we propose for this algorithm is shown in Figure 6:

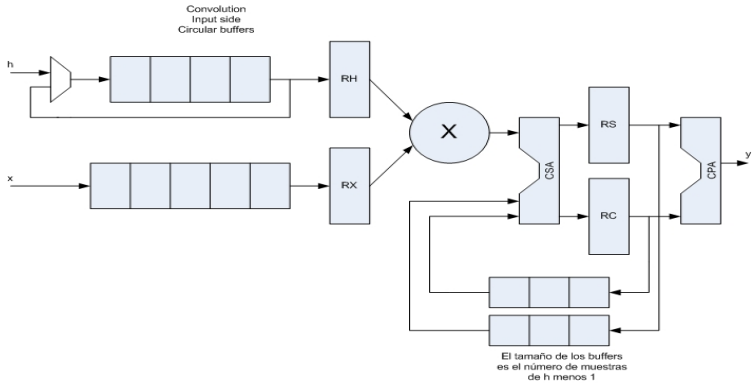

Fig. 6. Proposed architecture from the input side algorithm

In this case, the CPA-CSA combined accumulator has been changed by a CSA accumulator and two buffers for the outputs of the registers of the sum and the carries (RS and RC, respectively), but in order to get the final sum, it is necessary to add a conventional CPA adder to perform the final sum, as determined by the carry-save arithmetic. We can see that the iterative structure uses the RC and RS registers to allocate the sum and carry words in each iteration, and a final CPA is required to convert the result from redundant to conventional representation. The length of the RS and RC buffers should be the number of samples of h [n] minus one.

Otherwise, Spartan-3 FPGAs can configure the look-up table (LUT) in a slice as a 16-bit shift register without using the flip-flops available in each slice. Therefore, each input or output buffer uses a lookup table if it consists of 16 samples or less, two LUTs if the samples are between 16 and 32, and so on.

These designs have been simulated, evaluated, and synthesized using Mentor Graphics ModelSim, Xilinx ISE 9.1i, and the Xilinx Spartan XC3S500E-5 FPGA device. Even though new FPGA models have recently emerged, the LUT-based models studied in this paper are currently the most widely used models for low cost

applications. The proposed implementations use the embedded multipliers which can be found in most modern FPGA devices. In this way, the time and resources of the system are optimized. The implementation results for the convolution of 18 bits input data are shown in Table 1. This architecture use one 18x18 embedded multiplier and one 48 bit accumulator implemented using standard logic.

Table 1. Implementation results for 18x18 multiplier and 16 samples

Architecture	Fq (MHz)	LUTs
C-CPA	114	84
Input side	149	168
Output side	149	132

The number of lookup tables shown in Table 1 corresponds to the case of a number of samples of 16 or less. In the case that h [n] or x [n] is larger than 16, 18 x 2 LUT must be added to the results for each number of samples that exceed a multiple of 16. Besides, in the case of the input side implementation, another 18x2 LUTs must be added for each number of samples of h [n] that exceeds a multiple of 16, since the size of the output buffers is the number of samples of h minus one.

6 Summary and Conclusions

In this paper, we have presented several architectures to deal with convolution computation, using CSA to speed up the computation, and which are specially adapted to the inner architecture of FPGA devices. The designs have been tailored to the FPGA structure which allows to take full advantage of the FPGA resources. The architectures are compared from a qualitative and quantitative point of view and the implementation results over a specific FPGA are presented.

We have proposed a combined CSA-CPA architecture for the output side algorithm based on reusing the hardware required by the CSA in such a way than no CPA needs to be implemented in hardware for the last conversion from redundant to conventional arithmetic. This leads to the same computation time than that using CSA plus a final CPA for the input side algorithm. In other words, the final conversion from redundant to conventional arithmetic has no hardware cost for the proposed combined CSA-CPA architecture. On the other hand, the use of circular buffers does not add significant delays for the final calculation, though it does add major resources of the FPGA.

From the experimental results, we conclude that both CSA based architectures presented have the best time results (about 30% speed up for both), and the combined CSA-CPA achieves an important hardware reduction in comparison with the CSA plus CPA architecture. When larger input data is used and high speed is demanded, it is necessary to pipeline the multiplier. In this case, the speed up keeps and the extra hardware required by the CSA has a relatively lower weight in the area of the full system due to the large hardware required by the pipelined multipliers (only 8% more hardware than a pure CPA design with almost 30% speed up for 36 bit input data).

References

1. Beuchat, J.L., Muller, J.M.: Automatic generation of modular multipliers for FPGA applications. IEEE Transactions on Computers 57(12), 1600–1613 (2008)
2. Ortiz, M.A., Quiles, F.J., Hormigo, J., Jaime, F.J., Villalba, J., Zapata, E.L.: Efficient implementation of carry–save adders in FPGAs. In: 20th IEEE International Conference on Application-specific System, Architectures and Processors, ASAP 2009, pp. 207–210 (2009)
3. Ercegovac, M.D., Lang, T.: Digital Arithmetic. Morgan Kaufmann Publishers (2004)
4. Steven, W.S.: Digital Signal Processing, a Practical Guide for Engineers and Scientists. Elsevier Science (2003)
5. Xilinx, Spartan-3 FPGA Data Sheet, http://www.xilinx.com/support/documentation/spartan-3.htm
6. Moreno, C.D., Quiles, F.J., Ortiz, M.A., Brox, M., Hormigo, J., Villalba, J., Zapata, E.L.: Efficient Mapping on FPGA of Convolution Computation based on Combined CSA-CPA Accumulator. In: 16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2009, pp. 419–422 (2009)