Constant Coefficient Multiplication in FPGA Structures

Kazimierz Wiatr, Ernest Jamro
AGH Technical University, Institute of Electronics, Mickiewicza 30, 30-059 Kraków, POLAND
email: wiatr@uci.agh.edu.pl, jamro@uci.agh.edu.pl

Abstract

This paper investigates different architectures implementing bit-parallel constant coefficient multiplication in FPGA structures. At first the multiplierless multiplication (MM) architectures employing Canonic Sign Digit (CSD) and sub-structure sharing methods are addressed, and a novel algorithm for the conversion from two's complement to CSD representation is presented. In the second part of this paper the Look up table based Multiplication (LM) is investigated. Correspondingly, the usage of different memory modules and finding the optimal combination of the memory and adders are considered. The LM architecture considers also reduction of the address width for each memory cell and the possibility of memory sub-structure sharing. Finally the implementation results for Xilinx XC4000 and Virtex families are presented. As a result, the MM generally surpasses the LM architecture, however the actual choice between these two architectures is coefficient and input parameters dependent.

1. Introduction

The multiplication is a very common operation in digital signal processing. Unfortunately, in some applications DSPs or general purpose processors cannot cope with the amount of data, which has to be processed and therefore Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Area (FPGA) solutions have to be adopted. The way a multiplication is carried out in ASIC or FPGA designs initially does not seem to be very different. Both ASICs and FPGAs require the same algorithms to be implemented. For example the structure of parallel-array multipliers [1] or Wallace tree multipliers [2] for FPGAs and ASICs seem to be very similar. Nevertheless, the most important advantage of FPGAs over ASICs is that a circuit implemented in a FPGA structure can be quickly reprogrammed. This allows for a change of the multiplication coefficient to be realised either by the change of the multiplicand (an input to the variable coefficient, fully functional multiplier) or by the change of the constant multiplier circuit. The constant coefficient in comparison to variable coefficient multiplier has much lower hardware requirements (26-33% of the fully functional multiplier [3, 4]), and therefore is recommended providing that a coefficient value is relatively constant during the calculation process [5].

FPGAs implement logic cells as a Look Up Table (LUT) memory therefore the inherent way of caring out multiplication seems the LUT based Multiplication (LM) for which large LUT memory is split and combined with adders [3, 4, 12]. Conversely, VLSI solutions usually implement constant coefficient multiplication as a Multiplierless Multiplication (MM) where multiplication is carried out with the employment of additions and subtractions [8]. Therefore, multiplier architecture for FPGAs and ASICs appears to be different. However, after dedicated ripple carry logic has been incorporated into FPGAs [9] an adder occupies half of the previous area and its speed has increased rapidly. This improvement has not been considered to re-establish the actual relation between the MM and LM architectures. In consequence the MM architecture has been overlooked. Summing up, the aim of this paper is to investigate the LM and MM architectures and find their cost/speed relation in FPGA structures.

In the first part of the paper, the Multiplierless Multiplication (MM) employing the Canonic Sign Digit (CSD) and Sub-structure Sharing (SS) methods, is investigated. As the consequence of the restrictions on the FPGA dedicated adders (or subtractors) structures, a modified algorithm for conversion from two's complement to CSD representation is derived, which allows substantial hardware savings. In the next part of the paper the LUT based Multiplication (LM) is studied. A FPGA incorporates different memory modules (e.g. for Virtex 16x1, 32x1, 256x16, etc.) together with the dedicated adder circuit. Therefore, finding the best architecture for a multiplier is a complex task that has to be addressed. In addition, some memory cells has a shorter address width and two or more memory cells may contain the same data, therefore a single (shared) memory can be implemented instead. It should be noted that every part of the research is followed by implementation results, which substantially helps to analyse aspects of the considered architectures.

2. Multiplierless multiplication (MM)

A constant coefficient multiplier is usually implemented in a multiplierless fashion by using 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. It should be noted that the hardware requirements depend on the choice of a coefficient - the number of 1's in the binary representation of the coefficient should be as low as possible. Therefore several algorithms have been developed in order to reduce hardware by a proper choice of the multiplication coefficient (e.g. for FIR filters design [6]). However, in this paper the assumption is that the value of the coefficient is an input parameter to a design and therefore the coefficient value cannot be changed.

2.1. Canonic Signed Digit 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 [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).

In general, the conversion of a two's complement number $B=b_{n-1}, b_{n-2}, ..., b_0$ to the CSD form $D= d_{n-1}, d_{n-2}, ..., d_0$ can be described formally [8] as in fig. 1.

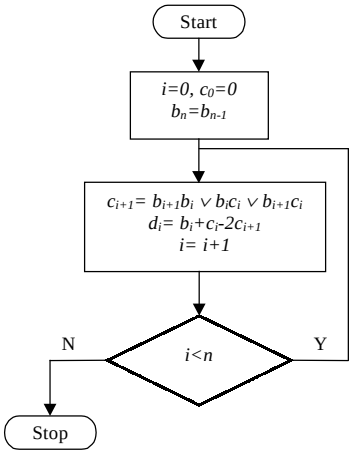

Fig. 1. The CSD conversion algorithm

The use of the CSD representation for each coefficient implies that the multiplication can be conducted in a shift and add (or subtract) fashion using the lowest number of add (subtract) operations. In the previous example: *B*= 14= 1110_2= $100\bar{1}0_{CSD}$, therefore *A* multiplied by *B* can be implemented as *(A<<4)-(A<<1)*. The CSD representation requires only one subtraction in comparison to the binary representation which requires two additions. One average, a CSD representation contains approximately 33% fewer non-zero bits than its binary counterpart [7]. This in turn implies hardware savings of about 33% per coefficient.

It should be mentioned that in the above CSD conversion algorithm a subtraction and addition are considered as the same cost operations. The addition *S= A+B* can be defined as:

$$s_i = a_i \text{ xor } b_i \text{ xor } c_i \quad (1)$$

$$\text{if } a_i = b_i \text{ then } c_i = a_i \text{ else } c_i = c_{i-1} \quad (2)$$

Similarly the subtraction *S= A – B* can be expressed as:

$$s_i = a_i \text{ xor not } b_i \text{ xor } c_i \quad (3)$$

$$\text{if } a_i = \text{not } b_i \text{ then } c_i = a_i \text{ else } c_i = c_{i-1} \quad (4)$$

For FPGAs (e.g. Xilinx XC4000), equations 1 and 3 are implemented in 4-input Look Up Tables (LUTs) which can implement any function of 4 arguments. Equations 2 and 4 are implemented in dedicated carry logic circuit [9], which allows the carry logic to propagate much quicker (propagation time for equations 2 and 4 is 5-50 times quicker that for eqs 1 and 3) and does not require additional area. It can also be seen from eqs 1-4, that the subtraction requires only the additional bit negation of the subtrahend, therefore the assumption of the equal cost of the addition and subtraction seems to be obtained. However in the case of the addition for which the first argument is shifted to the left, the least significant bits (LSB) of the second argument can be directly copied to the adder output, therefore additional hardware savings are achieved. Unfortunately for the subtraction, the subtrahend LSBs cannot be directly copied on the output because the subtrahend bits have to be inverted and a 1 forced into the carry input at the least significant bit position. Consequently, the addition and subtraction cannot be considered as the same cost operations. For example, for the multiplication by *B*= 3 = 11_2=$10\bar{1}_{CSD}$ for binary representation the LSB can be directly copied to the output but for CSD representation the copy of the LSB cannot be implemented. For ASIC designs these additional operations, which are carried out on the LSBs of the subtrahend, can be implemented as half adders which require much lower hardware in comparison with full adders. Unfortunately for FPGAs, the half adders usually occupies the same area as full adders.

2.2. Modified algorithm of the conversion to CSD representation

The standard algorithm of the conversion from the two's complement (TC) to the CSD representation does not consider the above conclusion, i.e. treats an addition and subtraction as the same cost operations. Therefore we have modified the conversion algorithm so that the conversion to the $\bar{1}$ symbol (negative one – the subtraction) takes place only if the total number of operations (non-zero symbols) decreases.

In order to describe the novel conversion algorithm a new function *Q(i,j)* for $j \geq i$ will be introduced. Let b_j is the *j-th* bit of two's complement representation of the multiplication coefficient *B* (*M= A·B*) and let define *Q(i,j)* in the iterative way as follows:

$$Q(i,i) = 0$$
$$Q(i,\ j+1) = \begin{cases} Q(i,\ j) + 1 & if\ b_{j+1} = 1 \\ Q(i,\ j) - 1 & if\ b_{j+1} = 0 \end{cases} \quad (5)$$

The function *Q(i,j+1)* is incremented if binary symbol b_{j+1} is *1* and decremented otherwise.

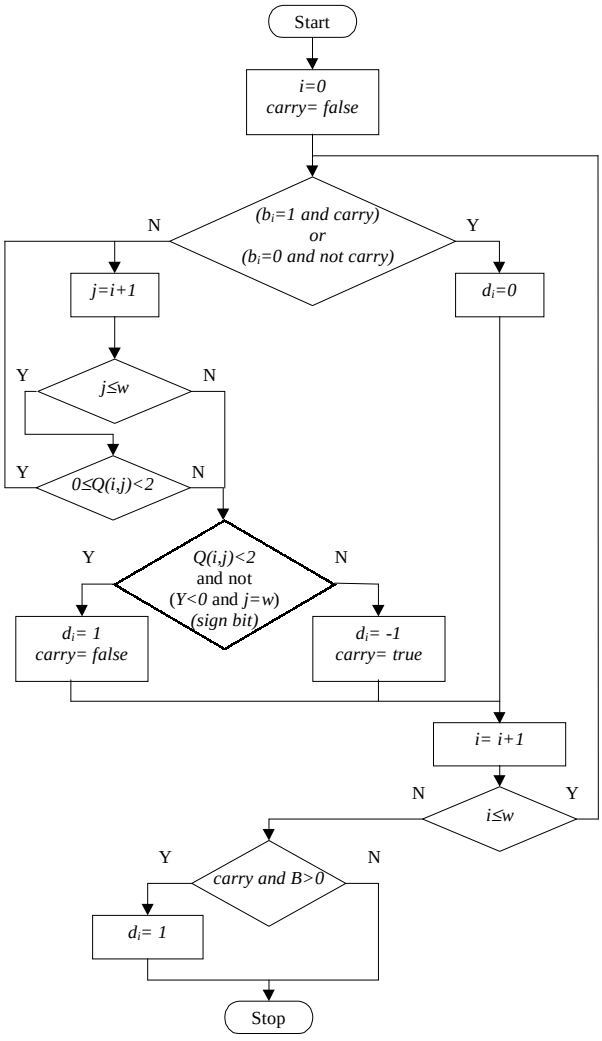

Fig. 2. The modified algorithm for the conversion form the two's complement to CSD representation; b_i- i-th bit of the binary coefficient, Q()- function defined in eq. 5, w- the index of the last bit of the multiplicand B

The modified algorithm is shown on fig. 2. It should be noticed that the conversion to CSD symbol $\bar{1}$ takes place only if the number of operations is reduced. This implies that the number of 1s in succession for TC representation should be at least three. If a 0-bit breaks the raw of 1s then the number of the successive 1s (skipping the 0-bit) should be increased by 2, or equivalently the counter of ones should be decreased by 1 (as it is the case for *Q(i,j)* function). The process of counting 1s (function *Q(i,j)*) should be stopped whenever count *Q(i,j)* is less than zero (consequently 0 or 1 symbol should be inserted) or is equal 2 ($\bar{1}$ symbol inserted).

An example of results for the standard and modified CSD conversion is given in tab. 1.

Coefficient	Binary (TC)	CSD	MCSD
3	11	$10\bar{1}$	11
7	111	$100\bar{1}$	$100\bar{1}$
11	1011	$10\bar{1}01$	1011
23	10111	$10\bar{1}00\bar{1}$	$1100\bar{1}$

Tab. 1. An example of results for the standard CSD and Modified CSD (MCSD) conversions

2.3. Sub-Structure sharing

Additional area reduction can also be achieved by a Sub-structure Sharing (SS) [10, 11]. For example, multiplication by $27=11011_2$ can be achieved by the use of an auxiliary variable *tmp* as it is shown in the following equations:

$$tmp = a + (a<<1) \quad (6)$$
$$27 \cdot a = tmp + (tmp<<3)$$

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 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. Conversely, the CSD may interfere with the SS therefore the SS should be considered separately on the two's complement and CSD representation.

2.4. Experimental results

The comparison of the area-reduction techniques presented in this section is a difficult task as the best algorithm depends on a coefficient value. However general conclusions, average and maximum circuit costs and the best algorithm occurrence can be derived.

The result for 8-bit unsigned input (the most common input data format for image processing) and coefficient width *K=3-12* is shown in tab. 2.

It can be seen from tab. 2 that the optimisation techniques (CSD, SS, CSD-SS) are more attractive for the large coefficient width. The average number of operations for the TC is roughly *K/2-1*, which for *K*= 12 gives 5 in comparison with 3.08 with the optimisation. However the cost of the multiplier increases almost linearly with the increase of a coefficient size *K* as it is shown on fig. 3. Consequently, the cost does not only depend on the

number of operations. Therefore the choice of the best technique cannot be taken only from the final number of operations (additions/subtractions) but the overall circuits cost has to be considered.

K	Avg.		Max.			Best algorithm occurrence			
	CLBs	L	CLBs	L	Coeff.	TC	CSD	SS	CSD-SS
3	2.71	0.57	5.5	1	7	6	1	0	0
4	4.13	0.87	9	2	11	12	3	0	0
5	5.52	1.16	10	2	23	22	9	0	0
6	6.92	1.44	14.5	3	43	39	17	4	0
7	8.44	1.75	15	3	75	68	43	16	0
8	9.8	2.03	17	3	183	114	94	44	3
9	11.1	2.31	20.5	4	309	188	193	107	15
10	12.3	2.57	23.5	4	747	300	407	254	62
11	13.6	2.83	24.5	4	1463	478	797	579	193
12	14.9	3.08	27.5	4	3381	746	1510	1285	554

Tab. 2. Average and maximum number of CLBs (XC4000) and corresponding number of operations (additions and subtractions) for the best technique. Input argument width L=8 bits, K- the maximum width of the coefficients (coefficient values [1, 2^K-1]). Avg.- Average requirements for the best algorithm (the algorithm that has the lowest cost chosen separately for each coefficient value), Max – the most hardware consuming coefficient. The best algorithm occurrence presents how many times each algorithm is the best. Considered algorithms: TC- two's complement representation; CSD- Canonic Sign Digit; SS- Sub-structure Sharing; CSD-SS – applying the SS algorithm on the CSD. If the results of two algorithms are the same then the simplest (most former) is taken

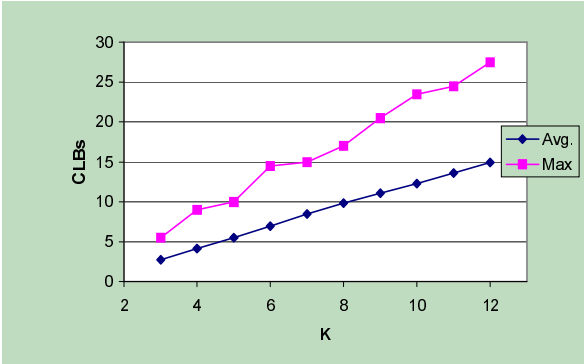

Fig. 3. The average and maximum area occupied by the 8-bit input multiplier for different widths of the coefficients K

3. LUT based Multiplication (LM)

3.1. Concept

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. This, in theory, gives the fastest possible implementation, since no actual arithmetic is required. 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. For example, for the *L*-bits wide argument and *K*-bits wide coefficient, the size of memory is $(L+K)\cdot 2^L$, which for *K=8, L= 8* gives 4k bits. It is, however, possible to create a practical implementation of the LM by combining a number of small LUTs and adders. The idea is to split the argument, use LUTs, and then use a tree of adders [3, 12, 13]. An example of the multiplier circuit for *K=8* and *L=8* is shown on fig. 4.

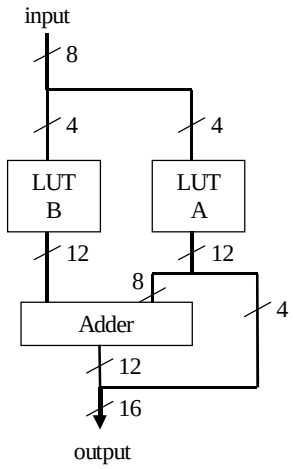

Fig. 4. The LM for input argument width K=8 and coefficient width L=8

The LUT contents for the multiplication $Y= A\cdot B$ can be evaluated directly form the multiplication as it is given in the example in tab. 3.

It should be noted 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 in tab. 3, the memory cell y_0 depends only on the address line a_0, memory cell y_1 depends on a_0 and a_1, y_2 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. This hardware saving will be denoted as LSBs Address Width Reduction (LAWR).

An additional decrease of the address width may be observed when the contents of the memory do not depend on a curtain address line. However this address width reduction cannot be generalised and differs for different coefficient values and address widths. Furthermore a complex search algorithm has to be employed to find a don't-care address line therefore this hardware optimisation will be denoted as Don't-care Address Width Reduction (DAWR). On the example given in tab. 3, the DAWR takes place for memory cells y_5 and y_4. It should be noted that the DAWR usually occurs for the MSB 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 can be implemented.

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

Tab. 3. 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

3.2. Implementation in FPGAs

The split of the multiplication argument should be carried out with respect to the size-cost relation of memory blocks and adders' cost. The XC4000 family incorporates 16×1 and 32×1-memory modules. The cost of the 32×1-memory module is 1 CLB, which is twice the cost of 16×1-memory module (½ CLB). The cost of the adder is ½ CLB/bit. In addition, there exists a virtual memory module 2×1 which does not occupy any CLB's area and can be implemented as either a connection from the input argument to the adder input or as feeding the adder with a fix value.

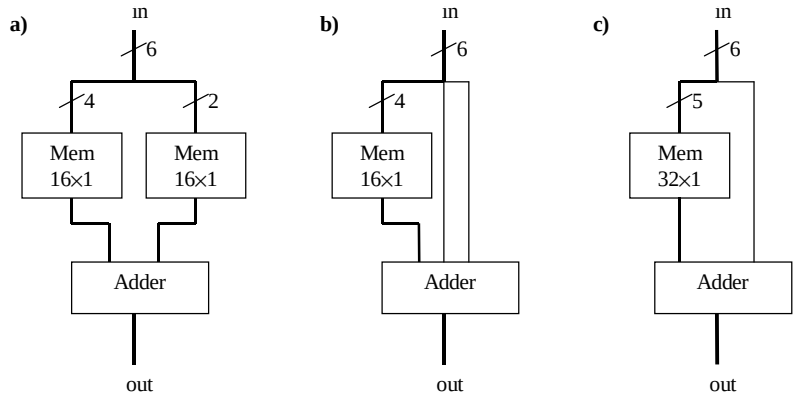

Fig. 5. Different reasonable methods for implementing the multiplier with the input bus width equals 6

Consequently, finding the optimum combination of the memory modules and adders is a difficult task, and the solution depends on the size of an input data and a given coefficient value. However, it can be derived by experiments that for the input width much greater than 4 the preferable memory blocks are 16×1. Unfortunately, if the input width cannot be divided by 4, different memory blocks may be used.

An example of different multiplier architectures for the input data width equals 6 is shown on fig. 5. The hardware requirements for these methods for coefficient equal 43 are: a) 11 ½ CLBs, b) 12 CLBs, c) 12 CLBs XC4000, therefore the difference is very slight. In general, however, there is a rule of thumb that the best or almost the best circuit is generated by the use of only 16×1 RAMs (and the direct connection to an adder if the remained input bus width is 1).

The design task is even more complicated if Virtex family is considered. Virtex FPGAs incorporate several large BlockSelectRAM+ (BSR) memories which are 4 kb in size and may have different data bus width: 4k×1, 2k×2, 1k×4, 512×8, 256×16 [9]. The area in silicon, occupied by

a BSR is equivalent to roughly 16 Virtex CLBs (64 16×1-RAMs). However the actual cost of these memories may differ respectably to the free FPGA resources, e.g. a design does not implement any BSRs but uses all CLBs. Consequently the trade-off for distributed RAMs and BSRs is design-dependent. However general conclusions can be derived from fig. 6. On average the equivalent cost of the BSR 256×16 is about 8÷11 Virtex CLBs (VCLBs) which is 32÷44 16×1 RAMs.

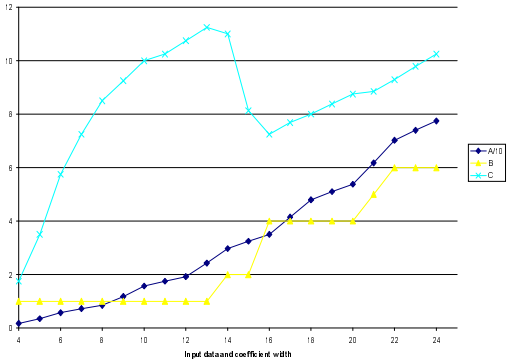

Fig. 6. The area of the LM for the same input data and coefficient width. A- area (in Virtex CLB= 2 XC4000 CLBs) scaled of 1:10, for the LM using only distributed 16×1 and 32×1 RAMs, B- number of used BSR 256×16, C- equivalent cost (in Virtex CLB) of a BSR 256×16 in comparison to distributed RAMs

It should be noted that the BSR blocks are rather large and therefore it is difficult to find an architecture for which the BSR is fully used. The efficiency of the BSR usage strongly influences its equivalent cost. Consequently, for small input data and coefficient width (<8) the equivalent cost of the BSR is rather small (see fig. 6). For the input width greater than 8 and up to 13 the equivalent cost of the BSRs increases. However for the width greater than 13, two or more BSRs and an additional adder are required, therefore the efficiency of the BSRs usage decreases as the effect of the large BSR quantization and distribution is again observed. For the input width equal 16 the number of BSRs increases rapidly as the BSRs are grouped into pairs to form a single 256×32 memory which implies low equivalent BSRs cost. As the width again increases the equivalent cost is growing. It seems, however that the equivalent cost equal 11 is a maximum value that is never surpassed.

Fig. 6A shows also the cost for multipliers using only distributed RAMs. It can be seen a rapid grow of cost for the width equal 9, 13, 17, 21, ... when the width surpasses the number divided by 4.

In this paper to find an optimal solution of a multiplier an exhausted search algorithm (with some obvious simplifications) has been implemented for which BSRs together with distributed RAMs and adders were combined and the best circuit taken. In order to visualise considered architectures an example of the LM for input data and coefficient width equal 14 is shown on fig. 7. In this example a combination of BSRs and distributed RAMs is implemented and the advantage hybrid solution is shown. The given example may be even more complicated if a concrete coefficient value is given, however in this section, general cases are investigated for which memory sharing (MS) and DAWR have not been considered.

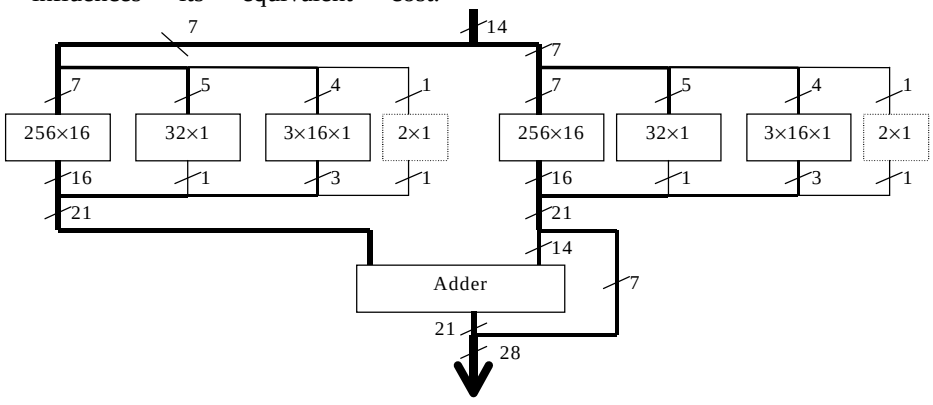

Fig. 7. The Lut Based Multiplier for input data and coefficient width equal 14

It should be noted that the optimisation program does not assume any initial relations between memory modules and adders' costs, therefore memory modules can be freely selected and the program can be implemented for any FPGA family or even for ASICs. The input values are an adder and memory parameters and costs. The output of the optimisation program is a VHDL file, which can be synthesised and implemented into any FPGA family.

In this paper only Xilinx XC4000 and Virtex families have been thoroughly studied, but almost the same properties have also different FPGAs. For example, Altera Apex 20K family [14] has almost the same cost-relation as Virtex has; the Apex family implements 16×1 LUTs or

dedicated carry logic in each Logic Element (LE) and also incorporates a large memory (128×16, 256×8, 512×4, 1024×2 or 2048×1) in each Embedded System Block (ESB). The size of Apex ESB RAMs is half the size of the Virtex BSR, however in most cases the Virtex BSR is not fully used therefore the main difference between Apex and Virtex seems to be the lack of 32×1 distributed RAMs for Apex family.

It should be also noted that the number of the BSRs depends on the cost relations. For example, for 16-bit width multiplier the number of BSRs can be gradually increases as the BSR cost decreases.

Cost BSR VCLBs	No. BSRs	Cost LUT RAMs VCLBs	Cost Adders VCLBs	Eq. Cost BSR VCLBs
≥7.75	0	19	16	-
≥6.5	2	9.5	10	7.75
≤6.25	4	0	6	7.25

Tab. 4. Influence of the BSRs cost for the best architecture for distributed (LUT) RAMs cost equal 16×1RAM= 0.25 VCLB and adder cost = 0.25 VCLB/bit. Relation for input data and coefficient width equal 16

While implementing LMs it can be seen that the Virtex BSR has not optimal parameters and often are not fully exploited. Memory modules 4k×1, 2k×2, 1k×4 and 512×8 have never been implemented, the memory module that has been only used is the 256×16 RAM. Even 256×16 memory modules are very seldom fully exploited. For example on fig. 7, one address line is not used, which causes that only half of the memory is used. Therefore a question arise what optimal memory size is. A general answer is that memory data width should satisfy:

$$W_D = W_C + W_A - W_L \quad (7)$$

where: W_D – memory data width, W_C – width of the coefficient, W_A- memory address width, W_L- address width of LUT RAM, W_L= 5 for Virtex.

From eq. 7 it can be seen that for W_C= *13* and W_A= *8* the result is W_D= *16* which correspond with the 256×16 memory module, therefore the maximum of equivalent BSR cost is observed in fig. 6 for the input width equal 13.

Additional hardware savings can be obtained if not full binary range of input data is used. For example, for the input data range 0-127 (binary range) and 0-99 (decimal range) and the coefficient equal 81 the implementation results are 14.5 and 13.5 XC4000 CLBs respectively.

Further design optimisation can be achieved for negative numbers. In general a design can be divided into 4 regions:

- Coefficient and input data are positive- there is not negative number optimisation.
- Coefficient is positive, input data is in two's complement format (negative or positive). In this case only the MSBs LUT operates on two's complement data. However the MSB LUT output can be either positive or negative therefore design optimisation cannot be implemented.
- Negative coefficient and positive data. All LUTs operate on two's complement numbers, but it can be seen that the outputs are always negative therefore additions can be replaced by subtractions and in this way all LUTs will operate only on positive data. In consequence, the LUT sign bit need not be coded - the width of each LUT will be one line shorter. However the double subtraction ($s=-a-b$) cannot be implemented into FPGAs and will be postponed to the next level of addition ($s= -(a+b)$), which implies that the result of the additions ($s= -a-b-c-d-...= -(a+b+c+d+...)$) should be negated, this however requires an additional circuit. Therefore the best solution is to implement substation for all but the LSB LUT. This implies that double subtraction chain is broken and the copy of LSBs (see Section 0) is achieved.
- Negative coefficient, two's complement data. In this case all but the MSBs LUT, operate on negative-only numbers and should be implemented as in the previous region. The MSBs LUT operates on either positive or negative numbers, therefore may be implemented as an addition. However this addition distracts subtraction-addition chain and causes that the LSBs copy does not occur. In conclusion the MSB LUT should be also implemented as a subtraction therefore the circuit is implemented in the same way as in the previous region.

4. Comparison of the multipliers

In this paper 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 on fig. 8. 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 fig. 8 shows only statistical relationship. Therefore both architectures should be

considered and the best of them chosen. However, form fig. 8 it can be seen that the gain from considering best of the LM and MM is insignificant for *K* greater than 5.

Form fig. 8 a general conclusion can be drawn. The MM implements more hardware reduction (using more efficiently the CSD and SS) with the increasing number of width *K*. Therefore for greater *K*, the MM is getting more and more attractive in comparision to the MM. The next aspect is how much hardware reduction is achieved by the use of the DAWR and MS for the LM. Experimental results show that the gain is on average 5÷20% depending on the input width *K*.

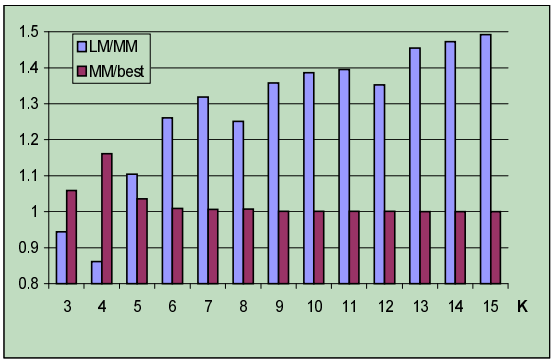

Fig. 8. Relation between average area of XC4000 occupied by: LM/MM – using only LM and only MM; MM/best – using only MM and the best of LM or MM. Results for the different input width K (input range $0 \div 2^K-1$) and coefficient values $1 \div 2^K-1$

5. Conclusions

This paper investigates two different methods of implementing multiplication: the LUT based multiplication (LM) and multiplierless multiplication (MM). The implementation results show that for a small input width, the LM is usually the best choice, but with the increase of the width the MM is getting more and more attractive due to greater effectiveness of the CSD and SS methods.

Furthermore, an improved algorithm for conversion from the two's complement to the CSD representation has been introduced. This algorithm considers that the cost of the subtraction is often higher than the cost of the addition as the copy of the LSBs cannot be implemented for the subtraction. Consequently, a subtraction (CSD representation $\overline{1}$) is implemented only if the total number of operation decreases. In addition, for the LM the combination of different memory modules has been studied. This aspect is very important as FPGAs incorporate different memories and therefore finding the optimal memory configuration is a complex task that has been solved. At the end, the cost/speed relationship between presented architectures has been presented for Xilinx XC4000.

In this paper the methods and implementation results for constant coefficient multiplier have been presented. But the main research effort, which has been passed over, was development of an multiplication automated tool (MAT) which can generated a synthesable VHDL code of any constant coefficient multiplier. The MAT does not assume any circuit constrains therefore can be used for any FPGA and even ASIC designs. The input parameters: an input data range, a coefficient value as well as an adder and flip-flop cost and memory module sizes and costs can be freely specified and the MAT will generate the optimal multiplier circuit. This allows that not only the generated circuit is hardware-efficient (better than commercial automated tool) but also the design time is significantly reduced.

References

[1] Waser S. *High-speed monolithic multipliers for real-time digital signal processing,* IEEE Computer Magazine, Vol. 11, No. 10, pp.19-29, 1978.

[2] Wallace C.S. *A suggestion for a fast multiplier,* IEEE Trans. On Electron. Comput., Vol. EC-13, pp. 14-17, 1964.

[3] Chapman K. *Constant Coefficient Multipliers for the XC4000E.* Xilinx App. Note, XAPP 054 December 1996.

[4] Petersen R., Hutchings B.L. *An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing,* In 5th International Workshop on Field Programmable Logic and Applications, Oxford England, pp. 293-302, August 1995.

[5] Wirthlin M.J., Hutchings B.L. *Improving Functional Density Through Run-Time Constant Propagation,* ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 86-92 (1997).

[6] Samueli H.,"An imroved search algorithm for the design of multiplierless FIR filters with power-of-two coefficients", *IEEE Transactions on Circuits and Systems,* Vol. 36, pp. 1044-1047, July 1989.

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

[8] Pirsch P., *Architectures for Digital Signal Processing,* Wiley 1998.

[9] Xilinx Co. *Using the Dedicated Carry Logic in XC4000E,* Xilinx Application Note XAPP 013 July 4, 1996.

[10] 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.

[11] Parker D.A. Parhi K.K. *Low-Area/Power Parallel FIR Digital Filter Implementations,* Journal of VLSI Signal Processings 17, 75-92, Kluwer 1997.

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

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

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

[15] Xilinx Co. *Core Generator* Foundation 2.1i Software Packet, 1999.