FPGA Implementation of Addition as a Part of the Convolution

Ernest Jamro, Kazimierz Wiatr

AGH Technical University, Institute of Electronics

Mickiewicza 30, 30-059 Kraków, Poland

tel. +48 12 6173033, fax +48 12 6332398, email: wiatr@uci.agh.edu.pl

Abstract

Addition is a fundamental operation for the convolution (FIR filters). In FPGAs, addition should be carried out in a standard way employing ripple-carry adders (rather than carry-save adders), which complicates search for an optimal adder structure as routing order has a substantial influence on the addition cost. Further, complex parameters of inputs to the adders tree have been considered, e.g. correlation between inputs. These parameters are specified in different ways for different convolver architectures: Multiplierless Multiplication, Look-Up Table based Multiplication, Distributed Arithmetic. Furthermore, optimisation techniques: Exhaustive Search and Greedy Algorithm have been implemented, and as a result, the Greedy Algorithm is the best solution when time of computation is of great importance. Otherwise, the Exhaustive Search should be employed for the number of the addition inputs $n \leq 8$. This paper is a part of the research on the AuToCon – Automated Tool for generating Convolution in FPGAs.

Topics: *image processors, processing arrays and FPGAs, reconfigurable structures*

1 Introduction

1.1 Implementation of convolution in FPGAs

An N tap convolution can be expressed by an arithmetic sum of products:

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

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

The multiplication in eq. 1 can be carried out employing three different techniques:

- Multiplierless multiplication (MM) [1] where multiplication employs only shifts and additions from the binary representation (BR) 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. To reduce the number of operation (non-zero symbols) required in the coefficient's two's complement representation, canonic signed digit (CSD) representation [2] should be employed. 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). Summing up, for the MM, the FIR filter (convolution) arithmetic can be carried out using only additions and substations.

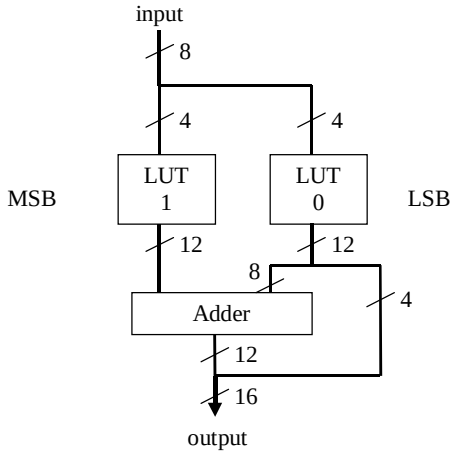

Figure 1. Look-up table based multiplication for 8-bit wide coefficient

- Look-up-table 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 [1, 3, 4, 5]. An example of this is given in Figure 1.

- Distributed Arithmetic (DA). The idea behind the DA [5, 6] is to compute the convolution in different order than for the LM. The following mathematical transformation is employed:

$$y(i) = \sum_{k=0}^{N-1} h(k) \cdot x(i-k) = \sum_{j=0}^{L-1} 2^j \cdot \sum_{k=0}^{N-1} h(k) \cdot x_j(i-k) \qquad (2)$$

where: L- width of the input argument x (in bits), x_j(i-k)- -j-th bit of the input argument at time (i-k).

Consequently, the DA carries out the convolution in a bit-plane order, i.e. every bit of inputs is considered separately. In comparison with the LM, the DA LUT output width is smaller because the inputs are at the same bit-significance. Therefore, the smaller memories are required which makes the DA more hardware-efficient than the LM.

For these three different techniques, the final and very often overlooked operation is addition (subtraction). For example, Thien-Toan Do et. al. [5] constructed the structure of the LM and DA and showed the final adders tree but the order of the additions seems to be intuitive rather than based on a thorough research. This draws a conclusion that general rules for constructing adders tree should be given and/or a design automation tool has to be developed in order to find an optimal order of additions.

1.2 Addition in FPGAs

For ASIC designs, the classic problem of carry propagation is resolved by numerous techniques, e.g. carry-look-ahead, carry-select [4], which reduce the delay of carry propagation at the expense of great increase in hardware complexity. Another approach to the carry propagation problem is to remove it completely through the usage of carry-save adders [4]. Consequently in ASICs, the usage of carry-save adders is a technique commonly implemented in the convolution (FIR filters) designs [7].

FPGAs incorporate a dedicated carry propagate circuit [8, 9] which is so fast and efficient that conventional speed-up methods are meaningless even at the 16-bit level, and of marginal benefit at the 32-bit level [8]. Furthermore the dedicated carry-propagate circuit is implemented outside the standard LUT logic and therefore does not occupy standard logic area. Consequently, using only ripple-carry adders in FPGA designs is the best solution with respect to the propagation time and occupied area.

As a result, there is a substantial difference between pipelining the ASIC and FPGA adders. For ASIC designs, pipelining flip-flops should be inserted every *N*-logic blocks (where *N* is an integer which value is application specific) therefore the carry-propagation chain is broken as it is shown in Figure 2. For FPGAs, the fast build-in carry logic significantly reduces carry-propagation time and therefore pipelining flip-flops should be rather inserted after every *K* additions (see Figure 2). Nevertheless, the build-in carry logic cannot nullify the carry propagation time, and therefore in the FPGA solution, the most time critical path is the carry-propagate circuit. For example, for Xilinx XC4000, delay through LUT logic, e.g. sum-generation circuit, is approximately six times longer than through the carry-propagate circuit. However, when the programmable interconnects delays are included, which essentially influence overall system performance, the carry propagate delay is much less significant. This holds as FPGAs incorporate dedicated and therefore very fast routing circuit form a carry-out to carry-in. Furthermore the propagation time through the programmable interconnects is usually comparable or even greater than the propagation time through LUT logic.

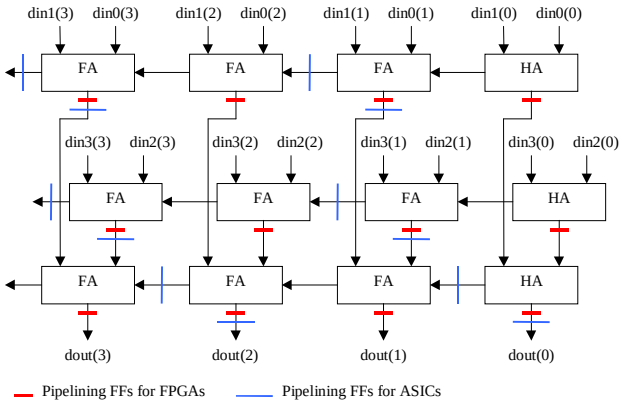

Figure 2. An example for different pipelining strategies for ASIC's and FPGA's additions for the equation: dout= din0 + din1 + din2 + din3. Pipelining parameters: N=2 (ASIC), K=1 (FPGA)

Nevertheless, in FPGAs, a long-width adder can be divided into several parts by inserting pipelining flip-flops every *M* carry-propagate blocks (like for VLSIs). This solution should be used together with the pipelining solution presented for FPGAs; i.e. a hybrid solution of the FPGA and ASIC designs (see Figure 2) should be employed. This, however, would complicate the system design and require additional flip-flops to be inserted according to the cut - set pipelining rule [10].

Therefore, this solution has not been implemented in the presented system, however, is considered in the next step of the design development. This hybrid solution seems to be better than delayed addition technique [11] for which a carry-in does not propagate to carry-out. Conversely, each 4-2 adder has the standard carry-out logic and 2 outputs and therefore 2 pipelining flip-flops are required for each 4-2 adder.

Summing up, for FPGAs the best solution seems to be using dedicated adders and pipelining after every *K* additions as it is shown in Figure 2.

1.3 Overview

In the next section, input parameters of the adder block will be specified. Initially it might seem that only input width is required, however, to achieve hardware savings the input range and even inputs correlation should be considered. Furthermore, the correlation between inputs depends on the FIR architecture: the MM, LM or DA, which makes implementation more difficult.

Further, different heuristics for finding an optimal adders tree are investigated. Implementation approaches and results are included to illustrate how the adders tree is optimised.

2 Addition parameters

2.1 Description of the input parameters

Section 1.1 describes methods of implementing FIR filters. Now, let consider the adders tree block alone, which is independent of these methods, and therefore let define input parameters to this block. The first intuitive parameter is the number of inputs and their bus widths or the minimum and maximum input values. Consequently, for the addition: $y = a + b$, the relation between the inputs and the output ranges is as follows:

$$y_{min} = a_{min} + b_{min} \qquad y_{max} = a_{max} + b_{max} \qquad (3)$$

It should be noted that for a subtraction $y = a-b$ the above equation also holds provided that the following substitution is carried out:

$$b_{min} = -b'_{max} \qquad b'_{max} = -b'_{min} \qquad (3a)$$

The use of minimum and maximum values instead of the bus widths can cause hardware savings, as some inputs might not use the full input range. For example, for input range from 0 to 9 adding three such inputs gives output range from 0 to 27, which requires 5 bit wide bus. If only the bus width is considered, the output width will be 6-bit wide. In addition, some inputs may have the LSBs fixed to zero as the argument is shifted to the left, therefore an additional shift parameter *s* should be also included.

Inputs Correlation

To further decrease additions width, correlation between inputs should be considered. Because an assumption is made that inputs *x(i-k)* (see eq. 1) are uncorrelated, the correlation occurs only within the multiplication $h(k)\cdot x(i-k)$ when the addition of the partial product takes place; e.g. addition of shifted *x(i-k)*. Consequently, the correlation is considered separately for each multiplication, and therefore in this section rather a multiplication than a whole FIR filter is considered.

The correlation should be considered for every intermediate addition; e.g. for the addition: $y = a+b+c$, at first auxiliary addition, $y_{ab} = a+b$, takes place and therefore only correlation between inputs *a* and *b* should be considered. Furthermore, the correlation should be calculated from the very beginning for every auxiliary addition; i.e. only inputs to the auxiliary adder block, which are involved into the considered partial addition, should be taken into account, and therefore the actual adders connection network within the auxiliary adder block is disregarded.

2.1.1 Multiplierless Multiplication

For the MM no correlation is observed unless a subtraction between the same argument takes place: $y = a - b$, where $a = 2^k b$. In this case, the eq. 3 should be replaced by:

$$y_{min} = a_{min} - b_{min} \qquad y_{max} = a_{max} - b_{max} \qquad (4)$$

2.1.2 LUT-based Multiplication

The correlation is more complicated in the case of the LM. Let $I_0, I_1, \ldots I_k$ be the inputs to LUT memories for a single multiplication, where I_0 represents the input to the LSBs LUT and I_k the input to the MSBs LUT; and $w_o, w_1, \ldots w_k$ represent the input width of the LUTs; $s_0, s_1, \ldots s_k$ represent the shift to the left of each LUT: $s_j = \sum_{l=0}^{j-1} w_l$, and *s* denotes the shift of the output. It can be seen that all LUTs but the MSBs LUT inputs operate on the positive binary range:

$$I_{j\,max} = 2^{wj}-1 \qquad I_{j\,min} = 0 \qquad \text{for } j = 0 \ldots k-1. \qquad (5)$$

The MSBs LUT is an exception for which the following equation holds:

$$I_{k\,max} = I_{max} >> s_k \qquad I_{k\,min} = I_{min} >> s_k \qquad (6)$$

where: I_{max}, I_{min} – maximum and minimum input values to the multiplier, >>s- denotes a shift to the right by s-bits.

The LUT output range can be defined as:

$$\begin{aligned} O_{j\,max} &= h \cdot I_{j\,max} & O_{j\,min} &= h \cdot I_{j\,min} & \text{for } h \geq 0 \\ O_{j\,max} &= h \cdot I_{j\,min} & O_{j\,min} &= h \cdot I_{j\,max} & \text{for } h < 0 \end{aligned} \quad (7)$$

where: h – the multiplication coefficient.

It should be noted that the total output range of the multiplication: O_{max}, O_{min} can also be obtained by employing the eq. 7 – only the index j disappears. The relation between the LUT output ranges is specified:

$$\begin{aligned} O_{\max} &\leq [\sum_{j=0}^{k} (O_{j\max} << s_j)] >> s \\ O_{\min} &\geq [\sum_{j=0}^{k} (O_{j\min} << s_j] >> s \end{aligned} \quad (8)$$

The above inequality becomes the equality if there is no correlation between outputs O_j or the correlation is not taken into account.

The algorithm of finding the correlated maximum and minimum of an auxiliary adder A (the set A contains indices of all inputs to the auxiliary adder block) is based on constructing a correlation set C ($C \subseteq A$). The set C contains the MSBs LUT k if the LUT k feeds the auxiliary adder A, i.e. $k \in A$, otherwise the set C is empty (no correlation is observed). The set C is further constructed in an iterative way, starting from the index $j = k-1$. The index j belongs to the correlation set C if the index $j+1$ also belongs to. Consequently, C contains successive elements: $j, j+1, ..., k-1, k$, where $j-1$ is the index of the MSB LUT which is not included in the auxiliary addition block i.e. $(j-1) \notin A$. The input range of the of set C is calculated in similar way as input range of the MSB LUT (eq. 6) and can be expressed as follows:

$$I_{Cmax} = I_{max} >> s_C \qquad I_{Cmin} = I_{min} >> s_C \quad (9)$$

where: $s_C = w - \sum_{j \in C} w_j = s_{\min(C)}$, *min(C) - the smallest index in the set C.*

The output range of the set C can be calculated in the following equation which is similar to eq. 7.

$$\begin{aligned} C_{A\,max} &= h \cdot I_{C\,max} & C_{A\,min} &= h \cdot I_{C\,min} & \text{for } h \geq 0 \\ C_{A\,max} &= h \cdot I_{C\,min} & C_{A\,min} &= h \cdot I_{C\,max} & \text{for } h < 0 \end{aligned} \quad (10)$$

Finally, the output range of the auxiliary adder A is calculated as follows:

$$\begin{aligned} O_{A\min} &= [(C_{A\min} << s_C) + \sum_{i \in A-C} (O_{i\min} << s_i)] >> s_A \\ O_{A\max} &= [(C_{A\max} << s_C) + \sum_{i \in A-C} (O_{i\max} << s_i)] >> s_A \end{aligned} \quad (11)$$

where: s_A - the shift of the auxiliary adder A; $s_A = min(s_i)$ for all $i \in A$.

It should be noted that the correlation set C is empty if the MSBs LUT is not included into the auxiliary addition block, i.e. $k \notin A$. In this case: $C_{A\min} = 0$, $C_{A\max} = 0$.

It is important to note that the correlation is not observed for the binary or two's complement full range of the input argument, e.g. for input range: *0 to 255* or *–128 to 127*.

Example 1

Let consider the example form Figure 1, for input range: *–99* to *99* (*8*-bit wide input) and coefficient $h=100$. Consequently, form eq. 5 and 6, the input range is $I_0 = 0$ *to 15* and $I_1 = -7$ *to 6*. Employing eq. 7 we obtain the output range: $O_{0min} = 0$, $O_{0max} = 1500$ and $O_{1min} = -700$, $O_{1max} = 600$. When the correlation is not taken into account, the output range of the addition is (from eq. 8) $O_A =$ *-11 200 to 11 100*. Otherwise (form eq. 11), $O_A =$ *-9 900 to 9 900*.

Hardware savings, after the correlation is taken into account, are more significant for less wide MSB LUTs. For example, for input range *–9 to 9* (5 bit wide input), $I_0 = 0$ *to 15* and $I_1 = -1$ *to 0*. Consequently, uncorrelated (from eq. 8) addition range is *-1600 to 1500*, in comparison to *–900 to 900* when the correlation is taken into account.

Correlation savings are even more efficient if the multiplication coefficient can be changed by employing the self-configurable multiplier [12] also denoted as the Dynamic Constant Coefficient Multiplier (DKCM) [13]; i.e. the LM for which RAMs instead of ROMs are employed in order to dynamically change coefficient values. In this case, instead of multiplication coefficient h, coefficient range h_{min} and h_{max} should be used. Consequently, eq. 7 should be replaced by:

$$\begin{aligned} O_{j\,max} &= Max\,(h_{max} \cdot I_{j\,max},\ h_{mim} \cdot I_{j\,min}) \\ O_{j\,min} &= Min\,(h_{max} \cdot I_{j\,min},\ h_{min} \cdot I_{j\,max}) \end{aligned} \quad (12)$$

It should be noted that for the DKCM, there is correlation between arguments even if full input binary range is used.

2.1.3 Distributed Arithmetic

For the DA, the correlation should be considered in a similar way as for the LM. However inputs to the LUT are from different filter inputs $x_j(i-k)$ (see eq. 1 and 2). Therefore each input to the LUT should have a separate correlation entry k_j, C_{jmin}, C_{jmax} (see Section 2.1.4) and should be regarded as an one-input LUT. This however complicates the input parameters set (one input has several correlation entities). Furthermore, the correlation is also much more difficult to be considered within the addition block, which significantly increases the calculation time.

Consequently, a simplified approach should be rather employed, for which the correlation is considered collectively for the set of inputs $x(i-k)$. Accordingly, k_j, C_{jmin}, C_{jmax} are calculated not for a single multiplication but for the sum of products. This, however, causes that if a single multiplication from the set does not comply to the correlation rules, i.e. an input miss occurs only for a single DA-LUT, the correlation of all inputs less significant than the missing input is disregarded. Therefore, the simplified approach may result in the hardware overheads. However, the case can be eliminated in most cases by sorting the correlation set – the more significant output of the DA LUT the larger index of the DA LUT.

2.1.4 Summary of the input parameters

In our approach for every input j, the following addition input parameters are specified:

- Input shift: s_j
- Input width: w_j
- Input range (e.g. for the LM, output of the LUTs) : Oj_{min}, Oj_{max}
- Kind of operation: *addition / subtraction*

Correlation parameters:

For MM:

- The multiplication input index: k. If two or more inputs have the same index k then for subtraction, the eq. 4 instead of eq. 3 should be employed.

For LM and DA:

- Correlation index : k – index of the MSBs LUT of: a) multiplication for the LM b) sum of products for the DA.
- Correlation range: C_{imin}, C_{imax}, these values are calculated in eq. 10 for the correlation set $C= \{j, j+1, ..., k\}$. If all inputs from j to k are involved in the auxiliary addition A then variables C_{Amin}, C_{Amax} rather then O_{jmin}, Oj_{max} and eq. 3 should be employed.

The correlation for the LM and DA is calculated correctly according to the above schedule if indices are correctly assigned. Correspondingly, for the LM, output indices are assigned incrementally and separately for every multiplication, according to increasing bit-significance of LUTs. Similar procedure is employed for the DA, however all inputs associated with the DA should be considered together.

2.2 Addition tree structure

Additional assumptions and rules for constructing the addition tree must also be specified. Consequently, a binary tree is employed for which the number of inputs to the next adder stage is halved. An example of the binary adder tree for 6 inputs is given in Figure 3. This assumption is rather intuitive and minimises the input-output delay. It should be however noted that for some cases when the delay time is disregarded, this assumption might exclude the best solution. This, however, is a rare case and therefore this assumption seems to be justified.

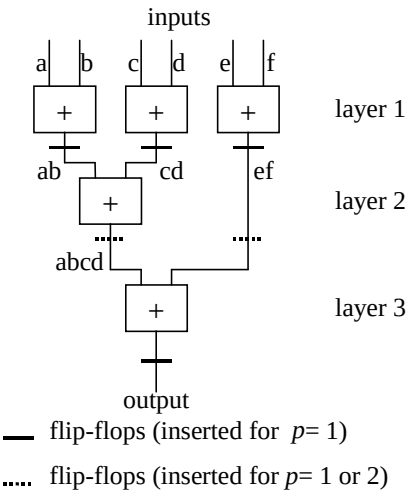

Figure 3. An example of the adders tree for 6 inputs

Consequently, the most complex part of the design is paring inputs to form two inputs adders. This task must be carried out with respect to the area of the adders structure. Therefore, the cost of a Full Adder (FA) cell should be another user-defined parameter. An alternative, universal solution is defining the cost of the adder for every possible adder width, e.g. from 1 to 32, as the average cost of a FA may depend on the adder width. Furthermore, the latter solution allows for area-time trade-offs, i.e. the cost of the adder increases rapidly with the increase of the adder width as the delay through the adder increases with the adder width. Consequently, the cost of the adders for different widths can be specified with respect to not only the occupied chip area but the delay time as well.

It should be noted that the actual width of adder is smaller than the width of the addition result in the case when one argument is shifted to another. An example of shifted arguments is given in Figure 1, for

which 4 LSBs are directly copied to the output. For subtraction, however, the LSBs of the subtrahend cannot be copied as it is the case for addition, because conversion to two's complement format has to be carried out on the subtrahend before the addition is implemented. Two's complement conversion involves negation of every bit of the subtrahend and adding *1* at the LSB position of the subtrahend. These operations can usually be carried out within a standard addition block in FPGAs, therefore no extra chip area is required. However, if the subtrahend is shifted to the right, the LSBs of the subtrahend have to be converted to two's complement representation even if the subtraction is not performed on these bits, and therefore additional logic and arithmetic is required in comparison to addition. Consequently, subtraction and addition have to be treated in a slightly different way [1] and different optimisation rules applied for additions and subtractions.

In order to speed-up the addition, pipelining is implemented. Consequently, additional user-defined parameter: level of pipelining p has been introduced. Parameter p defines that pipelining flip-flops are inserted after every p-layers of adders. An example of pipelining is given in Figure 3. For p= 1, flip-flops are inserted after every adders layer; for p= 2, flip-flops are inserted only after layer: 2, 4, 6, etc. It should be noted that flip-flops are incorporated in FPGAs after every logic element, therefore, it might seem that no additional chip area is occupied by the pipelining flip-flops. However, some bits of an addition result may be directly copied to the output. This happens when either inputs are shifted to each other (the case discussed in the previous paragraph) or an input cannot be paired (e.g. signal *ef* in Figure 3). In these cases no logic cell is required and consequently pipelining flip-flops may not be attached to any logic. Consequently, the design area may be specified by the number of flip-flops rather than the number of logic elements; and this causes the increase of the chip area. Summing up, the chip area is usually defined by the number of flip-flops for pipelining parameter p= 1 and by the number of logic elements for p≥2. However, for increasing p the design throughput is reduced, therefore a compromise between area and throughput is observed.

3 Greedy algorithm

Several techniques have been employed to optimise the adder block. One of them is the Greedy Algorithm (GrA) [14] which considers the estimated best partial solution. The drawback of this algorithm is that taking series of the best partial solutions often does not lead to the best overall solution, therefore an approximate solution is usually obtained. This algorithm is the quickest algorithm form all considered in this paper and very often gives an acceptable solution. The most significant part of the GrA, which strongly influences the overall result, is criteria which defines priorities according to which a partial solution is taken. In our project, different criteria have been specified for the first input and for the second input to an adder. The following rules has been selected:

1. First input

1.1. Take input with the smallest input shift s_1. If two or more inputs have the same input shift s_1, consider the next rule for these inputs.

1.2. Take input with the smallest input width w_1.

2. Second input

2.1. Take input with the smallest significance of the MSB $m_1 = s_1 + w_1$. Disregard this rule if the significance of the MSB of the first input $m_1 = s_1 + w_1$ is greater or equal than m_2 ($m_1 \geq m_2$). If two or more inputs have the same m_2 or $m_1 \geq m_2$, consider the next rule for these inputs.

2.2. Take input with the smallest shift s_2. If two or more inputs have the same shift s_2 consider the next rule.

2.3. Take input, which does not generate carry out signal of the adder (the input with the smallest addition result). If two or more inputs have the same smallest addition width consider the next rule for these inputs.

2.4. Take input with the greatest input maximum value $I_{2\,max}$.

Rule 1.1 causes that the first input is taken to sort inputs according to their shifts. Consequently, this rule considers the overall solution rather than the best partial solution as the unattached inputs tends to be of a greater shift and therefore easier to be grouped in the next iterations. Rule 1.2 tries to minimise partial solution by taking the smallest input width. This rule also supports finding a good overall solution as wider inputs can be easier grouped with inputs of a greater shift, which are left for the next iterations. The second input is taken rather to optimise at first partial and then overall solution. Rule 2.1 finds input which generates the smallest result width. Rule 2.2 finds input with the smallest shift and therefore tries to optimise overall solution similarly as rule 1.1. It should be noted that if input shifts are different ($s_1 < s_2$), (s_2-s_1)-bits are copied directly to the addition output (this copy does not require any hardware) and this justifies that rule 2.1 is more significant than rule 2.2. Rule 2.3 finds input which produces the smallest output. Furthermore, to improve the overall solution the input with the maximum value is chosen according to rule 2.4.

The above rules, although based on extensive research, are rather intuitive, therefore probable better

criteria may be found. Furthermore, the priority queue may be different for different input parameters; e.g. for subtraction, bit copy of shifted inputs cannot be implemented and therefore different rules may be specified. Furthermore, the average cost of a full adder (FA) may be different for different adder widths to allow area-time trade-offs, and this causes that a different priority queue should be specified, etc.

4 Exhaustive search

4.1 Concept

The best possible result can be always found by searching through all possible solution. The problem of finding the best solution for adders tree is NP-complete and therefore only simple adders blocks can be routed using the exhaustive search algorithm. At first, let consider an example of 5 input adder. The following solutions have to be examined (the bottom layer is only taken into consideration, the example shows how inputs (latters: *a* to *e*) are paired together):

(a+b)+(c+d)+e; (b+c)+(a+d)+e; (c+a)+(b+d)+e;
(b+c)+(d+e)+a; (c+d)+(b+e)+a; (d+b)+(c+e)+a;
(c+d)+(e+a)+b; (d+e)+(c+a)+b; (e+c)+(b+a)+b;
(d+e)+(a+b)+c; (e+a)+(d+b)+c; (a+d)+(e+b)+c;
(e+a)+(b+c)+d; (a+b)+(e+c)+d; (b+e)+(a+c)+d;

In order to specify the number of possible combinations, at first let define the function $S_1(n)$ which returns the number of all possible combinations within a single adder layer for a given number of inputs *n*:

$$S_1(n) = \begin{cases} n \cdot (n-2) \cdot (n-4) \cdot \ldots \cdot 3 \cdot 1 & for\ odd\ n \\ S_1(n-1) & for\ even\ n \end{cases} \quad (13)$$

The total number of possible solutions $S(n)$ is defined in an iterative way and is a product of the number of combination on this layer and the total number of combination for the upper (closer to the output) layers, i.e. for adders block for which number of inputs is halved:

$$S(n) = S_1(n) \cdot S(\lceil n/2 \rceil) \quad (14)$$

where $\lceil\ \rceil$ *- the ceiling function*

N	# layers	# combinations
2	1	1
3	2	3
4	2	3
5	3	45
6	3	45
8	3	315
10	4	42 525
12	4	467 775
14	4	42 567 525
16	4	638 512 875
18	5	1 465 387 048 125

Table 1. The number of possible combinations for a given number of inputs n to the adder block.

It can be seen from Table 1 that the number of possible solutions is growing rapidly, making the exhaustive search (ES) method useless for the input number greater then about 11-16.

4.2 Constrained Search (CS)

As the number of possible solutions is growing rapidly with the growing number of adder inputs, a modification of the exhaustive search (ES) method is here proposed. This method considers at first the cost of the GrA solution for every layer *l*. Consequently, the cost $C(l)$ of the partially routed adder (up to the adder layer *l*) is first calculated (initially using the GrA) for every layer *l* and then the similar to exhaustive search method is implemented. This method, however, stops calculating a group of solutions in its early stage (on layer *l*) if the cost of the partially routed adder is greater than $C_b(l) + t$; where: $C_b(l)$ – the cost $C(l)$ for the best overall solution so far found (initially found by the GrA), *t*- a certain threshold number. The procedure of comparison is executed after every layer of the adders tree is completed.

The CS technique saves the calculation time, as solutions which are less-likely to give the better solution are skipped on a low layer and therefore upper layers and their combinations are not calculated for the given partially routed adder. Conversely, it is possible then an adder block has a very high cost on the bottom layer(s), however the upper layers are much less costy, and therefore this adder block solution is skipped although it would give the best result. Consequently, the key problem is a proper choice of the threshold number *t*. Increase of the threshold number *t* increases the total number of considered solutions but decreases the probability of not finding the best solution.

N	ES	CS (layer 1)	CS (layers 1 and 2)
6	45	15	45
8	315	105	315
10	42 525	945	14 175
12	467 775	10 395	155 925
14	42 567 525	135 135	14 189 175
16	638 512 875	2 027 025	212 837 625
18	1 465 387 048 125	34 459 425	32 564 156 625

Table 2. Theoretical number of considered solutions for different number of adder inputs N

Table 2 shows the theoretical number of possible solutions for the CS assuming that the calculation process is constrained only to layer 1 and layer 1 and 2. Experiments proved that the number of iterations for the CS is in these ranges, and is greater for greater threshold *t*. It can been seen that the total number of considered solution has decreased significantly, however it is still unacceptable for the inputs number *N* greater than 18.

4.3 Implementation Results

In this section the results for the greedy algorithm (GrA), exhaustive search (ES) and constrained search (CS) algorithms are given. Table 3 shows the cost of the generated circuits by GrA, ES and CS (for different thresholds *t*).

Filter example	# inputs	ES	CS (t=5)	CS (t=2)	CS (t=0)	CS (t=-1)	GrA
a	16	93	93	93	93	93	111
b	11	72	72	72	73	73	74
c	13	123	126	126	126	126	128

Table 3. The implementation costs (number of full or half adders) for different filters and techniques

It can be seen from Table 3 that acceptable results are achieved using only the GrA. The improvement of about 2-7 % can be obtained by the use of the ES. The drawback of the ES is its computation cost therefore the reasonable solution seems the CS (for the number of inputs up to 16).

5 Conclusions

In this paper, thorough analysis of addition as a part of the FIR filters has been presented. Complex input parameters of the adder block have been considered: inputs range (not only input width), inputs shift and even the correlation between inputs. Consequently, finding an optimal network of the adders tree is a difficult task which has been investigated. Different approaches as: greedy algorithm, exhaustive search, constrained search have been implemented and the results given. Consequently GrA gives the worst solution but at very low calculation cost. Conversely, the best solution is obtained by checking all possible solutions in the ES, however the calculation time is unacceptable for the number of inputs, *n*, greater than about 12. Therefore, the Constrained Search (modification of the ES) has been proposed. The CS checks less solutions however the number of possible solutions increases rapidly with growing *n*, and therefore, this solution can be implemented for the number of inputs *n* less than about 14 – insignificant improvement in comparison to the ES. For small *n*, the SA usually finds the best solution and requires much lower number of iterations in comparison to the ES. However, for $n \leq 8$, the ES searches at most 315 solutions and therefore the computation cost is low. In conclusion, for $n \leq 8$ the ES solution should be implemented.

Addition block is a part of the convolution and therefore all procedures, described in this paper, are included in the Automated Tool for Convolution in FPGAs (AuToCon). The AuToCon [1, 13, 15] generates an optimised VHDL code of the convolver for the given coefficient values.

References

[1] Wiatr K., Jamro E. Constant Coefficient Multiplication in FPGA Structures, Proceedings of the 26th Euromicro Conference, Maastricht, The Netherlands, Sep. 5-7, 2000, Vol. I, pp. 252-259.

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

[3] Chapman K. *Fast Integer Multiplier fit in FPGA's*, EDN 1993 Design Idea Winner, END May 12th 1994.

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

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

[7] Hawley R.A., et, al. Design Techniques for Silicon Compiler Implementation of High-Speed FIR Digital Filters, IEEE Journal of Solid-State Circuits, vol. 31, no 5, pp. 656-667, May 1996

[8] Xilinx Co. *The Programmable Logic* Data Book 1999.

[9] Altera Co. *Apex 20K Programmable Logic Device Family, Data Sheet*, ver. 2.05, Nov. 1999.

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

[11] Luo Z., Martonosi M. *Using Delayed Addition Techniqus to Accelerate Integer and Floating-Point Calculation in Configurable Hardware*, SPIE Conference on Configurable Computing: Technology and Applications, Boston, Massachusetts, Nov. 1998, Vol. 3526, pp. 202-211.

[12] Wojko M., ElGindy H. *Configuration Sequencing with Self Configurable Multipliers* 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico, USA, April 1999, pp. 643-651.
[13] Wiatr K. Jamro E. *Implementation of Multipliers in FPGA Structures*, ISQED March 2001 San Jose, California
[14] Cormen T.H., Leiserson C.E., Rivest R.L. *Intoduction to Algorithms* Massachusetts Institute of Technology, 1994
[15] 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.