# Constant Coefficient Convolution Implemented in FPGAs

## Abstract

*This paper reviews different architectural solutions for calculating constant coefficient convolution operation in FPGAs and. At first, different architectures of multipliers are approached, as the multiplication is the most complex operation performed in the convolutions. Nevertheless, disregarding the multiplier entity allows for further circuit optimisations, therefore Look-Up-Table (LUT) based Convolver (LC) versus the sum of the LUT-based Multipliers is described. Further, an alternative technique - (Parallel) Distributed Arithmetic Convolver (DAC) is approached. The key issue of this paper is, however, a novel architectural solution: Irregular Distributed Arithmetic Convolver (IDAC) which, in comparison to the DAC, has an irregular form, and therefore allows for better circuit optimisation. All architectural solutions described hereby can be automatically generated by the Automated Tool for generation Convolvers in FPGAs (AuToCon).*

## 1. Introduction

An *N*-tap convolution (FIR filter) can be expressed by an arithmetic sum of products:

$$y(i) = \sum_{k=0}^{N-1} h(k) \cdot x(i-k) \qquad (1\text{-}1)$$

*where: y(i), x(i) and h(i) represent response, input at the time i and the convolution coefficients, respectively.*

The convolution is usually carried out employing separate multipliers and finally an adder. An example of a convolver is shown in Figure 1-1.

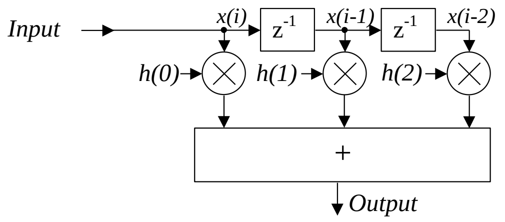


**Figure 1-1. An example of the 3 tap convolver**

Multiplication is the most complex operation in the convolver, and several techniques have been adopted to perform it more efficiently. First at all, coefficient values are usually constant, therefore the values of the coefficient can be built-in the circuit, this solution is further denoted as Constant Coefficient Multiplier (KCMs). The KCM occupies 17÷23% on average or 29÷41% on maximum [1], area of the fully functional, Variable Coefficient Multiplier (VCM). Consequently, the KCM should be implemented whenever the coefficient values are constant. Alternatively, when coefficients are changed infrequently, a part of the Field Programmable Gate Array (FPGA) can be reconfigured in order to change the coefficient. FPGA can be reconfigured in a few milliseconds time, nevertheless a new multiplier circuit must be redesigned and re-implemented, which is much more time-consuming than FPGA reconfiguration. Therefore this approach can be practically adopted provided that only a finite number of coefficient values are allowed. In this case every coefficient value has a separate pre-implemented entry which can be quickly download into the FPGA.

FPGAs implement logic cells as a Look Up Table (LUT) memory, therefore the inherent way of performing multiplication seems the LUT based Multiplication (LM) [2, 3] where the value of the coefficient is coded into the contents of the LUT memory.

Distributed Arithmetic Convolver (DAC) [4, 5] is an alternative way for performing convolution. The DAC similarly like the LM employs LUT memory however the multiplication is carried out in a bit-plane order. Consequently all inputs to the DAC LUTs are at the same bit-significance and therefore the width of the LUT data bus is smaller. The DAC has a regular structure – the same for different input bit-significance, and this causes that grouping the inputs to the LUT is constrained. Consequently a novel design approach Irregular Distributed Arithmetic (IDAC) is proposed for which LUT inputs can be different for different bit-significance.

## 2. Constant Coefficient Multipliers (KCMs)

### 2.1 Multiplierless Multiplication (MM)

The KCM is usually implemented in a multiplierless fashion by using only hardwired shifts and adders from the binary representation of the multiplicand. For example, *A* multiplied by $B = 14 = 1110_2$ can be implemented as *(A<<1)+(A<<2)+(A<<3)*, where '<<' denotes a shift to the left. It should be noted that the hardware requirements depend on the choice of the coefficient, i.e. the number of 1's in the binary representation of the coefficient should be as low as possible.

**Canonic Signed Digit (CSD) Representation**

This area reduction technique attempts to reduce the number of 1s required in the coefficient's two's complement representation by the use of Canonic Signed Digit (CSD) representation [6, 7]. The CSD representation is a signed power-of-two representation where each of the bits is in the set $\{0, 1, \bar{1}\}$ (0 – no operation, 1 – addition, $\bar{1}$ – subtraction). It should be noted that the general conversion algorithm to CSD [6, 7] considers addition and subtraction to be the same cost operations. However, for a subtraction in the case when the subtrahend is shift to the right with respect to the minuend, the LSB cannot be directly copied to the output as it is the case for the addition. Therefore a modified conversion algorithm has been proposed in [3], for which a $\bar{1}$ is introduced only when the total number of non-zero symbols is reduced. In this paper, the modified CSD conversion algorithm is implemented.

**Substructure sharing**

Additional area reduction can also be achieved by Substructure Sharing (SS) [8]. For example, multiplication by $27=11011_2$ can be implemented by the use of an intermediate variable *tmp*, as it is shown in the following equations: $tmp= a + (a<<1)$, and $27 \cdot a= tmp + (tmp<<3)$. By the use of the SS the number of required additions has been reduced from 3 to 2.

It should be noted that the SS area-reduction may be implemented also on the CSD, therefore the combination of the SS and CSD techniques should be also considered during the optimisation process.

## 2.2. LUT based Multiplier (LM)

In FPGAs, a multiplication can be carried out employing Look-up-table (LUT) based multiplication (LM). In principle, the evaluation of any finite function can be carried out using a look-up table (LUT) memory that is addressed with the argument for the evaluation and whose output is the result of the evaluation. Unfortunately, the use of a single LUT for the multiplication is unlikely to be practical for any but the smallest argument, because the table size grows rapidly with the width of the argument. Therefore the solution is to split the argument, use LUTs, and then use a tree of adders [2, 3, 4]. An example of this is given in Figure 2-1.

For the LM some optimisation can be achieved [3]. To further describe these optimisation techniques, an example of LUT contents for the multiplication $Y= 19 \cdot X$ is given in the example in Table 2-1.

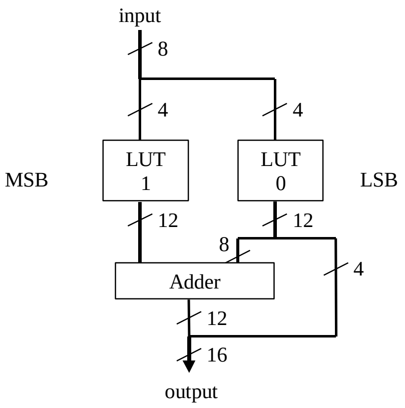


**Figure 2-1. Look-up table based multiplication for 8-bit wide coefficient**

It can be easily proved that an output bit of the LUT depends only on the address bits which weights are lower or equal to the output bit weight. In the example, the memory cell $y_0$ depends only on the address line $a_0$, memory cell $y_1$ depends on $a_0$ and $a_1$, etc. In general, an output bit $y_i$ depends on the $MAX(i+1, n)$ address lines, where $n$ denotes the width of the LUT address bus. In consequence, $(n-1)$ LSBs require smaller memory modules, which implies substantial hardware savings. These hardware savings will be denoted as LSBs Address Width Reduction (LAWR).

| Address | Value | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 19 | 0 | 1 | 0 | 0 | 1 | 1 |
| 2 | 38 | 1 | 0 | 0 | 1 | 1 | 0 |
| 3 | 57 | 1 | 1 | 1 | 0 | 0 | 1 |
| address width | | 1 | 1 | 2 | 2 | 2 | 1 |

**Table 2-1. The contents of the memory ($y_5$-$y_0$) for different address values and the coefficient equal 19. Address width – the width of address bus for each memory cell**

An additional decrease of the address width may be observed when the contents of the memory do not depend on a curtain address line. This address width reduction cannot be generalised and differs for different coefficient values and LUT address widths. Therefore, a complex search algorithm has to be employed to find a don't-care address line. This saving is denoted as Don't-care Address Width Reduction (DAWR). In the example given in Table 3-3, the DAWR is observed for memory cells $y_5$ and $y_4$. It should be noted that the DAWR usually occurs for MSBs of the product.

Further savings can be achieved by Memory Sharing (MS). In the given example, memory cells $y_0$ and $y_4$ are the same therefore only one of them is needed. This optimisation requires similar complex search as the DAWR does.

The XC4000 family [9] incorporates basically only 16×1 RAMs and 32×1 RAMs, however the latest occupies twice the area of a 16×1 RAM, and therefore it is not recommended. Figure 2-1 shows a very simple example for which input is 8 bit-wide, therefore the split of the input is rather intuitively selected to be 4+4= 8. In general case, optimal splitting of the input argument is much more complicated [3]. In addition to 16×1 and 32×1 small distributed memories, Virtex family [9] incorporates several large BlockSelectRAM (BSR) memories which are 4 kb in size and may have different data width: 4k×1, 2k×2, 1k×4, 512×8, 256×16. The area in silicon, occupied by a BSR is equivalent to roughly 16 Virtex CLBs or 64 LEs (a Logic Element (LE) is approximately equivalent to a single 16×1 LUT). However the actual cost (area) of these memories differs with respect to available FPGA resources. For example, a design does not implement any BSRs but uses all available CLBs, therefore it is recommended to allocate more logic into the BSRs.

Consequently, a trade-off between the distributed RAMs and the BSRs is design-dependent, and the actual cost of each memory module should be specified independently for different designs. This, however, complicates circuit optimisation which should consider variable cost of different memory modules, adders and flip-flops, and generate a circuit with the lowest cost. The optimal circuit will differ with respect to the cost-relation between basic elements therefore the optimisation procedure cannot make any circuit presumptions which significantly complicates the procedure.

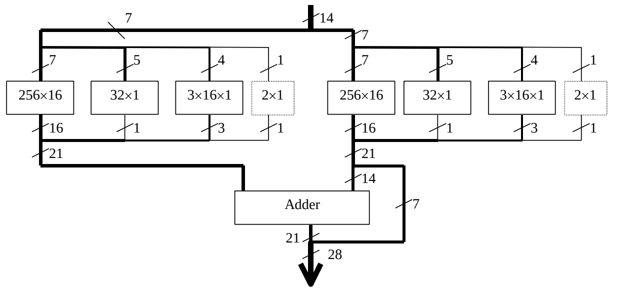


**Figure 2-2. A LM for input and coefficient width equal 14**

As a result, an exhaustive search algorithm (with some obvious simplifications) has been implemented [3], therefore the BSRs together with the distributed RAMs and adders are combined and the best circuit taken. In order to illustrate considered architectures, an example of the LM for input and coefficient width equal 14 is shown in Figure 2-2. The given example shows only the LAWR optimisation and therefore may be even more complicated if a concrete coefficient value is given, for which the MS and DAWR optimisations are implemented. It should be noted that the exhaustive search algorithm can be smoothly implemented only for a multiplier, and in the case of the convolver the number of possible solution is prohibitively large, which causes that a novel design approach should be developed.

## 2.3. Comparison of the KCM

### 2.3.1. Area

In Section 2.1 and 2.2 two different multiplication techniques have been presented: the multiplierless multiplication (MM) and the LUT based multiplication (LM). Therefore a question arises which of them is more hardware efficient. The statistical cost-relation between the MM and LM for XC4000 is shown in Figure 2-3. Accordingly, the LM is usually more attractive for the input and coefficient width less than 5, for the greater widths a better result is usually obtained by the use of the MM. It should be noted that the choice of the best architecture depends on the actual coefficient value and Figure 2-3 shows only statistical relationship. Therefore both architectures should be considered and the best of them chosen for an individual coefficient. However, experimental results show that the gain from considering the best of the LM and MM is insignificant for $K$ greater than 5. Therefore for $K>5$ only the MM should be employed.

The general conclusion can be drawn from Figure 2-3. The MM optimisation techniques (CSD and SS) are more and more efficient with the increase of width $K$. Therefore for greater $K$, the MM is getting more and more attractive in comparison to the LM.

The next question is how much hardware reduction is achieved by the use of the DAWR and MS for the LM. Experimental results show that the reduction is on average 5÷20% depending on the input width $K$.

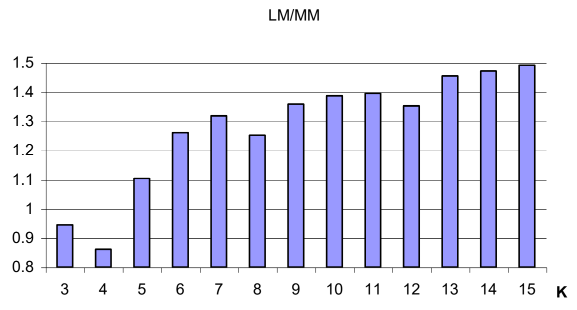
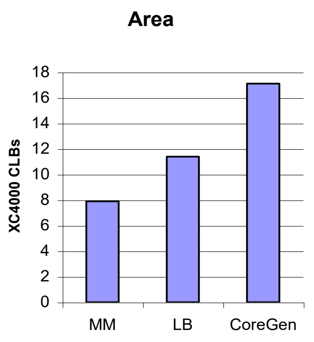


**Figure 2-3. Relation between average area of XC4000 occupied by the LM vs. MM. Results for the different input width K (input range $0 \div 2^K-1$) and coefficient values $1 \div 2^K-1$**

### 2.3.2. Speed

In the previous section only area occupied by the multipliers has been considered. However, relation between the design cost and speed should be also considered. Consequently in order to increase the design throughput, design pipelining has been implemented. FPGAs incorporate a flip-flop (FF) after each logic cell. Therefore conceptually design pipelining can be implemented without any hardware overheads. However, some design paths do not require any logic, therefore frequently FFs have to be inserted without associated logic (according to cut-set method [7]). In consequence, for a fully pipelined circuit (a flip-flop inserted after every logic element), the area is defined by the number of flip-flips rather than the number of logic cells, and as a result, there is a pipelining overhead of about 0÷50%. This overhead disappears if the number of pipeline stages is decreased (flip-flops are not inserted after every logic cell) but in consequence the circuit speed decreases. Conversely, design pipelining considerably increases the throughput, therefore the design efficiency [7] is usually improved and therefore the slight hardware overhead can be neglected. It should be noted that the design pipelining has been also taken under consideration when searching for the optimal architecture. For example, the sub-structure sharing architecture tends to incorporate more flip-flops than the CSD architecture.



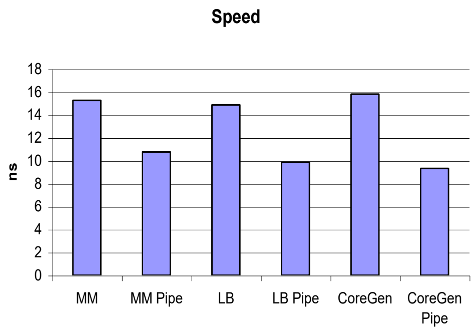


**Figure 2-4. Average area without pipelining and system period without and with pipelining for the MM, LM and Core Generator [10] multiplier. Implementation results for XC4000E-1 and for the 8-bit unsigned input and 5 randomly chosen coefficients**

Figure 2-4 shows average hardware requirements and the system clock for the MM and LM multipliers. It can be seen that the MM multipliers are generally more hardware efficient than the LM counterparts. Besides, the MM and LM developed during the course of this work, surpass the multipliers generated by Core Generator [10] – a commercial program.

## 3. LUT based Convolver (LC)

The structure of the constant coefficient LUT based Convolver (LC) is similar to the sum of products obtained employing LUT based multipliers (LM). However to optimise the structure of the adders, all additions are performed within a single adders block, therefore multiplier entities are disregarded. To illustrate savings obtained by the use of the LC instead of the sum of the LMs, an example is given in Figure 3-1, for 2-tap convolution and 8×8 multipliers.

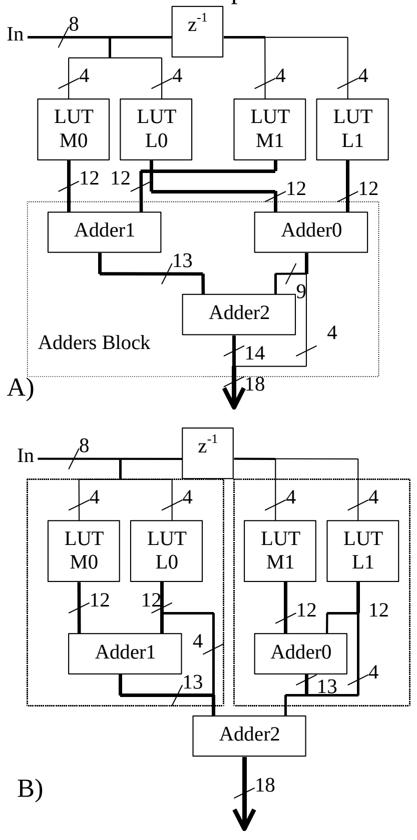


**Figure 3-1. The structure of the convolver for Y= $A \cdot B_0 + z^{-1} \cdot A \cdot B_1$ for input and coefficient width K= 8. A) LC, B) sum of LM**

Let us consider savings obtained by disregarding the multiplier bounds, for LUT output width equal *w= 12* and LUT address width (shift between the same multiplier LUTs) *s= 4*. For the LM, the adder width within the multipliers (Adder0 and Adder 1 in Figure 3-1B) equals roughly *w*. The final adder (Adder2) width equals roughly

$w+s$. Therefore total number of 1-bit adders for the sum of the LM is equal

$$w_{LM}= 3\cdot w+s. \quad (3\text{-}1)$$

For the LC, three adders of width equal $w$ are employed, and therefore the total number of Full / Half Adders is equal

$$w_{LC}= 3\cdot w. \quad (3\text{-}2)$$

A penalty factor $p$, a result of employing the sum of LMs instead of the LC, is roughly

$$p = \frac{w_{LM} - w_{LC}}{w_{LC}} = \frac{s}{3\cdot w} \quad (3\text{-}3)$$

It should be also noted that employing the LC rather than the sum of LMs reduces the maximum width of the adders from roughly $w+s$ to $w$, and therefore reduces maximum propagation time.

Figure 3-1 shows 2-tap convolver which is a very simple example. In general case, the adder network is more complicated. Therefore finding an optimal adder tree, i.e. the netlist of adders for which every adder has only two inputs and the total sum of adder widths is the lowest, is a difficult task, which cannot be solved in an intuitive way. Consequently, different optimisation algorithms, such as an Exhaustive Search, Greedy Algorithm, Genetic Programming and Simulated Annealing, have been implemented [11, 12]. As a result, the Greedy Algorithm should be chosen whenever the circuit-generation time is an important factor, otherwise the Simulated Annealing is recommended as about 10-20% adders area reduction is obtained in comparison to the Greedy Algorithm [12].

The LC employs the sophisticated optimisation algorithm for the adder tree. However, the LC requires also optimisation of LUT memory, esp. when different memory modules can be used (see Figure 2-2). For the LM, which requires few LUT memories and rather a small adder tree, the exhausted search algorithm has been used. Therefore the adder tree and LUT memories together with optimisation techniques described in Section 2.2, are optimised all together. Unfortunately, for the LC, this optimisation technique is impractical to be implemented. Consequently for the LC, only local exhausted search optimisation is implemented, for which each multiplier (the LUT memories and associated adder tree) is optimised separately using the exhausted search technique. Then, all adder trees associated with every multiplier are merged into a single adder tree which is then separately optimised by the techniques described in [11, 12].

In addition, optimisation techniques characteristic only for convolvers are employed.

**Similar Coefficients Optimisation (SCO)**

FIR filters are very often implemented as linear phase filters, for which the impulse respond is symmetric. By exploiting this symmetry, the number of multipliers can be nearly halved through mirroring of the signal flow graph in the point of symmetry of the coefficients [7]. Nevertheless, different symmetries and coefficient combinations can be used, especially for 2D filters [13]. Therefore, the Automated Tool for generation Convolvers in FPGAs (AuToCon) [15] compares all coefficients and groups them into similar coefficients blocks. Coefficients grouped together can be shifted and negated. Grouped inputs are shifted in respect to the coefficient value, and then added (subtracted). Finally, a single multiplier is only implemented. This method allows for reducing the number of multipliers.

For example, for the filter:

$$H(z)= H_1(z) + 5\cdot z^{-i} - 5\cdot z^{-j} - 10\cdot z^{-k} + 20\cdot z^{l} \quad (3\text{-}4)$$

similar coefficient inputs are added:

$$A_5= z^{i} - z^{j} - 2\cdot z^{k} + 4\cdot z^{l}, \quad (3\text{-}5)$$

and the final result is:

$$H(z) = H_1(z) + 5\cdot A_5. \quad (3\text{-}6)$$

In this example the number of multipliers has been reduced by 3.

**Pipelining Optimisation**

The AuToCon generates a convolver with a sophisticated pipelining architecture, for which an additional pipelining parameter $p$ defines maximum number of logic elements between pipelining registers. Figure 3-2a shows an example of a convolver with a straightforward pipelining architecture. For this method additional pipelining registers are often required to compensate different pipelining delays. To reduce this drawback, the pipelining optimisation is implemented, for which feeding points of arithmetic units are relocated in order to reduce unnecessary registers (similar optimisation is implemented in [8]). A result of the optimisation is shown in Figure 3-2b. It should be noted that the total convolver pipelining delay is often reduced by the optimisation. This optimisation technique is implemented for every architecture described in this paper.

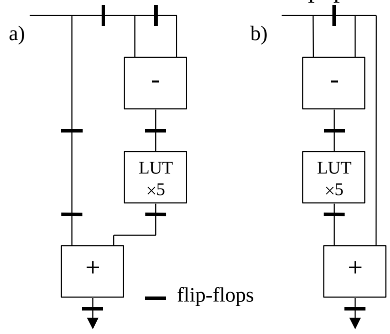


**Figure 3-2. Implementation of $(2 + 5\cdot z^{-1} - 5\cdot z^{-2})$ filter for pipelining parameter p= 1 and a) without b) with pipelining optimisation**

## 4. Distributed Arithmetic Convolver (DAC)

The idea behind the DAC [4, 5] is to compute the convolution in different order than for the LC. The following mathematical transformation has been employed:

$$\sum_{i=0}^{N-1} h_i \cdot a_i = \sum_{i=0}^{N-1} h_i \cdot \sum_{j=0}^{L-1} 2^j \cdot a_{i,j} = \sum_{j=0}^{L-1} 2^j \cdot \sum_{i=0}^{N-1} h_i \cdot a_{i,j} \quad (4\text{-}1)$$

*where: N- size of the convolution kernel, L- width of the input argument a (in bits),* $h_i$*- i-th coefficient of the convolution,* $a_{i,j}$*- -j-th bit of the i-th input argument.*

In comparison with the LC, the LUT data width of the DAC is smaller, as it can be seen from eq. 4-2.

$$W_{DAC}=K+\lceil log_2(N+1) \rceil \quad (4\text{-}2a)$$

$$W_{LC}= K+W_{IN} \quad (4\text{-}2b)$$

*where:* $W_{DAC}$ *- data width of LUTs for the DAC,* $W_{LC}$ *- data width of LUTs for the LC,* $W_{IN}$ *- width of the input of the LUTs, K- width of the coefficients of the convolution, N- the size of the convolution kernel.*

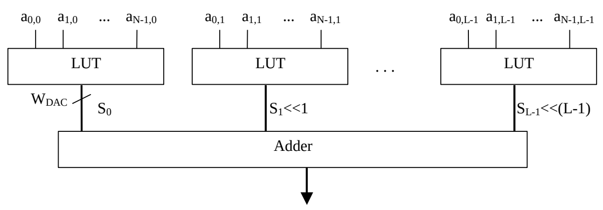


**Figure 4-1. Diagram of the Distributed Arithmetic Convolver**

The data width of the LUTs is a direct sum for the LC, and is a sum of the logarithm of the number of inputs to the LUT for the DAC. This is a consequence that input bits are at the same significance for the DAC. The lower output width of the LUTs causes substantial FPGAs area savings, because not only smaller memory modules but also shorter adders are required. As a result, the DAC is preferable to the LC.

A diagram of the DAC is shown in Figure 4-1. Similarly as for the LM, the size of the LUT memory grows rapidly with the size of the convolution kernel *N*. Therefore the LUT memory should be split into two or more independent LUTs, and then adders employed. The split of the memory should be implemented with respect to the cost-relation between different memory modules and adders, similarly like for the LM.

Consequently, in some cases the LUT based Hybrid Convolver (LHC) [14] - a hybrid solution of the LM and DAC, should be implemented in order to obtain optimum memory split. For example, for the 3×3 convolution, the number of multipliers equals $N=9=3\cdot3$, for coefficient width $K=8$ and input width $L=8$, two different memory modules should be used: four and five input memory blocks (4+5=9), but the 32×1 memory module occupies twice the area of the 16×1 module. Alternatively, the LHC employs the DAC for $N=8$ and a single LM. The cost for the pure DAC is 226 XC4000 CLBs and 209 CLBs for the LHC [14]. Therefore 17 CLBs are saved by the use of the LHC.

## 5. Irregular Distributed Arithmetic Convolver (IDAC)

The DAC architecture assumes that its structure is regular, i.e. the same LUT memory assignments for different significance of input bits. However, this need not be the case, and bits of different significance can be grouped together in the same LUT, in such a way that the total LUT data width is the lowest. Therefore more or less a combination of the LC and DAC is obtained. This novel, introduced by the authors of this paper, design approach is denoted as Irregular Distributed Arithmetic Convolver (IDAC). An IDAC optimisation algorithm should optimise rather the address and data widths of memories and adder widths, and the bit-significance of inputs is only an input parameter which influences the LUT data widths.

A greedy algorithm for the IDAC is proposed. This algorithm optimises a partial solution, i.e. determines the LUT address width and the LUT inputs, according to the algorithm given in Listing 5-1. The key issue of this algorithm is not only an optimal assignment of inputs to the memory but also selecting the optimal size of the memory – it should be noted that similarly like for the LM (see e.g. Figure 2-2) different memory modules can be used. Before the optimisation algorithm is applied, every coefficient is shifted to the left until it is made odd. This reduces the data width of the LUT as the LSB(s) of an even coefficient is fixed to zero. The input bit for which the coefficient is shifted is further treated as the input bit with significance increased by the number of shifts.

The algorithm given in Listing 5-1 employs the following priority queue: At first input bits with the lowest shift $s_i$ (step S1) are selected. Step S2 tends to allocate firstly inputs for which coefficient width is the lowest and this step is applied only to input bits at the lowest shift, i.e. for input bits selected at step S1. Step S3 optimises sign of the output, i.e. allocates at first input bits which representation (either positive or two's complement) corresponds with the representation of the LUT output. Step S3, however, is of the lowest importance and is considered only if two previous steps do not give the best solution.

**Listing 5-1. Algorithm choosing the best partial solution for the IDAC**

```
$c_{best}= \propto$ (Initial conditions)
$width= 1$
Start of the loop
        S1: Find an unassigned input bit with the lowest shift $s_i$
        S2: If two or more input bits are found with the lowest
                shift $s_i$, take the input with the lowest coefficient
                width $w_i$.
```

S3: If two or more input bits are found in step S2, take the one which output sign corresponds with the output sign of the LUT.
S4: Calculate average cost $c_a$ per input bit (consider also inputs found in the previous iterations of this loop)
S5: If $c_a < c_{best}$ then $c_a = c_{best}$ (the better circuit has been found)
S6: *width= width+1*
S7: If *width>max_width*
then finish the algorithm and return the circuit with the lowest cost $c_{best}$
else go to the start of the loop

where:
*width* – address width of the considered IDAC LUT
*max_width* – maximum address width for the considered memories (or the number of unassigned input-bits if smaller)
$s_i$ – shift of the input bit, $s_i$= significance of the input bit + shift of the coefficient (while making coefficient an odd value)
$w_i$ – width of the coefficient after the coefficient is shifted (made odd).
$c_a$ – average cost of the input bit,
$$c_a = \frac{Cost\_memory + Cost\_adder}{width}$$
*Cost_memory* – cost of the memory module which address width is equal or greater than *width* and data width is obtained from output range of the memory.
*Cost_adder* – adder cost which width is equal the width of the memory data + 1.
$c_{best}$ – the lowest average cost per input bit – this cost is associated with the best-found circuit.

Step S4 calculates the average cost (per input bit) of the memory module and the associated adder. In this step, an assumption is made that the width of the adders is equal the memory data width plus one. Actual width of the adder depends on routing the adders in the adder tree. Step S5 determines the best circuit, i.e. the circuit for which average cost per bit is the lowest. The next steps S6 and S7 are loop control instructions. An example of circuit obtained by this algorithm is given in Section 6.

The above novel algorithm deals with the problem which has not been considered. Probably better optimisation techniques can be developed and better optimisation criteria in the greedy algorithm selected. Furthermore, optimisation techniques, which focus on global optimisation, should be implemented, e.g. a Simulated Annealing.

## 6. Implementation Results

To compare implementation results of the DAC and IDAC, an example of a filter is given below:

$$H(z)= 59 + 183{\cdot}z^{-1} + 162{\cdot}z^{-2} - 7{\cdot}z^{-3} - 48{\cdot}z^{-4} + 12{\cdot}z^{-5} + 9{\cdot}z^{-6} + 2{\cdot}z^{-7}. \qquad (6\text{-}1)$$

The IDAC circuit for this filter is given in Figure 6-1.

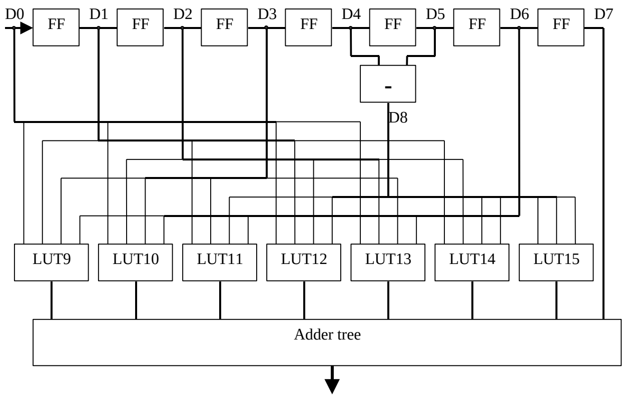


**Figure 6-1. Block diagram of the IDAC**

Before the algorithm given in Listing 5-1 is applied, similar coefficients are first grouped and addition/subtraction on grouped inputs implemented. Similar coefficients are coefficients which values are shifted and/or negated with respect to each other. For the given filter, coefficients: $-48{\cdot}z^{-4} + 12{\cdot}z^{-5}$ are similar, therefore associated inputs are shifted and subtracted from one another, and a single multiplication is applied (signal *D8* in Figure 6-1). Then coefficients are shifted until made odd. Consequently after shifting the coefficient 162, a coefficient 81 with shift equals 1 is obtained. The final coefficient values are shown in Table 6-1 in row 2 (coef).

According to the algorithm given in Listing 5-1, input *D6* (coef=9), then input *D3* (coef=-7), input *D0* (coeff= 59) and finally input *D1* (coeff= 183) are selected. According to the given cost of memories and adders, the lowest average cost per coefficient is obtained for 4-input RAM (16×1), therefore the next input bits are grouped to the next DA-LUTs. It should be noted that input *D7* for which coefficient is equal a power of 2 is fed directly to the adder tree.

| Input | D0 | D1 | D2 | D3 | D6 | D7 | D8 |
|---|---|---|---|---|---|---|---|
| Coef | 59 | 183 | 81 | -7 | 9 | 1 | 3 |
| Bit 0 | 9,2 | 9,3 | | 9,1 | 9,0 | | |
| Bit 1 | 10,2 | 11,0 | 10,3 | 10,1 | 10,0 | D | |
| Bit 2 | 12,0 | 12, 2 | 12,1 | 11,3 | 11,2 | I | 11,1 |
| Bit 3 | 13,2 | 14,0 | 13,3 | 13,1 | 13,0 | R | 12,3 |
| Bit 4 | | | 14,2 | | | E | 14,1 |
| Bit 5 | | | | | | C | 14,3 |
| Bit 6 | | | | | | T | 15,0 |
| Bit 7 | | | | | | | 15,1 |
| Bit 8 | | | | | | | 15,2 |

**Table 6-1. IDA-LUT assignment for different inputs bits (compare with Figure 6-1)**

The implementation results for the filter in eq. 6-1 are given in Table 6-2. CORE Generator [10] developed by the Xilinx Inc. generates the circuit employing (Parallel)

Distributed Arithmetic Convolver (DAC). The AuToCon generates the circuit employing the IDAC for which two different pipelining options are given in Table 6-2. For $p=\propto$, pipelining is not implemented; for $p= 1$, pipelining flip-flops are inserted after every logic cell. It can be seen from Table 6-2 that the IDAC significantly outperforms the DAC, which corresponds with the proposal of this paper.

The architecture of the IDAC is irregular in comparison the DAC, therefore it might seem that this would cause a rapid increase of the minimal clock period *T*. Nevertheless it is not the case. In some cases (e.g. in Table 6-1) the IDAC outperforms the DAC even when the minimal period is only considered. This can be explained by the fact that a lower area circuit can be better placed and routed in a FPGA.

| Circuit | # 16×1 LUTs | # FFs | T [ns] |
|---|---|---|---|
| 1) Core Generator | 169 | 205 | 9.2 |
| 2) AuToCon p=∝ | 125 | 28 | 28.4 |
| 3) AuToCon p=1 | 126 | 179 | 8.7 |

**Table 6-1. Implementation results for XC4005PC84-09 for different filter options**

## 7. Conclusions and suggestions for further work

This paper proposes a novel architectural solution for a convolver: the Irregular Distributed Arithmetic Convolver (IDAC). The IDAC improves the DAC by introducing irregularity, which makes possible the circuit optimisation. The implementation results show that the IDAC allows for significant area reduction in comparison to the DAC.

An IDAC optimisation algorithm has been proposed. This algorithm is a greedy algorithm therefore better priority queue might be found. Furthermore, a more sophisticated optimisation algorithm, such as Simulated Annealing (e.g. a similar algorithm as proposed in [11] for optimisation of the adder tree) should be proposed, which might reduce the IDAC area. Furthermore, this algorithm might consider address width reduction (the LAWR and DAWR, see the LM). For example, the IDAC LAWR is observed in LUT 11 for which only a single address line is required for bit 1.

In Section 2.1 a Multiplierless Multiplication is proposed for caring out multiplication. Implementation results presented in Section 2.3 shows that the MM outperforms the LM. Consequently a similar technique – Multiplierless Convolution might be implemented for the convolver. This is a suggestion for comparing different convolver architectures and selecting the better of them.

The IDAC architecture is incorporated in the Automated Tool generating 2D Convolvers in FPGAs (AuToCon) [15]. The AuToCon generates automatically a VHDL description of a convolver for different parameters such as kernel size and coefficient values, etc.

## References

[1] Wiatr K., Jamro E. *Implementation of Multipliers in FPGA Structures*, Proc. of the IEEE Intern. Symposium on Quality Electronic Design, San Jose, California, 26-28 March 2001, pp. 415-420, IEEE Computer Society Press.

[2] Omondi A.R *Computer Arithmetic Systems. Algorithms Architecture and Implementations,* Prentice Hall 1994

[3] Wiatr K., Jamro E. *Constant Coefficient Multiplication in FPGA Structures*, Proc. of the IEEE Int. Conf. Euromicro, Maastricht, The Netherlands, Sep. 5-7, 2000, Vol. I, pp. 252-259, IEEE Computer Society Press.

[4] Burrus C.S.: *Digital filter structure described by arithmetic*, IEEE Transaction on Circuits and Systems, pp. 674-680, 1977

[5] Do T.T., Reuter C., Pirsch P. *Alternative approaches implementing high-performance FIR filters on lookup table-based FPGAs: A comparison.* SPIE Conference on Configurable Computing and Applications, Boston, Massachusetts, pp. 248-254, 2-3 Nov. 1998.

[6] Garner H. *Number Systems and Arithmetic,* Advances in Computing, vol. 6, pp. 131-194, 1965

[7] Pirsch P., *Architectures for Digital Signal Processing,* Chichester UK, Wiley 1998.

[8] Hartley R.I. *Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers,* IEEE Transactions on Circuits and Systems II – Analog and Digital Signal Processing, vol. 43, no. 10, Oct. 1996.

[9] Xilinx Inc. *The Programmable Logic Data Book,* San Jose, California, 2000.

[10] Xilinx Inc. Core Generator, Foundation 2.1, 1999

[11] Jamro E., Wiatr K., *Genetic Programming in FPGA Implementation of Addition as a Part of the Convolution*, Proc. of the IEEE Int. Conf. Digital System Design, Warszawa, Poland, 4-6 Sep. 2001, pp. 466-473, IEEE Computer Society Press.

[12] Jamro E., Wiatr K. *Implementation of convolution operation on general purpose processors* Proceedings of the Euromicro Conf. Warszawa, Poland, 4-6 Sep. 2001, pp. 410-417, IEEE Computer Society Press.

[13] Lu W.-S., *Two-Dimensional Digital Filters,* Marcel Dekker, New York, 1992.

[14] Wiatr K., Jamro E., *Implementation of image data convolutions operations in FPGA reconfigurable structures for real-time vision systems.* International IEEE Conference on Information Technology: Coding and Computing, Nevada 2000, pp. 152-157.

[15] Jamro E. *Parameterised automated generation of convolvers implemented in FPGAs*, Ph.D. Thesis, AGH Technical University, Kraków, Poland, June 2001.