Parameterised automated generation of convolers implemented in FPGAs

Ernest Jamro

Supervisor

dr hab. inż. Kazimierz Wiatr, prof. n. AGH

A DISSERATION SUBMITED TO

UNIVERSITY OF MINING AND METALLURGY

DEPARTMENT OF ELECTRONICS

IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Kraków, Poland

June 2001

Abstract

Silicon technology now allows us to build chips consisting of tens of millions of transistors. As a result, more and more projects are constrained by the design time and complexity rather than available chip resources. This thesis describes a (C++ based) Automated Tool for generation 2-dimentional Convolers (2D FIR filters) implemented in FPGAs (AuToCon). The AuToCon can automatically generates a VHDL description of a wide range of convolers giving the list of parameters, such as: an input width, a convolution kernel size, coefficient values, a pipelining option, etc.

A novel synthesis approach has been proposed: the AuToCon does not assume any cost-relations between available memories, adders, multiplexers and flip-flops resources, these values are input parameters to the AuToCon. Even different memory types can be freely defined. Consequently, migration from one device family to another is rather effortless. Furthermore, within the same FPGA, cost-relations between different resources might differ and depend on the number of available resources (some resources might be already allocated by other designs incorporated into the same FPGA). Therefore, the AuToCon generates different circuits, i.e. allocates different resources, according to the cost-relations between the FPGA resources.

FPGAs, in comparison to ASICs, can be quickly reconfigured, therefore design functionality can be significantly improved by constant propagation through functional reconfiguration. In the course of this work, different architectures have been studied: constant coefficient architecture, where coefficient values are built-in the circuit. This architecture is the most hardware efficient, however any coefficient change requires the circuit to be redesigned. The second solution is variable coefficient option (usage of fully functional multipliers) which consumes much more chip-area, but coefficient can be changed without restrictions. There is also a mid-way solution for which coefficient is dynamically changed by employing in-circuit reconfiguration.

The AuToCon considers a wide range of possible architectures, employing sophisticated optimisation techniques such as exhaustive search, greedy algorithms, simulated annealing and genetic programming. These techniques have been employed e.g. to optimise the adders tree. As a result, the AuToCon does not only significantly reduces design time but also a generated circuit is, in most cases, more hardware efficient than a hand-crafted counterpart and comparable commercial solutions.

The greatest effort has been put into development of the AuToCon. Nevertheless this thesis presents a wide range of novel architectural solutions and algorithms, such as: a novel binary to Cannonic Sign Digit conversion algorithm, usage of different memory modules, implementation of dual port memories for Dynamic Constant Coefficient Multipliers and adaptive systems, extensive usage of Multiplierless Multiplication in FPGAs, advance optimisation techniques for LUT-based Multiplication, novel structure of Irregular Distributed Arithmetic Convoler, and the algorithm which trade-offs between multiplierless and LUT-based convolution.

In the course of this work, implementation of the convoler on different architectures, such as general-purpose processors, DSPs, dedicated VLSI convolers and FPGAs, has been presented. As a result, FPGA implementation usually outperforms the other solutions, and the developed synthesis tool significantly reduces design time and hardware requirements of a convoler. In conclusion, as convolution or similar operations (e.g. a sum-of-products) are fundamental operations in most digital signal processing systems, this work might be a crucial contribution in electronic digital designs.

Acknowledgement

I would like to thank professor Kazimierz Wiatr for his considerable support guidance and expertise throughout the duration of the project.

Contents

Glossary of terms

ALU – Arithmetic & Logic Unit

ASIC – Application Specific Integrated Circuit

AU – Address Unit

AuToCon – Automated Tool generating Convolers implemented in FPGAs

BR – Binary Representation

BSR – Block SelectRAM – large memory module (e.g. in Virtex 4kb DP RAM)

CCM – Configurable Computing Machine

CLB – Configurable Logic Block

CS – Constrained Search

CSD – Canonic Sign Digit

DAC – Distributed Arithmetic Convoler

DAWR – Don’t Care Address Width Reduction

DEA – Direct External Access

DKCM – Dynamic Constant Coefficient Multiplier

DKCM-D – DKCM with Dual Port RAM

DKCM-L – DKCM for which multiplexing is in logic (in CLB)

DKCM-P – DKCM for which two parallel set of RAMs are incorporated

DKCM-T – DKCM for which multiplexing is in tri-state buffers

DKLC – Dynamic Constant coefficients LUT based Convoler

DMA – Direct Memory Access

DP – Dual Port (Memory)

DSP – Digital Signal Processor

DU – Data Unit

ES – Exhaustive Search

FIR – Finite Impulse Response (Filter)

FA – Full Adder

FF – Flip-Flop (usually D-type)

FPGA – Field Programmable Gate Array

GA – Genetic Algorithm

GAU – Global Address Unit

GP – Genetic Programming

GrA – Greedy Algorithm

HA – Half Adder

HDL – Hardware Description Language

HLC – Hardware Loop Control

IDAC – Irregular Distributed Arithmetic Convoler

KCM – Constant Coefficient Multiplier

KLC – Constant coefficients LUT-based Convoler

LAF – Local Acceptance Function

LAU – Local Address Unit

LAWR – LSB Address Width Reduction

LC – LUT-based Convoler

LE – Logic Element – element which basically contains a single 4-input LUT (and an associated flip-flop), roughly 1LE = ½ CLB Xilinx XC4000 = ¼ CLB Virtex

LHC – LUT based Hybrid Convoler

LM – LUT based Multiplier

LSB – Least Significant Bit

LUT – Look-Up Table (Memory)

MAC – Multiply and ACcumulate

MC – Multiplierless Convoler

MCSD – Modified Canonic Sign Digit conversion algorithm

MM – Multiplierless Multiplier

MMX – MultiMedia eXtension

MP – Master Processor

MS – Memory Sharing

MSB – Most Significant Bit

PFCU – Program Flow Control Unit

RAM – Random Access Memory

ROM – Read Only Memory

SA – Simulated Annealing

SIMD – Single Instruction-stream Multiple Data-stream

SCO – Similar Coefficients Optimisation

SP – Single Port (Memory)

SS – Substructure Sharing

SSE – Streaming SIMD Extensions

TC – Transfer Controller

TOC – Trade Off Coefficient – a coefficient for which estimated cost of the MC is higher than for the IDAC

TSB – Tri-State Buffer

VC – Video Controller

VCM – Variable Coefficient Multiplier (fully functional multiplier)

VHDL – Very (High Speed Integrated Circuit) Hardware Description Language

VLIW – Very Long Instruction Word

VLSI – Very Large Scale Integration

Thesis:

Parameterised automated generation of convolers implemented in FPGAs allows for generation of convolers for different convolution parameters, in particular: input data range, convolution kernel size and coefficient values. Such an automated generation allows for effective FPGA utilisation by exploiting device-specific features and searching through different architectural solutions, allowing designer to achieve optimal performance and resource usage while minimising the need for knowledge of low-level details, and significantly reducing design time.

1. Introduction

1.1. Convolution Operation

It is hard to enumerate aspects of electrical engineering where filtering is employed. Examples of filtering operations include noise suppression, enhancement of selected frequency range, bandwidth limiting, etc. Analog filters suffer from sensitivity to noise, nonlinearities, dynamic range limitations, inaccuracies due to variations in component values, lack of flexibility and imperfect repeatability [Por97]. Consequently, digital filters (and FIR filters - convolers) are getting more and more attractive. The major drawback of digital filters is high computational requirements, especially for high frequency signals. Real time image processing is an example of such a system. This thesis concentrates on image convolution (two dimension FIR filtering), however similar conclusions can be drawn for 1D filters, matrix multiplication, or partially on artificial neural networks, etc.

This thesis briefly reviews computing machines which implement convolution, and as a result, Field Programmable Gate Arrays (FPGAs) seem to be one of the most suitable architectural solutions [DeH98, Bos99]. Different architectural solutions can be adopted in FPGAs, e.g. coefficient values can be constant [Wia00b], variable [Wal64], or dynamically changed [Woj98]. Consequently, a part of this thesis describes and compares different solutions.

A two-dimensional convolution (or a 2D FIR filter) is specified as follows:

$$b_{y+N/2,\ x+M/2} = \frac{1}{D}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1} h_{i,j} \cdot a_{y+i,x+j} \qquad (1\text{-}1)$$

where: N, M – size of the convolution kernel (usually odd numbers), $a_{y,x}$ *–input,* $b_{y,x}$ *–output,* $h_{i,j}$ *– coefficient of the convolution, D – common denominator.*

In this thesis, the assumption is made that all variables in eq. 1.1 are integers, which significantly simplifies the architecture of the convoler. This assumption seldom confines the filter characteristic which can be adjust by a proper change of coefficient values and a value of the common denominator *D*. Furthermore the denominator *D* is assumed to be a power of two, consequently the division is substituted by a bit-shift. In some solutions, even

coefficients $h_{i,j}$ are a power of two [Gon87, Tad92] and a multiplication can be substituted by an addition. Examples of such filters are given in Figure 1-1.

a) D=16

1	2	1
2	4	2
1	2	1

b) D=1

-1	-2	-1
0	0	0
1	2	1

c) D=1

1	1	1
1	-8	1
1	1	1

Figure 1-1. Examples of standard image processing convolution kernels which do not require multiplication: a) low-pass b) Sobel gradient c) Laplacean edge detection

Convolution is frequently a computationally demanding operation. For example, for image parameters: resolution 512×512 ($N_X= 512$, $N_Y= 512$), number of frames $N_F= 25/s$ and kernel size $N\times M= 3\times3$, real time image convolution [Wia98, Zam94, Zam95] requires $L_M= N_X\cdot N_Y\cdot N_F\cdot N\cdot M= 58\ 982\ 400$ multiplies and $L_A= N_X\cdot N_Y\cdot N_F\cdot(N\cdot M-1)= 52\ 428\ 800$ additions per second. Such the amount of operations is a challenge for nowadays' architectures. Furthermore, the parameters of the convolution can change; e.g. the size of the convolution kernel 3×3 is one of the smallest and often larger kernels are adopted, e.g. 63×63. The image resolution and the refresh rate might increase [Wia98], which significantly increases computational requirements of the convoler.

Due to the high-performance requirements of the considered systems, the bit-parallel approach has been adopted in this thesis. Nevertheless, there are bit-serial solutions as: serial distributed arithmetic [Pei99, Min92], employing multiply-accumulate unit [Cho93], or bit-serial FIR filters [Har90, Val98].

The given example of the real time image convolution is presented only to illustrate implementation problems. Nevertheless, a convolution operation (or a very similar operation: sum of products) is a fundamental operation which quick and efficient implementation is a crucial factor for variety of systems. For example, the developed system can be employed as a part of an artificial neural network [Meh97, Tad93], matrix multiplication, etc. In conclusion, there is great pressure to produce a faster and faster convoler to cope with new, more computationally demanding requirements.

1.2. Design Automated Tools

Silicon technology now allows us to build chips consisting of tens of millions of transistors. This technology promises new levels of system integration onto a single chip, but also presents significant challenges to the chip designer [Kea98]. As a result, many ASIC developers and silicon vendors are re-examining their design methodologies, searching for ways to make effective use of the huge numbers of gates now available. These designers see

current design tools and methodologies as inadequate for developing million-gate from scratch, and new design strategies are under development such as high level synthesis [Ele98], hardware/software codesign [Sta97], design reuse [Kea98], and core-based design [Rox00].

A core can be soft, firm or hard [Gup97]. A soft core consists of a synthesizable HDL (Hardware Description Language) description that can be retargeted to different semiconductor processes. A firm core is basically defined on gate level netlist that is ready for placing and routing. A hard core includes layout and technology depending timing information. Also a novel class of parameterised cores which produce a wide range of soft (or in some cases firm) cores should be introduced. A parameterised core can automatically generate a wide range of implementations [Luk96, Jam97, Jam99]. The parameterised core solution is especially adequate for convolers for which a wide range of input parameters is well defined. For reconfigurable computing such a core can be a part of a reconfigurable core [Rox00]. Reconfigurable cores consist of a configuration data stream plus software to modify the configuration data stream based on customisation required at run-time.

The Automated Tool for generation 2D convolers implemented in FPGAs (AuToCon) is an example of a parameterised core. The AuToCon incorporates a C++ written program, some VHDL-like templates [Jam97] and predefined VHDL files. More detailed description of the system is included in Appendix A, and a similar design approach has been described in [Jam97, Jam99, Jam00]. The hybrid solution of C and VHDL has been introduced independently and almost at the same time by the author of this thesis [Jam97] and Bramer et. al. [Bra97]; and proved to be an efficient way for generating parameterised libraries. The major advantage of such a system is that parametric specification of structural VHDL is achieved by the use of the C++ written program. This is an important factor as high-level synthesis tools are still under development [Ele98], and a high-level synthesised circuit is usually less hardware-efficient than a corresponding lower-level counterpart. Furthermore, the generated VHDL files are additional intermediate products which are not parameterised and therefore easier to be analysed, which allows for better testing and understanding the generated circuit. The generated VHDL files also significantly improve detecting design errors, especially when developing the hybrid system. More detailed argument for the hybrid solution is included in [Jam99, Jam00].

The AuToCon not only speeds up development time, but also the generated circuit outperforms hand-crafted one or generated by comparable automated tools. This is satisfied as novel architectural solutions have been introduced and advance search through different solutions gives superiors results.

1.3. Overview of the thesis

The structure of this thesis is as follows. Chapter 2 describes different architectural solutions which can implement convolution operation. General-purpose processors, DSPs, dedicated VLSI and FPGA solutions are overviewed and compared. In conclusion, according to the author strong believe, FPGA solution is most promising, and therefore there will be a strong demand for an automated tool that will generate a FPGA-based description of convolers.

In most cases convolution coefficient values are fixed, and multiplication is a basic operation for convolution. Chapter 3 approaches a Constant Coefficient Multiplier (KCM) and different architectures performing the KCM, such as Multiplierless Multiplication (MM) and LUT-based Multiplication (LM). Implementation results for these techniques together with some architectural modification are included.

Chapter 4 studies multiplier architectures for which coefficient change is a feasible factor influencing circuit design. Therefore, the KCM and fully functional multiplier, and a middle-way solution - Dynamic Constant Coefficient Multiplier (DKCM) are investigated and optimal multiplier architectures for different reconfiguration options studied.

Multiplication is basically a fundamental operation for convolution. However, there are architectural solutions, e.g. the Distributed Arithmetic, for which multipliers are not incorporated in the convolution. Alternatively, several multipliers combine with each other to form a more optimised circuit. Chapter 5 describes not only these architectures but also studies sophisticated algorithms which search for optimal solutions.

Addition is extensively employed during convolution, e.g. for multiplierless multiplication or convolution, addition is the only arithmetic operation employed. Therefore adders structure substantially influences the circuit area. Chapter 6 studies different algorithms which optimise adders tree, including exhaustive search, greedy algorithm, simulated annealing and genetic programming.

2. Different machines implementing convolution

This chapter briefly describes possible architectures performing convolution operation, especially 2D image convolution. At first, general-purpose processor and DSP solutions are approached. Then an example of dedicated VLSI solution is studied, and finally FPGAs are introduced. The major intention of this chapter is to present alternative solutions to FPGAs. These solutions, according to the author believe, are more or less saturated, i.e. great increase of the hardware complexity results in the insignificant computational speed-up. Conversely, FPGAs have been rapidly developed recently (much quicker than the other architectures) and therefore FPGA solutions are getting more and more attractive for implementation of convolution (and other digital signal processes).

2.1. General purpose processors

Most commonly used general-purpose processor is a family of 8086 processors [And95, Bre97]. These processors have complex architectures that are not optimal for convolution, however, they are commonly used, and therefore can be quickly and easily adopted as a convolution processor.

To illustrate the 2D convolution process on a general-purpose processor, an example of C-language procedure is given in Listing 2-1.

Listing 2-1 can be further optimised by the following procedures:

1. the loops unrolling
2. rewriting the convolution procedure in the assembler language
3. rewriting the assembler language procedure with respect to the superscalar architecture of the Pentium processor
4. employing MMX processor and its Single Instruction–stream Multiple Data-stream (SIMD) [Fly66] architecture

Listing 2-1. C-language procedure for the convolution

```
const int M= 3, N= 3; //  the size of the convolution kernel: M- horizontal; N- vertical
const int Nx= 512, Ny= 512; // image resolution: Nx- horizontal, Ny- vertical
const int D= 16; // the common denominator (see eq. 1.1)
BYTE a[Ny+1+N][Nx]; // the source image which has been enlarged to eliminate padding effect. The actual
        image is from line 1+N/2 to Ny+N/2. The rest of image a is specified to minimise the padding effect.
BYTE b[Ny][Nx]; // the destination image
int w[N][M]={1,2,1, 2,4,2, 1,2,1}; // coefficients of the convolution given in Figure 1-1a

void
convol2()
{   BYTE *pa= &a[0][Nx-M/2]; // the pointer to the source pixels (points top-left pixel of the conv. window)
    BYTE *pb= &b[0][0];  // the pointer to the destination pixel
    for(int y= 0; y<Ny; y++)  // for every line of the image
    {   for(int x=0; x<Nx;  x++, pb++, pa++) // for every pixel in the line
        { register BYTE *pw=w[0]; // the pointer to the coefficients.
          register BYTE *pa1=pa; // pointer to the current source pixel
          register int sum= D/2; // accumulation result – initially D/2 to minimise division rounding error
          for(int i= 0; i<N; i++)  // vertical convolution
         {  for(int j=0; j<M; j++) // horizontal convolution
                sum+= *pw++ * *pa1++; // the kernel of the convolution
            pa1+= Nx-M; // pa1 will point the first pixel in the next line
         }
         sum/= D; // division by the common denominator (D is a power of two so it is substituted by a bit-shit)
         *pb= (BYTE) sum; // conversion from int (4 bytes) to 1 byte variable, save the result.
        }
    }
}
```

2.1.1. Loop unrolling

In most cases, the size of the convolution kernel is fixed and therefore the convolution loops (loops: *i* and *j* in Listing 2-1– horizontal and vertical part of the convolution) can be easily unrolled by writing down $N \times M$ times the multiplication and addition operations. The loop instruction contains two assembler instructions:

- *dec*: decrement the loop counter
- *jnz*: conditional jump.

The loop unrolling causes not only fewer instructions to be executed but also fewer processor stalls occur. The stalls are caused by the conditional jump instructions which interfere with pipeline architecture of the processor [Mad95, Hwa93]. The pipelining causes that every instruction is executed partially in subsequent clock cycles. For example, Pentium 75 executes an instruction in 5 stages [And95]:

1. fetching
2. decoding (stage 1)
3. decoding (stage 2)
4. execution
5. updating registers

Instructions following a branch shall not be executed unless the branch is not taken. As a result of pipelining, the branch is finally executed in the last stage, however the branch-following instructions (which depend on whether the branch is taken or not) should be also partially executed according to the pipeline scheme. Consequently in 80486 and older processors, the instruction following a branch is not executed until the branch has been finally executed. This caused the processor stalls. To improve Pentium processors performance, branch prediction together with a Branch Prediction Buffer (BPB) [And95, Int97a] have been introduced. Consequently, the processors can efficiently execute instructions following branches to keep the instruction pipeline full [Int97b]. The drawback of the branch prediction is that branches are predicted incorrectly with a certain probability, $p>0$. Each misprediction causes a restart of the pipeline, which has similar effects as not fetching the instructions until the branch is finally executed.

To decrease misprediction ratio, Pentium 75 uses a two-bit up-down counter with saturation to keep track of the direction a branch is more likely to take [Smt81, And95]. Taking into account the convolution process (Listing 2-1) and the above branch prediction procedure, an assumption can be made that the processor will predict the loop to be executed infinitely [Wia00a], therefore every loop-braking causes the processor stalls.

The penalty for misprediction is even greater for the latest processors, as the number of pipeline stages has increased, e.g. Pentium Pro has 12 stages [Int97b]. This is confirmed by implementation results presented in Table 2-1 where $t_a/t_b= 1.42$ for P75, and 2.22 for Athlon 800MHz; where: t_a - calculation time without loop unrolling, t_b - with loop unrolling. The 2-bit up-down counter prediction scheme is rather primitive, and nowadays more sophisticated branch prediction procedures have been demonstrated, e.g. by Evers et. al. [Eve98] where branch prediction scheme can detect loops and an additional loop counter is included.

It should be noted that loop unrolling not only decreases the number of instructions to be executed, eliminates branches and branch misprediction effects but also improves instruction level parallelism, which will be approached in the next section.

2.1.2. Superscalar architecture

In a superscalar processor [Hwa93] multiple instructions pipelines are used. This implies that multiple instructions are issued per cycle and multiple results are generated per cycle. Superscalar processors are designed to exploit more instruction level parallelism in a user program.

In Pentium 75 (P75), the superscalar architecture was introduced [And95] which incorporates two parallel integer processing units:

- Integer unit *U*
- Integer unit *V*

The number of parallel units has increased in the latest processors. For example, Pentium Pro incorporates three-way superscalar architecture [Int97b], Pentium 4 incorporates Rapid Execution Engine [Int00] for which the ALUs run two times the frequency of the processor core.

Taking into account the P75, two integer instructions can be executed in a single clock cycle. However, some instructions cannot be executed in parallel, e.g. 'V' unit cannot execute shift instructions or two multiplication cannot be executed in parallel [And95, Int97a]. Besides, instruction-level parallelism is deteriorated by register contention, when e.g. the result of a 'U' instruction which is currently executed, is input to a 'V' instruction.

In conclusion, all units of a superscalar processor are not fully exploited. Only independent instructions can be executed in parallel without causing a wait state and therefore a superscalar processor depends strongly on an optimising compiler to exploit parallelism.

As a result of the above conclusions, Listing 2-2b presents an assembler code which better exploits the superscalar architecture of the P75 [Wia99a, Wia99b, Wia00a]. Nevertheless, optimised code (Listing 2-2b) is executed only 10% quicker (see Table 2-1 for the P75) than non-optimised code (Listing 2-2a). This tiny improvement can be explained as follows. The multiplication is a complex instruction which requires several clock cycles to be executed and cannot be carried out in parallel. Besides non-optimised code can also exploit the superscalar architecture of the processor. For convolution filters without multiplication, the optimised code is executed 25% quicker.

To further exploit the superscalar architecture, speculative execution has been introduced in the latest processors, which allows for out-of-order execution – a code is internally optimised for a superscalar architecture of the particular processor. Consequently, a programmer need not optimise a code every time the number of parallel units increases. Nevertheless, for speculative execution and three parallel units, the calculation time is shorter (up to 8%, see Table 2-1) for the two-way optimised code than for the non-optimised code. This is because, probably, the speculative execution algorithm is not optimal, and the optimised code is easier to be executed out-of-order. Besides the number of instructions in the loop has been reduced from 37 to 35 for the optimised code.

Listing 2-2. A fragment of a 3×3 convolution assembler code for a) scalar, b) superscalar architecture

a)

```
// pixel 3 (top-right pixel in the convolution window)
xor edx,edx // clear edx
mov dl, byte ptr [ecx+2] // load data a (pixel 3)
imul edx,dword ptr [edi+8h]//multiply: pixel3 * w[0][2]
add eax, edx  // accumulate the result of the multiplication
// pixel 4 (left middle pixel)
xor edx,edx // clear edx
mov dl, byte ptr [ecx+200h] // load pixel 4
imul edx, dword ptr [edi+0Ch] // edx= pixel4 * w[1][0]
add eax, edx // accumulate the result of the multiplication
```

b)

```
xor edx, edx //start of calculation for pixel 3: clear
imul ebx, dword ptr [edi+4] //pixel2: ebx=pel2*w[0][1]
mov dl, byte ptr [ecx+2] // pixel 3: dl=pel3
add eax, ebx // end of calculation for pixel2: eax+= ebx
imul edx, dword ptr [edi+8]  // pixel 3: edx=pel3*w[0][2]
xor ebx, ebx // start calculation for pixel 4: clear
add eax, edx // end of calculation for pixel 3: eax+= edx
mov bl, byte ptr [ecx+200h] // pixel 4: bl= pel4
```

It has been observed that the average value of instructions executed in parallel is around 2 for code without loop unrolling [Hwa93]. Even with loop unrolling, instruction-issue degree in a superscalar processor has been limited to 2 to 5 in practice [Hwa93, Tul95]. Let consider the case of the Pentium processors, it can be seen from Table 2-1 and Table 2-2 that average number of instructions executed by the P300 and Athlon 800 in a single clock cycle is up to the 2.3; for P166 it is about 1.17 (option without multiplication). It should be however noted that the improvement for the P300 and Athlon 800 has been also achieved by reducing the number of clock cycles required to perform multiplication. In conclusion, a superscalar architecture quickly gets saturated, i.e. increasing the number of parallel units requires much greater hardware expense but results in insignificant improvements.

An alternative solution is simultaneous multithreading, a technique permitting several independent threads to issue instructions to a superscalar's multiple functional units in a single cycle [Tul95]. This techniques allows for better utilisation of superscalar units as different threads can issue their instructions in such a way that register contention, memory miss or conflict and even branch misprediction penalty is significantly reduced. For example, while one thread waits for data transfer from external memory, others can issue their instructions to keep all processing units busy.

2.1.3. Very Long Instruction Word (VLIW)

The superscalar architecture requires complex instruction decoding, dispatching and speculative execution units. An alternative solution is a VLIW [Hwa93] architecture for which different fields of a long instruction word correspond to different functional units and therefore decoding and dispatching instructions is much easier. Unfortunately, a VLIW code has to be recompiled for a specified VLIW machine. Furthermore, for a superscalar processor, code density is greater as the fixed VLIW format includes bits for non-executable operations, while the superscalar processor issues only executable instructions.

The VLIW architecture is seldom employed in general purpose processors as tasks of the processors are unspecified and therefore it is rather difficult to optimise functions of the processors units. On the contrary, the convolution operation is well defined and requires basically four (six - if processor does not support addressing with offset) different operations:

- load the coefficient (and increment the coefficient pointer),
- load the input pixel (and increment the input pixel pointer),
- multiply,
- accumulate.

Consequently, these four (six) operations can be executed as a single VLIW instruction in DSPs which are optimised for digital signal processing as it will be described in Section 2.2.

Crusoe processor by Transmeta adopted a novel solution [Kla00]. This processor has a VLIW architecture which is able to perform 3 integer and one floating point operations in parallel (see Figure 2-1).

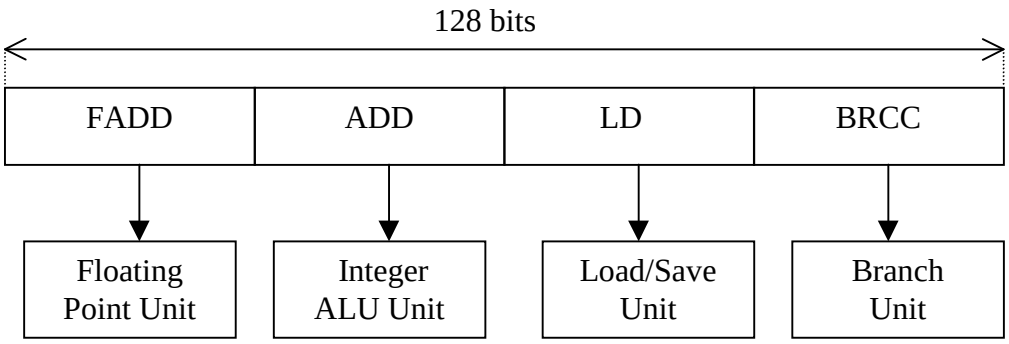

Figure 2-1. Crusoe processor can execute up to four operations in parallel

In comparison to Pentium processor, Crusoe processor does not require the complex decoding and dispatching module. Its structure is simpler, and therefore it consumes much less power. Furthermore, Crusoe processor incorporates dynamic translation system, denoted

as Code Morphing which complies the x86 instruction set into the host VLIW instruction set. The processor can therefore run standard x86 programmes because code translation is invisible to the external system. It should be noted that Pentium processors decode and dispatch instructions every time they are executed. Conversely, Transmeta's software translates instructions once, saving the resulting translation in a translation cache [Kla00]. The next time the (now translated) x86 code is executed, the system skips the translation step and directly executes the existing optimised translation. As in most cases code is executed several times in a loop, translation overheads have little influence on the system performance. Furthermore, the translation is carried out only once and therefore it can implement a complex algorithm which better optimises code than e.g. Pentium processor does. Besides, as an application is executed, Code Morphing 'learns' more about the program and improves it so it will execute faster and faster. For example, aware of the branch history, the programs can favour the most frequently taken path, or execute code from both paths and select the correct result later if both paths are taken with equal probability. It is important to note that Crusoe hardware can achieve excellent performance because it has been designed specifically with dynamic translation in mind.

New Pentium 4 processor adopted a similar but much simpler instruction decoding solution. The hardware instruction decoder can decode maximum one instruction per clock cycle. The decoded instructions are stored in an execution trace cache (TC) [Int00] and then are executed directly from the TC. This removes decoding costs on frequently-executed code. The TC can hold up to 12K μops and can deliver up to three μops per cycle.

2.1.4. SIMD

A single Instruction stream over Multiple Data stream (SIMD) [Fly72] architecture allows a single instruction to be executed on several independent data simultaneously. This significantly simplifies the instruction decoding process, as only a single control unit is required.

In Pentium processors, a MultiMedia eXtension (MMX) [Int97b] coprocessor has been introduced which operates on 64 bit data and therefore can process eight independent 8-bit-wide data simultaneously. In the case of the image convolution, input pixels are in 8-bit unsigned format, however intermediate results are 16-bit wide, and therefore up to four data can be processed simultaneously. Examples of MMX instructions are given in Figure 2-2.

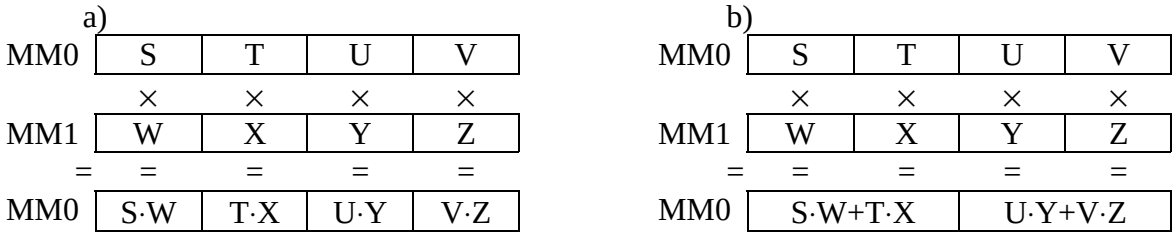

Figure 2-2. Example of MMX instructions: a) multiplication PMULLW MM0, MM1, b) multiply and accumulate (MAC) PMADDWD MM0, MM1

Additional computation power is obtained by superscalar architecture of the MMX coprocessor, as two MMX instructions can be executed in parallel provided that different MMX resources are employed.

Listing 2-3. Fragments of 3×3 convolution programs for different options and convolution kernel given in Figure 1-1a

m)

```
beg:  // label for loop start
movd mm0, dword ptr [ecx] // load row 0
movd mm1, dword ptr [ecx+200h]  // load row 1
movd mm2, dword ptr [ecx+400h]  // load row 2

punpcklbw mm0, mm5 // convert byte to word
punpcklbw mm1, mm5 // mm5 – contains only zeros
punpcklbw mm2, mm5

pmaddwd mm0, mm6 // MAC; mm6= 0, 1,2,1
pmaddwd mm1, mm7 // mm7= 0,2,4,2
pmaddwd mm2, mm6

paddd mm0, mm1 // accumulating the MAC results
paddd mm0, mm2 // result in mm0
// integer units operations
movq [edi], mm0 // save register MMX to memory
movd eax, mm0 // load LSB half of the register MMX
add eax, [edi+4] // add LSB and MSB half of the register MMX
add eax, 8 // add 8 to reduce rounding error
inc esi // increment pointer to the destination
sar eax, 4 // division by 16 – prescaling
inc ecx // increment pointer to the source pixel
mov byte ptr [esi], al. // load the result to memory
dec ebp  // decrement the loop count
jnz beg // finish the loop?
```

n)

```
beg: // label for loop start
movd mm0, dword ptr [ecx] // load pixel (0,0)
punpcklbw mm0, mm7 // convert byte to word; mm7= 0

movd mm1, dword ptr [ecx+1] // load pixel (0,1)
pmullw mm0, mm6 // multiplication for pixel (0,0), mm6= 1,1,1,1
punpcklbw mm1, mm7 // convert byte to word, pixel (0,1)

movd mm2, dword ptr [ecx+2] // load pixel (0,2)
pmullw mm1, mm5 // multiplication for pixel (0,1); mm5= 2,2,2,2
punpcklbw mm2, mm7 // convert byte to word for pixel (0,2)
paddw mm0, mm1 // add products for pixels (0,0) and (0,1)
// continue for the rest of pixels
...
pmullw mm2, mm6 // multiplication for pixel (2,2), mm6= 1,1,1,1
paddw mm0, mm2 // the final result

paddw mm0, mm3 //add to reduce rounding error, mm3= 8,8,8,8
psrlw mm0, 4 // divide by 16
packuswb mm0, mm7 // convert the result from word to byte
add ecx, 4 // increment source pixels pointer
movd dword ptr [esi], mm0 // save the result
add esi,4 // increment the result pointer
dec ebp  // decrement the loop count
jnz beg // quit the loop?
```

Convolution operation can be carried out in two different ways [Wia00a]. In the first one, given in Listing 2-3m only one result is obtained at the time. This solution exploits the MMX multiply and accumulate (MAC) instruction (PMULLW), therefore it might seem that this is the best solution. Unfortunately, input data format causes that every MAC operation performs four multiplies (for every input row) but only three are used (for 3×3 kernel). Besides two partial results are obtained in two halves of the MMX register, and therefore integer units have to be used to carry out the final addition. The alternative solution is presented in Listing 2-3n, for which four results are obtained simultaneously. This option

performs multiply and add instructions independently, but input and output data match better the convolution process. Consequently, MMX instructions are fully exploited (operate on four independent data), and all instructions (except the loop instructions) employ only the MMX unit. Summing up, the latest solution reduces calculation time in comparison to the former solution (see Table 2-1). It should be noted that option *n* allows for saturating the result (e.g. setting the output to the maximum value *(255)* if the result is overflowed *(≥256)*) during conversion from the word (16-bit) to byte (8-bit) format.

New Pentium 4 can operate on 128-bit data using SSE2 instructions which are similar to the MMX instructions [Int00]. Consequently, the speed-up by the use of SIMD instructions will be even greater, and it seems that in the future, new releases of processors will be able to process greater and greater data width in SIMD instructions.

2.1.5. Implementation results

Table 2-1 and Table 2-2 gives implementation results for different processors and different options. The following convolution options have been implemented:

a) standard algorithm written in C language (Listing 2-1),

b) after unrolling the convolution kernel (loops *i, j* in Listing 2-1) written in C language,

c) like option *b* but program is written directly in assembler language,

d) like option *c* – after optimisation for the superscalar architecture,

e) like option *c* but without multiplication (only shifts are implemented, the convolution kernel is given in Figure 1-1a),

f) like option *e* – after optimisation for the superscalar architecture,

g) like option *f* but input and output data pointers are not incremented in order to avoid cash misses,

m) employing MMX coprocessor (Listing 2-3m),

n) employing MMX coprocessor, four pixels are calculated simultaneously (Listing 2-3n).

All options, except option *g*, have been referred in the previous sections. Therefore only option *g* will be now approached. It can be seen from Listing 2-2 and Listing 2-3 that approximately every second instruction communicates with memory and consequently memory-transfer might be a bottleneck of the system. Fortunately, all tested processors incorporate internal cache memory [Hwa93, And95], which significantly reduces external memory transfers. Nevertheless, cache misses might still deteriorate the processor performance [Wia00a]. In option *g* input and output data pointers are not incremented and therefore the convolution operates only on 9 input and 1 output pixels which are in the cache

memory. As a result, the convolution is corrupted, conversely the external memory transfer is not required. Option *g*, in comparison to the corresponding option *f*, gives up to 10% improvement. For the latest versions of the processors, however, the overhead of cache misses increases (see Table 2-1). It should be noted that SSE instructions allow software controlled data-perfecting, which can eliminate the cache misses.

Option	a	b	c	d	e	f	g	m	n
Number of instructions in the loop	-	-	42	42	37	35	35	21	43/4
Time [ms] 486DX4 –100	670	465	460	435	147	134	131	-	-
P75	587	414	403	366	95	70.3	67.6	-	-
P166	247	175	169	159	45	32	30	60	26
P300	48.6	25.8	22.6	22.8	16.6	15.6	14.0	22.1	9.1
Athlon 800MHz	25.8	11.6	10.4	9.9	6	5.5	5	12.6	7.2

Table 2-1. Number of assembler instruction in the loop and calculation time for different processors and options

Option	a	b	c	d	e	f	g	m	n
486DX4 –100	256	177	175	166	56	51	50	-	-
P75	168	118	115	105	27.2	20.1	19.3	-	-
P166	156	111	107	101	28.5	20.3	19.0	38.0	16.5
P300	55.6	29.5	25.9	26.1	19.0	17.9	16.0	25.3	10.4
Athlon 800MHz	78.7	35.4	31.7	30.2	18.3	16.8	15.3	38.5	22.0

Table 2-2. Number of clock cycles required to calculate a single output pixel for different processors and options

2.2. Digital Signal Processors (DSPs)

DSP architectures are optimised for digital data processing, e.g. for convolution, therefore DSPs perform convolution in more efficient way than general purpose processors do. A DSP architecture and its influence on the convolution process is outlined for TMS320C80 processor by Texas Instruments [Tex97]. The convolution process is performed in a Parallel Processor (PP), which will be approached in the next section. Further in Section 2.2.2, the whole structure of the TMS320C80 is overviewed. Finally, a new DSP, TigerSHARC by Analog Devices is briefly described.

2.2.1. Parallel Processors

The PP, shown in Figure 2-3, is a 32-bit integer processor which incorporates the following units:

1. Data Unit (DU).

2. Address Unit (AU).
3. Program Flow Control Unit (PFCU).

These units are optimised for the convolution, and as a result, all fundamental convolution operations:

1. load the coefficient (executed by the AU)
2. increment coefficient pointer (executed by the AU)
3. load the input pixel (executed by the AU)
4. increment input pixel pointer (executed by the AU)
5. multiply, (executed by the DU)
6. accumulate (executed by the DU)
7. control the loop (executed by the PFCU)

can be executed in parallel in a single clock cycle. The PP has also three buses, for transferring a 64 bit VLIW instruction and two (local and global) 32-bit data in a single cycle.

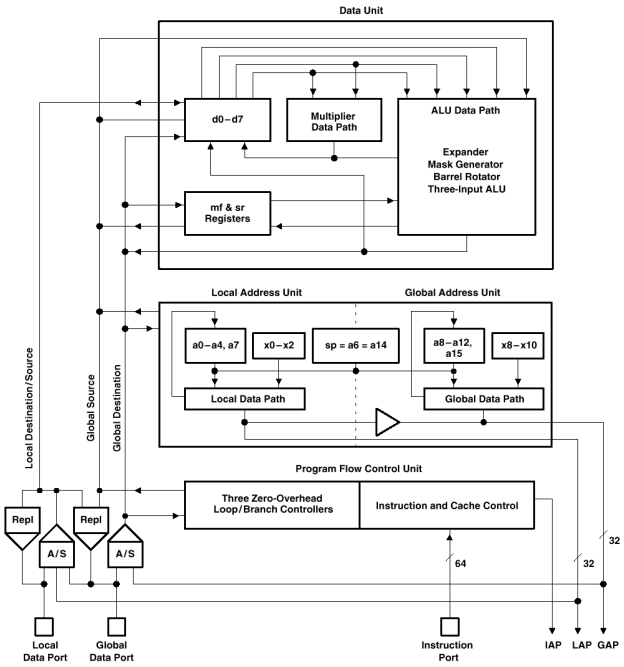

Figure 2-3. The PP block diagram

2.2.1.1. Address Unit (AU)

The PP incorporates two address units: local address unit (LAU) and global address unit (GAU) which operate independently of each other. Each of them is responsible for both accessing memory and computing address locations. The data buses, local and global one, are associated with the AU, which allows two data transfers to be carried out independently in a

single clock cycle. Each AU has five address and three index registers and data path for computing addresses.

Taking into account the convolution, each AU (global and local) allows for accessing memory and simultaneously incrementing the address pointer either by *1* (load the next pixel or the next coefficient) or by N_X-$M+1$ (load the pixel from the next line, where N_X- horizontal image size, M- convolution kernel horizontal size; N_X-$M+1$ is stored in the index register). Consequently, the LAU is responsible for feeding the data unit (DU) with coefficient values, and the GAU for feeding the DU with pixel values (or vice versa).

2.2.1.2. Data Unit (DU)

Data unit basically incorporates two paths: multiplier data path and ALU data path. Consequently multiplication and addition are carried out in parallel, and are executed in a single clock cycle. The multiplier can perform two simultaneous 8-bit by 8-bit multiplies referred as a split multiply. Similarly, two 16-bit additions (a split addition) can be performed in parallel in the ALU. These SIMD operations allow two results to be obtained simultaneously, therefore, in theory, doubling the PP computation power. The ALU instructions allow also for shifting and saturating, which speeds-up the final processing on the convolution result.

2.2.1.3. Program Flow Control Unit (PFCU)

The PFCU has a VLIW architecture, which significantly simplifies decoding and dispatching process. The PFCU has a separate 64-bit instruction bus, and therefore fetching an instruction does not interfere with the data paths. The PP incorporates three stages of pipelining:

1. Instruction fetch
2. Address unit computation
3. Execute data unit operation and memory transfer.

The pipelining of the PP, unlike Pentium processors, must be considered in the program code for every branch or other instructions changing order of a program flow. Two delay-slot instructions following a branch must be legal operations. Listing 2-4 gives an example of proper handling with a branch instruction. The branch instruction is executed in the two delay slots therefore *Instruction2* and *Instruction3* are executed, although in Pentium processors only *Instruction1* is executed in the loop.

Listing 2-4. Example of delayed branch

```
Branch1: Instruction1;  the first instruction in the loop
         br = Branch1; branch instruction, branch to the beginning of the loop
         Instruction2;  delay slot 1
         Instruction3;  delay slot 2
         Instruction4;  the instruction outside the loop
```

The delayed branch schedule in the PP eliminates a branch penalty and simplifies the processor architecture, as the branch prediction and pipeline flushing is not required. The penalty of the delay branch solution is out-of-order program flow which must be considered by a compiler or assembler language programmer. Besides, in some cases a result of the conditional branch is known only after the last instruction of a loop has been executed (*Instruction3* in Listing 2-4). In this case two additional *nop* (no operation) instructions must be inserted after the branch, and therefore the branch is executed with two-cycle penalty, which has a similar result like for pipeline flushing.

The PP incorporates hardware loop control (HLC) which eliminates the loop overhead completely, even the branch instruction is not included in the program code. The following three sets of registers are employed to control up to three loops:

- Loop End Register: le2, le1 or le0 which points to the last instruction in the loop. During each instruction fetch, the le is compared to the program counter (pc). When the le matches the pc, the loop hardware action is invoked.
- Loop Start Register: ls2, ls1 or ls0 which points to the first instruction in the loop. The ls register is copied over the pc when the loop hardware wants to branch back to the beginning of the loop.
- Loop Counter Register: lc2, lc1 or lc0 contains a counter of the number of branches to the start of the loop. The branch is taken provided that lc>0; the lc is decremented each time the branch is taken.
- Loop Reload Register lr2, lr1 or lr0 contains an initialisation value for the associated lc. This reinitialisation takes place after the last time through the loop. This prepares the loop counter for the next time the loop is entered.

In conclusion, for the PP and the HLC, loop unrolling gives, basically, no additional processor speed-up. The HLC is superior to the complex branch prediction solution implemented in the general-purpose processors. It should be noted that the HLC gives very good result if the

loop-count is well-defined (as it is the case for the convolution), however in a general case, unpredictable loop breaking often takes place, which makes the HLC useless.

To illustrate how the convolution is implemented in the PP, a fragment of the convolution code is given in Listing 2-5. This code is executed in a single clock cycle.

Listing 2-5. Fragment of the convolution code for the PP

```
mult =m pixel * filter ||          ; Multiplication
result += mult ||                  ; Accumulate the product of the previous multiplication
filter =b *Ga_filt++ ||            ; Load the next coefficient and increment the coefficient address
pixel =ub *(La_pt++=offset1)       ; Load the next pixel and increment the address to point the next line
                                   input pixel
```

2.2.2. DSP TMS320C80

TMS320C80 incorporates (see Figure 2-4):

- Four Parallel Processors (PP) described in Section 2.2.1.
- Floating-point Master Processor (MP) which is a 32-bit RISC which primary role is to perform the general-purpose computations necessary to direct the MP's on-chip resources.
- Video Controller (VC) which provides the video interface to control two independent frame systems.
- On-chip 50kB memory (organised into 2KB blocks) accessed by crossbar switching architecture.
- Transfer Controller (TC) provides an interface between the TMS320C80 processors – the MP, PPs, VC and external memory.

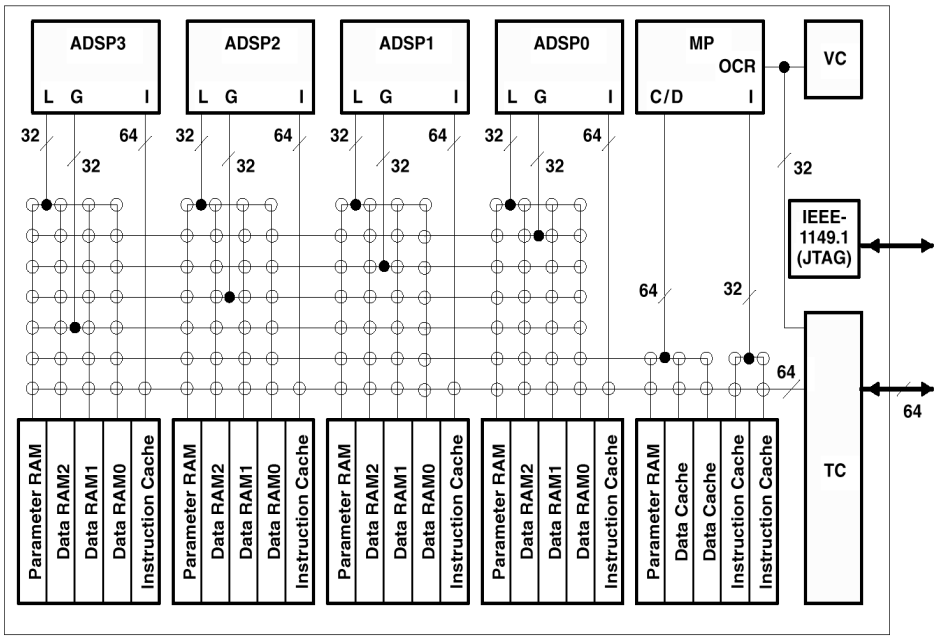

Figure 2-4. Block diagram of TMS320C80

Each PP can perform two independent parallel data accesses to the on-chip shared RAMs and one instruction fetch each cycle as each PP incorporates three ports:

- Instruction port accesses instructions from the PP's instruction cache. Each PP has its own 2K-byte instruction cache for storing up to the 256 64-bit instructions. This amount of cache memory is large enough to store the convolution program code.
- Global Port connects to any of the shared RAMs. If the access is attempted over the global port to the address that is not in the on-chip RAM, a direct external access (DEA) request is sent to the transfer controller. A DEA requires minimum 11 cycles for a load and 8 cycles for a store, therefore is not recommended unless only for a few bytes of data. For a block transfer, the Transfer Controller should be programmed to execute a Pocket Transfer which is performed in parallel with normal PP operations.
- Local Port connects a PP to any of four local RAMs. If a PP attempt a memory access over the local port to an address that is not in its local RAMs, the access is diverted to the global port and tried on the following cycle. This causes one cycle penalty.

Each memory block can be accessed only once in a clock cycle. Therefore, taking into account the convolution operation, triple buffering technique should be employed [Tex97], for which three separate RAMs are used for input pixels, output pixels and packet transfers to/from the external memory. Table 2.3 illustrates this technique for which different memory blocks are assigned for the different transfers, depending on the calculated output pixels. The assignment is repeated in a cycle for every 6k output pixels.

Output Pixels	[0, 2k)	[2k, 4k)	[4k, 6k)
RAM0	Input Data	Output Data	Packet Transfer
RAM1	Packet Transfer	Input Data	Output Data
RAM2	Output Data	Packet Transfer	Input Data

Table 2-3. Memory assignment for different time intervals (output pixels)

The above algorithm, however, does not consider the input blocks overlapping (the padding effect); e.g. for convolution kernel 5×5 and image size 512×512, to compute 1k output pixels (2 image lines) more than 3k input pixels are required (additional 2 input image lines before and after the corresponding destination pixels). Table 2-4 shows a modified algorithm. Nevertheless, for a convolution kernel greater than 5×5, this algorithm cannot be used, and more than three RAMs must be assigned to the PP and consequently another PP might not be able to compute its own task.

The convolution process can be easily performed in parallel, every PP and even MP calculate a separate output pixel block. TMS320C80 incorporates four parallel processors and each of them can calculate a separate part of the output image; as a result, the calculation time is, in theory, four times shorter. However, the processors (PPs, MP and TC) synchronisation is a key issue. Therefore TMS320C80 incorporated special interprocessor commands: reset, halt, unhalt, task interrupt, message interrupt and instruction-cache reset.

<table>
<tr><td>Output Pix.</td><td colspan="2">[0, 1k)</td><td colspan="2">[1k, 2k)</td><td colspan="2">[2k, 3k)</td><td colspan="2">[3k, 4k)</td><td colspan="2">[4k, 5k)</td><td colspan="2">[5k, 6k)</td></tr>
<tr><td>RAM0</td><td colspan="2">ID</td><td colspan="2">ID</td><td>OD</td><td>ID</td><td colspan="2">OD</td><td colspan="2">PT</td><td>ID</td><td></td></tr>
<tr><td>RAM1</td><td colspan="2">PT</td><td>ID</td><td></td><td colspan="2">ID</td><td colspan="2">ID</td><td>OD</td><td>ID</td><td colspan="2">OD</td></tr>
<tr><td>RAM2</td><td>OD</td><td>ID</td><td colspan="2">OD</td><td colspan="2">PT</td><td>ID</td><td colspan="2">ID</td><td colspan="2">ID</td></tr>
</table>

Table 2-4. Memory assignment input blocks overlapping. Where: ID- Input Data, OD- Output Data, PT- Pocket Transfer

According to [Tex97] each PP requires 11 cycles (8.5 cycles for split operation) to perform 3×3 convolution. This for image size 512×512 and four parallel processors clocked by 50 MHz gives convolution theoretical time 14.5ms (11.2ms for split operations). Experimental result obtained by [Matrox] is 10.3ms.

Summing up, TMS320C80 is a very sophisticated processor which architecture is optimised for convolution operation, thus it has been characterised hereby. Conversely, it is rather difficult to be programmed, because of triple or even more buffering technique, memory contentions, interprocessor synchronisation, etc. Besides it is clocked only by 50MHz (designed in 1995 and not developed since then). Consequently TMS320C80 is not recommended for new designs.

2.2.3. TigerSHARC

Current DSP architectures are more or less similar to general-purpose processors and TMS320C80. TigerSHARC ADSP-TS001 by Analog Devices [Ana99] is an example of such a DSP. TigerSHARC is clocked by 150MHz and incorporates the following blocks (see Figure 2-5):

- Two Compute Blocks (CB) that can operate either independently in parallel or as a SIMD engine. The DSP can issue each cycle up to two compute instructions per CB, instructing the ALU, multiplier or shifter to perform independent, simultaneous operations. Taking into account the convolution, the multiplier block performs a single 32-bit MAC or quad 16-bit SIMD MACs (8 MACs in two CBs) in a cycle. Unfortunately, the multiplier does

not support the byte operations, however, the ALU can operate on 8-bit arguments to produce 16-bit results. The CBs operates in a very similar way like the PPs in TMS320C80, however each CB has only one data bus (there are two data buses associated with each PP in TMS320C80). Conversely, data bus is 128-bit wide, therefore instead of four consecutive 32-bit data accesses, a single data access is performed.

- Two Integer ALUs (IALU, denoted as J-ALU and K-ALU in Figure 2-5) that provide powerful address generation capabilities and perform move operations.
- Program Sequencer has a static superscalar architecture. The term "static superscalar" is applied because instruction-level parallelism is determined prior to run-time and encoded in the program. Therefore decoding instruction is easier than in Pentium processors but the compiler or programmer has to respect instruction dependency caused either by resources sharing or pipelining, etc. The DSP uses the following pipeline stages: three instruction fetch (together with Instruction Alignment Buffer which incorporates instructions FIFO buffer and dispatches instructions to the DSP units), a decode, integer, operand access, execute1 and execute2. Consequently to reduce branch stalls caused by the pipelining, the DSP incorporates Branch Target Buffer (BTB). This solution is similar to Pentium processor, however, the branch prediction algorithm is much simpler in this DSP – a branch is taken if the branch was also taken in the previous iteration of the loop.
- The TigerSHARC DSP contains three blocks of 2 Mbits each of on-chip, 128-bit wide SRAM. The processor has also three 128-bit wide buses, each one connected to one of the internal memories. Memories (and their associated buses) are a resource that must be shared between the CB, the IALUs, the program sequencer and the external port. In general, if during a particular cycle more than one unit in the processor attempts to access the same memory, one of the competing units is granted access, while the other is held off for further arbitration until the following cycle. However, because of the large bandwidth available from each memory block (128-bit memory access) some bandwidth is available for use by another unit.
- DMA peripheries. The most effective way to access external data in the TigerSHARC is through the DMA. This runs in the background, allowing the core to continue processing while new data is read in or processed data is written out.

The TigerSHARC offers powerful features, tailored to off-chip multi-processing systems. However, like for TMS320C80, parallel processing complicates the design, programming and may cause bus conflicts overhead.

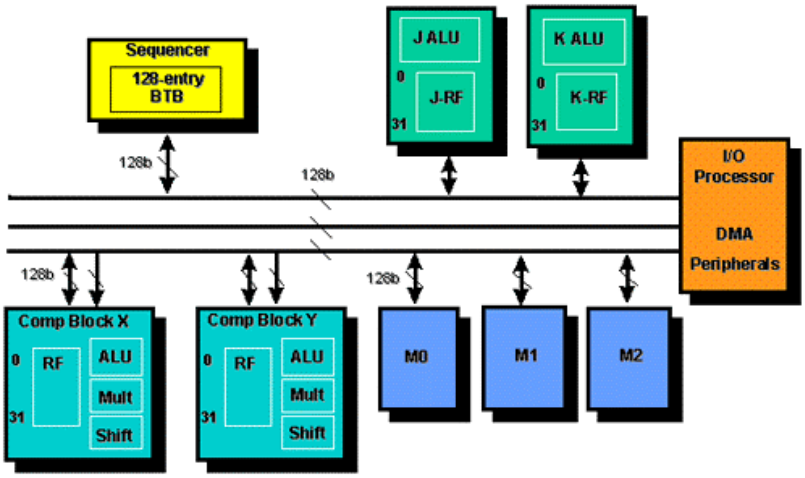

Figure 2-5. Block diagram of TigerSharc by Analog Devices

2.3. Dedicated VLSI Convolution Processors

An alternative architecture for performing convolution is a dedicated VLSI processor where the convolution operation is built-in the silicon structure. Consequently, there is no complex instruction decoding and large cache module. The architecture of the processor is optimised only for the convolution operation. The typical structure of the VLSI processor is given in Figure 2-6, and incorporates multipliers and adders to perform arithmetic operations, and delay elements to feed the arithmetic block with proper data. Two different delays are required:

- Pixel buffer – flip-flops to delay input signal by a single clock cycle (z^{-1} blocks in Figure 2-6).
- Line buffer – First-In First-Out (FIFO) buffer to delay input signal by a whole image line (in the considered solution, by 512 clock cycles).

Different vendors produce different VLSI convoler chips, e.g. HSP-48908 by Harris [Har94], PDSP-16488 by Plessey [Ple90] or IMS-A110 by SGS Thomson [Tho90]. Detailed description of these processors is outside the scope of this thesis, and is included in [Wia99a]. Hereby, only IMS-A110 will be further approached.

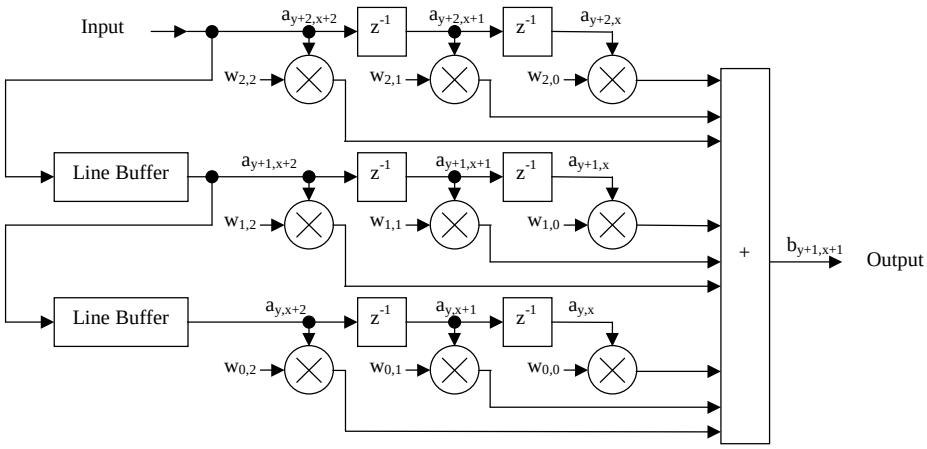

Figure 2-6. VLSI architecture for 3×3 convolution

IMS-A110, which block structure is given in Figure 2-7, incorporates 3 arithmetic blocks, 3 line buffers and asynchronous function block. Each arithmetic block performs up to 7 MACs; in total, 3×7 convolution kernel is supported. The coefficients word width is 8 bits. Two banks of coefficients are provided, thus in any instant one set of coefficients is in use, and the other set can be altered. Three shift registers (line buffers) are 8 bit wide and each programmable from 0 to 1120 clock cycles in length, therefore different resolutions of the image can be processed on. IMS-A110 has also an advance post-processing block which allows for shifting right with rounding, basic statistics monitor (e.g. maximum and minimum output value), saturation, Look-Up Table (LUT) for 8-bit to 8 bit transformation [Cas96]. IMS-A110 is controlled via a host microprocessor interface, which is independent to the image processing data path.

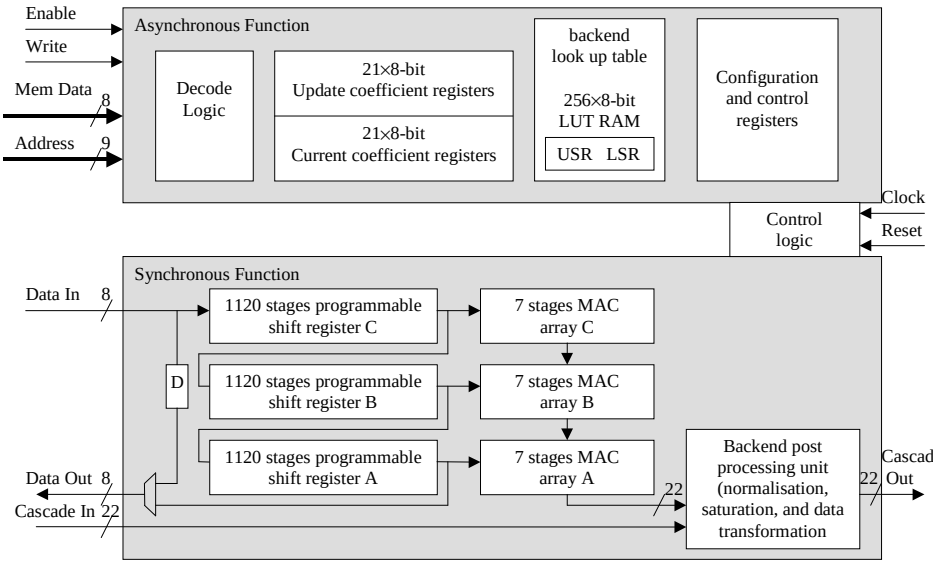

Figure 2-7. Block Diagram of IMS A110

IMS-A110 is clocked by 20 MHz (the chip was issued in 1990) which is rather insignificant frequency. However, all operations are performed in parallel; hence for 3×3 convolution, 9 MACs are computed in each clock cycle. In consequence, computation power of IMS-A100 is comparable to TMS320C80 which operates at 50 MHz. To increase the convolution kernel, several IMS-A110 can be cascaded without any additional logic or time overheads. There are other dedicated VLSI convolers [Wia00a] which can be clocked with greater frequency, e.g. PDSP-16488 by Plessey [Ple90] clocked at 40MHz. IMS-A110 has been described hereby because of its advance post-processing unit.

2.4. Field Programmable Gate Arrays (FPGAs)

A FPGA [Xil99b, Alt99] convolution processor [Jam01d] is, similarly like the dedicated VLSI processor, implemented according to the block diagram presented in Figure 2-6. Nevertheless in Chapter 5 additional modifications of this circuit are presented, which significantly reduces the hardware requirements of a FPGA-based convoler. For FPGAs, a designer defines the structure of the convoler, therefore he has to be familiar with the digital circuit design. Nonetheless, there are several tools which automatically generate arithmetic, delay units or a memory interface, e.g. CoreGenerator by Xilinx [Xil99] or the AuToCon [Wia00b, Wia01a]. Conversely, the designer can allocate FPGA's resources up to his needs; e.g. the size of the convolution kernel, multipliers and adders width, post-processing operations, etc. For a dedicated VLSI processor, the designer is constrained by its pre-designed functions.

For FPGAs further savings can be obtained. Instead of employing a fully-functional, denoted as Variable Coefficient Multiplier (VCM), the designer should employ a Constant Coefficient Multiplier (KCM), as the coefficient values usually do not change during a calculation process. This causes significant hardware savings [Wia01a]. In some cases, a coefficient value is a power of two (e.g. for filters in Figure 1-1), thus the multiplication can be substituted by an addition and a bit-shift. When coefficient values are relatively constant, e.g. changed every image frame, a dynamic constant coefficient multiplier [Wia00b] should be employed, which is a midway solution between the KCM and VCM. In addition, FPGAs can be quickly reconfigured, and this allows for (dynamic) change of FPGAs functions, e.g. reconfiguration of the FPGA every time a coefficient is changed.

To illustrate the architecture of FPGAs, an example of Virtex family by Xilinx [Xil99] is given hereby. Basically Virtex comprises of four major components:

a) Input/Output Blocks (IOBs) which interface between chip internal and external signals.

b) Configurable Logic Blocks (CLBs), shown in Figure 2-8, perform logic and arithmetic functions. Logic functions are performed mainly in 16×1 Look-Up Tables (LUTs), allowing to carry out any 4 input function. Instead of logic, each LUT supports a 16×1-bit synchronous RAM. Furthermore, the two LUTs can be combined to create a 16×2 or 32×1 RAM, or a 16×1 dual-port RAM. A storage element, which is usually configured as a D-type flip-flop is associated with each LUT. This allows for efficient implementation of pipelining. A ripple carry addition [Omo94] $S= A+B$ is performed according to the following equations:

$$s_i = a_i \oplus b_i \oplus c_{i-1} \tag{2-1}$$

$$c_i = \begin{cases} c_{i-1} & if\ a_i \neq b_i \\ a_i & if\ a_i = b_i \end{cases} \tag{2-2}$$

Eq. 2-1 is performed in the LUT ($O_{LUT}= a_i \oplus b_i$) and the XOR gate (see Figure 2-8). The multiplexing in eq. 2-2 is performed in CY multiplexer. It should be noted that carry logic is produced in the dedicated and therefore very fast circuit. This allows for building very fast and efficient adders in the FPGAs. The 16×1 LUT or RAM together with the dedicated carry logic and the storage element is further denoted as a Logic Element (LE).

c) Block SelectRAMs (BSRs) of 4kbs each, have been introduced in Virtex FPGAs. The data width of a BSR is programmable to 1, 2, 4, 8 or 16 bits. For convolution, the BSRs are generally employed as a line buffers (see Figure 2-6). The BSR of 8-bit width is 512 in depth, therefore ideally suits the 8-bit image data and 512×512 image resolution. BSRs can be also employed as LUT memories in multipliers.

d) Programmable routing (PR) provides connections between IOBs, CLBs and BSRs. The propagation time through the PR is comparable (or even greater) than the propagation time through logic in CLBs. Therefore routing optimisation by a place-and-route software is an important aspect of every design and therefore a great number of papers have studied the subject e.g. [Nag98]. Routing delays increase with the decrease of free CLBs resources, therefore it is not recommended to allocate more than 80-90% of available CLBs. Furthermore, throughput of the convoler is strongly influenced by the FPGA logic capacity and other functional units implemented in the same FPGA. This makes system benchmarking more difficult. The PR provides tri-state buffers, which allows multiplexing to be performed outside CLBs. This multiplexing solution saves CLB resources, however, is significantly slower than multiplexing in CLBs.

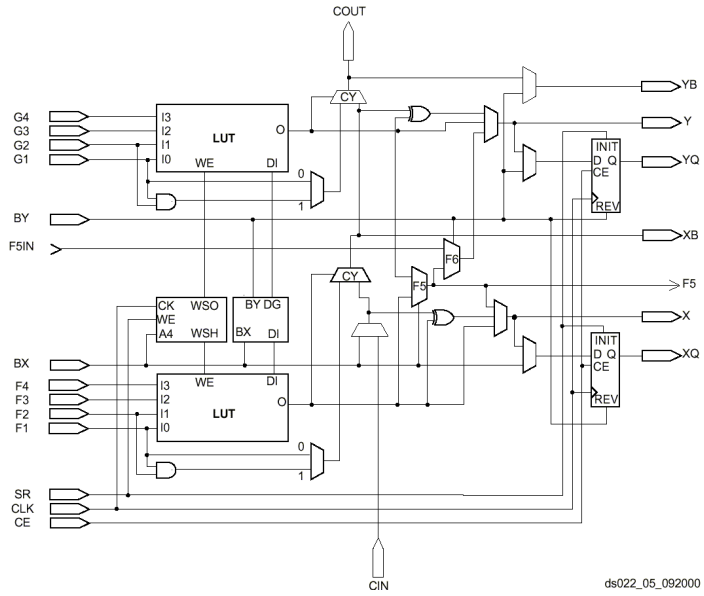

Figure 2-8. Structure of the Virtex Slice = ½CLB

On average, 8×8 KCM occupies roughly 20 LE [Wia01a] and a 16-bit adder occupies roughly 16 LEs. Virtex XCV3200E incorporates 104×156 CLB array which gives 64 896 LEs, enough to fit roughly 1800 a single clock cycle MACs. The clock frequency is up to the 130MHz [Xil00], which gives, in theory, 230 GMACs per second. Consequently, FPGAs significantly outperform e.g. TigerSharc DSP which can perform, in theory, 1.2 GMACs or Athlon 800MHz with 0.33 GMACs per second.

2.5. Conclusions

A general-purpose processor, in spite of its complex architecture, is the easiest solution for implementation of convolution, because the processor and its development environment are commonly available. Conversely, optimisation of the convolution code requires assembler and system level programming which knowledge is limited. However the primary drawback of the general-purpose processor is tasks sharing; the convolution operation may interfere with other tasks and vice versa. Besides in spite of its rapid computational speed-up observed in the history, the processor is still not able to process large convolution kernels and image resolutions.

An alternative solution is a DSP. DSP architectures are optimised for the convolution operation, however they evolve similarly like the architecture of the general-purpose processors. DSPs incorporates cache memories, branch prediction, sophisticated pipelining, SIMD instructions. Similarly like for general-purpose processors, complexity of DSPs

increases rapidly. It should be noted that clock frequency for DSPs is significantly lower than for the general-purpose processor.

A dedicated VLSI convolution processor initially seems to be the best solution because the architecture of the processor is optimised only for the convolution operation. Consequently, no complex instruction decoding, branch prediction and cache memory is required. Conversely, the dedicated processor design is not flexible, for example, an increase of the convolution kernel or the image resolution, etc., may make the processor useless. Consequently, a dedicated processor often incorporates functions which are seldom employed. For example, three IMS-A110 chips can be cascaded to support 7×7 convolution kernel. Nevertheless, during system-development it might happen that 9×9 kernel is required, this makes the design useless; or only a 3×3 kernel is needed therefore the design has two chips overheads.

FPGAs are an option to the dedicated VLSI processors. FPGA's architecture can be quickly modified which is an important issue because not only the FPGA design is more flexible but also design development and test are much quicker and easier. These are major reasons why dedicated VLSI convolution processors are, nowadays, infrequently adopted. Also manufacturing cost of the VLSI processors is often greater than for FPGAs, as for example Xilinx Inc. introduced cheap Spartan family for ASIC replacement.

Comparing FPGAs to the general-purpose processors and DSPs it can be seen that the architecture of the microprocessors begins to saturate; a significant increase of hardware complexity results in a much less significant seed-up. Therefore in the future, further development of the complex superscalar processors is unlikely. Instead many processors will operate in parallel or a simultaneous-multithreading processor will be introduced. This however will increase the demand for the cache memory which occupies significant chip area. Furthermore, parallel processing has several drawbacks like memory access contention, multiple-threads synchronisation, etc. [Hwa94], which significantly complicates the architecture and programming of parallel systems. Besides, obtained speed-up is often not proportional to the number of additional parallel units. In the future, further increase of the microprocessors computational power may be also confined by CMOS technology limits [Won99]. Conversely, FPGAs are very scalable on highly concurrent tasks, esp. convolution. Furthermore, taking into account the silicon area and throughput, the FPGAs significantly outperform microprocessors [DeH98], and in the future the performance gap will further increase. Microprocessors must confront overheating effect, which significantly constrains design of the microprocessors. For FPGAs, however, this problem is much less significant, and often, is not considered at all.

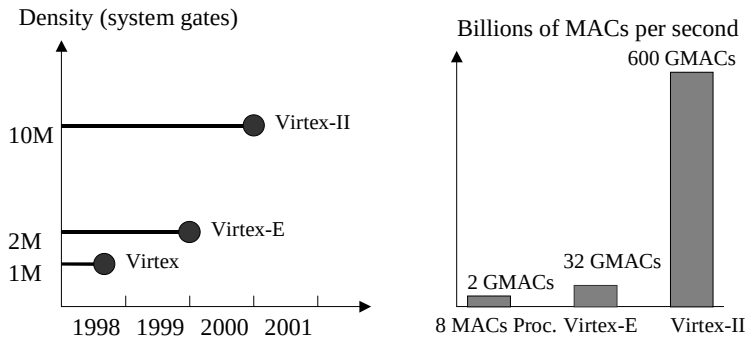

Figure 2-9. History of Virtex family density grow, and number of MACs per second according to Xilinx Inc.

It should be noted that FPGAs density increases very rapidly (about 10 times in two years) and FPGAs expand much quicker than microprocessors do. According to Figure 2-9 FPGA convolution processors are capable of performing roughly 100 times more MACs per second than DSPs do. In conclusion, FPGAs seem to be the most efficient and prosperous architecture for the future.

As a result, a great number of FPGA-based coprocessors have been developed in order to implement computationally demanding operations on FPGAs [DeH98, Hut97, Lis97, Kos97]. Reconfiguration techniques have been also proposed for microprocessors to improve their efficiency [Xu00], and Configurable Computing Machines (CCMs) [Kur00], relaying on FPGAs, have been constructed. High performance of CCMs is achieved by (dynamically) building custom computational operators, pathways, and pipeline suited to specific properties of the task. Furthermore FPGAs are often used as a part of embedded systems, for which application specific hardware together with software solutions are incorporated. Optimal (automated) partitioning algorithms to hardware and software (denoted as codesign) is, nowadays, a major issue for system designers [Dic98, Slo00].

3. Constant Coefficient Multiplication (KCM)

Multiplication is a fundamental operation performed in the convolution. The way a multiplication is carried out in ASIC and FPGA designs initially seems to be very similar. Both ASICs and FPGAs require the same algorithms to be implemented. For example the structure of parallel-array multipliers [Was78] or Wallace tree multipliers [Wal64] for FPGAs and ASICs are very similar. Nevertheless, the most important advantage of FPGAs over ASICs is reconfiguration which allows for a change of the multiplication coefficient either by the change of the multiplicand (an input to a fully functional multiplier denoted further as Variable Coefficient Multiplier VCM) or by the change of a Constant Coefficient Multiplier (KCM) circuit. The KCM, in comparison to the VCM, has much lower hardware requirements [Cha96, Pet95, Wia01a], and therefore is recommended providing that a coefficient value is relatively constant during the calculation process [Wir97, Wia01a].

FPGAs implement logic cells as a Look Up Table (LUT) memory, therefore the inherent way of performing multiplication seems the LUT based Multiplication (LM) for which large LUT memory is split and combined with adders [Chap94, Chap96, Pet95, Wia00b]. Conversely, ASIC solutions usually implement the KCM employing a Multiplierless Multiplication (MM), where multiplication employs additions and subtractions [Pir98, Par97]. Therefore, architecture of a multiplier for FPGAs and ASICs appears to be different. However, after dedicated carry logic has been incorporated into FPGAs, a ripple carry adder [Omo94] occupies half of the previous area and its speed has increased rapidly [Xil96]. This improvement has not been considered to re-establish the actual relation between the MM and LM architectures, and therefore the MM architecture has been overlooked. Summing up, in this chapter, the LM and MM architectures and their cost/speed relations in FPGAs are investigated.

Recently, large memory blocks, which primary intention is to store large amount of data, have been introduced to FPGAs. In some applications, these memory blocks are not occupied, which leads to a waste of chip area. However, these memory blocks can be used as a part of a arithmetic unit very efficiently, therefore a novel synthesis approach is hereby proposed. This approach combines different memory blocks and finds the best architecture of a multiplier according to the cost-relations between FPGA resources [Wia00b]. A similar

approach has been proposed independently and almost at the same time for logic synthesis by [Wil00], and is denoted as heterogeneous technology mapping.

In the first part of the chapter, the Multiplierless Multiplication (MM) employing the Canonic Sign Digit (CSD) and Sub-structure Sharing (SS) methods, is investigated. As the consequence of restrictions put on the FPGA dedicated adders (or subtractors) structure, a modified algorithm for conversion from two's complement to CSD representation is derived, which allows substantial hardware savings. In the next part of this chapter, 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 of a multiplier is a complex task which is addressed hereby, and novel architectural solutions are proposed. In addition, some memory cells have a shorter address width and two or more memory cells may contain the same data, therefore a single (common) memory can be implemented instead. It should be noted that every part of the research is followed by implementation results, which helps to analyse aspects of the considered architectures.

3.1. Multiplierless multiplication (MM)

A constant coefficient multiplier is usually implemented in a multiplierless fashion by using only shifts and adders 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, i.e. 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 [Sam89]). However, in this paper the assumption is made that the value of the coefficient is an input parameter to the design and therefore the coefficient value cannot be changed.

3.1.1. 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 [Gar65, Pir98]. 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 [Pir98] as in Figure 3-1.

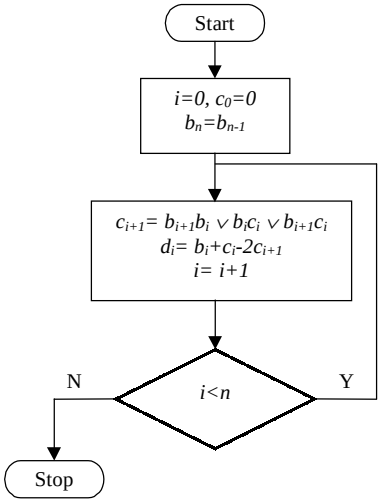

Figure 3-1. The CSD conversion algorithm

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 given 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 [Har96]. This in turn implies hardware savings of about 33% per coefficient.

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

$$s_i= a_i \text{ xor } b_i \text{ xor } c_i \tag{3-1}$$

$$\text{if } a_i = b_i \text{ then } c_i= a_i \text{ else } c_i= c_{i-1}. \tag{3-2}$$

Similarly subtraction $S= A – B$ can be expressed as

$$s_i= a_i \text{ xor not } b_i \text{ xor } c_i \tag{3-3}$$

$$\text{if } a_i = \text{not } b_i \text{ then } c_i= a_i \text{ else } c_i= c_{i-1}. \tag{3-4}$$

For FPGAs (e.g. Virtex), eqs 3-1 and 3-3 are implemented in LUTs and the XOR gates (see Figure 2-8). Eqs 3-2 and 3-4 are implemented in the dedicated carry logic circuit. It can be also seen from eqs 3-1÷3-4, 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 justified. However, in the case of the addition for which the first argument is shifted to the left, the least significant bits (LSBs) of the second argument can be directly copied to the adder output, therefore additional hardware savings are achieved. Unfortunately, for the subtraction, the subtrahend's bits next to the LSB cannot be directly copied on the output because the subtrahend bits have to be negated 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.

3.1.2. Modified algorithm for conversion to the CSD representation

The standard algorithm for 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, a modified conversion algorithm has been implemented [Wia00b], so that the conversion to the $\bar{1}$ symbol (subtraction) takes place only if the total number of operations (non-zero symbols) decreases.

In order to describe the modified 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 multiplicand *B* ($M = A \cdot B$) and let define *Q(i,j)* in the iterative way as follows:

$$Q(i,i) = 0$$
$$Q(i,j+1) = \begin{cases} Q(i,j)+1 & if\, b_{j+1} = 1 \\ Q(i,j)-1 & if\, b_{j+1} = 0 \end{cases} \qquad (3\text{-}5)$$

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

The modified algorithm is shown in Figure 3-2. It should be noted 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 function *Q(i,j)*). 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 is inserted).

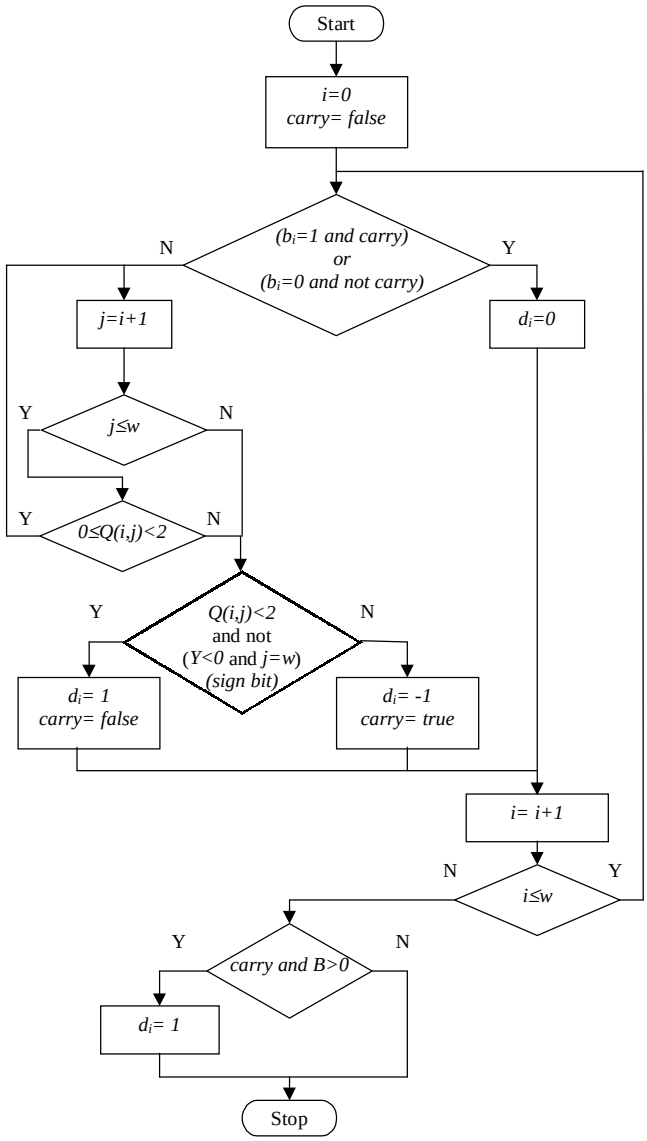

Figure 3-2. The modified algorithm for the conversion from the two's complement to CSD representation; b_i- i-th bit of the binary coefficient, Q()- function defined in eq. 3-5, w- the index of the MSB of multiplicand B

An example of results for the standard and modified CSD conversion (MCSD) is given in Table 3-1. In can be seen that for the coefficient values: 3, 11 and 13 symbol $\bar{1}$ has not been inserted for the MCSD as the number of operation would not decrease.

Coefficient	Binary (TC)	CSD	MCSD
3	11	$10\bar{1}$	11
7	111	$100\bar{1}$	$100\bar{1}$
11	1011	$10\bar{1}0\bar{1}$	1011
23	10111	$10\bar{1}00\bar{1}$	$1100\bar{1}$

Table 3-1. An example of results for the standard CSD and Modified CSD (MCSD) conversions

3.1.3. Substructure sharing

Additional area reduction can also be achieved by Sub-structure Sharing (SS) [Har96, Par97]. 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:

$$tmp = a + (a<<1) \qquad (3\text{-}6)$$

$$27 \cdot a = tmp + (tmp<<3).$$

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

3.1.4. Experimental results

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

K	Average		Maximum			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

Table 3-2. Average and maximum number of CLBs (XC4000) and corresponding number of operations L (additions and subtractions), and the most hardware consuming coefficient

Results for 8-bit unsigned input (the most common data format for image processing) and coefficient width *K=3-12* (coefficient values [1, 2^K-1]) are shown in Table 3-2. These results were obtained for the best algorithm (selected separately for every coefficient) of two's complement representation (TC), CSD, SS, and CSD-SS – applying the SS on the CSD representation. The best algorithm occurrence presents how many times each algorithm gives

the best result. If results of two algorithms are the same, the simplest (former) solution is taken.

It can be seen from Table 3-2 that the optimisation techniques (CSD, SS, CSD-SS) are more attractive for large coefficient widths. The average number of operations *L* for the TC is roughly *K/2-1*, which for *K*= 12 gives *L*= 5 in comparison with *L*= 3.08 for the optimisation. However the cost of a multiplier increases almost linearly with the increase of a coefficient size *K* as it is shown in Figure 3-3. In conclusion, 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.

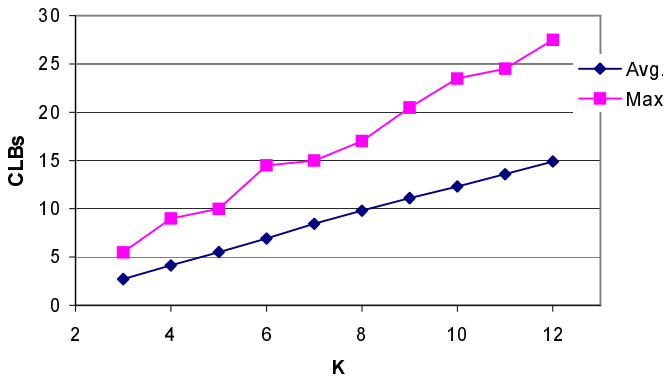

Figure 3-3. Average and maximum area occupied by the 8-bit input multiplier for different widths of coefficients K

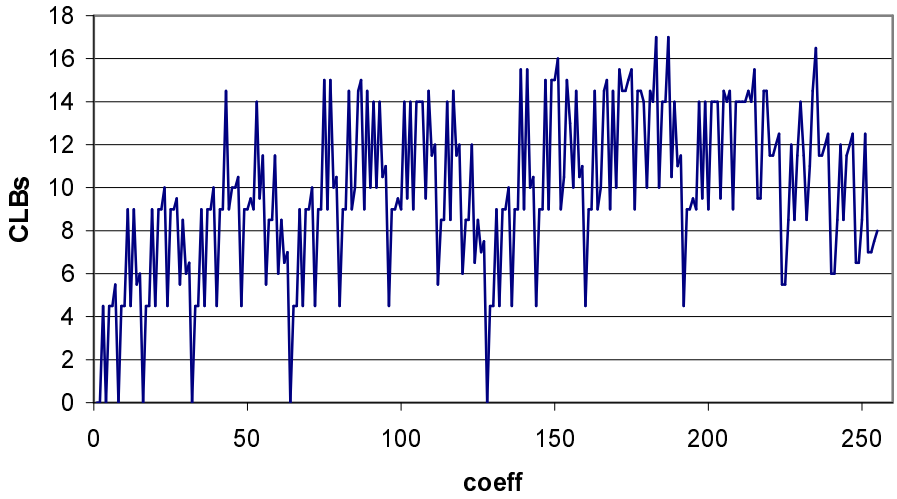

Figure 3-4. Cost of the 8-bit-wide input multiplier for XC4000 and different coefficients

Cost of the multiplier depends strongly on a given coefficient value, which is shown in Figure 3-4. It can be seen that the costliest multipliers are for the coefficient values in 60-75% of the coefficient full binary range.

3.2. LUT based Multiplication (LM)

3.2.1. 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 [Chap96, Omo95]. An example of the multiplier circuit for $K=8$ and $L=8$ is shown in Figure 3-5.

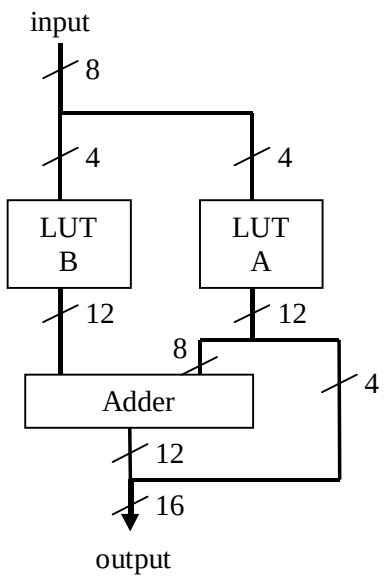

Figure 3-5. The LM for input argument width L=8 and coefficient width K=8

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

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 3-3. 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

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).

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.

3.2.2. 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 2 LEs (2LEs= 1 CLB for XC4000 or ½ CLB of Virtex), which is twice the cost of 16×1-memory module (1 LE). The cost of the adder is 1 LE/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 zero.

Finding optimum combination of different memory modules and adders is a difficult task, and the best solution depends on a size of input data and a given coefficient value. The

LM incorporating only 16×1 memory modules has been presented in [Cha96, Gre97, Pas01, Woj99], 16×1 or 32×1 [Ser01], however no hint for optimal combination of different memories or implementation of any of optimisation techniques described in Section 3.2.1 has been given.

As a result of the author research, the following conclusion has been derived. 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 equal 6 is shown in Figure 3-6. The hardware requirements for these circuits for coefficient equal 43 are: a) 11 ½, b) 12 and c) 12 CLBs of 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).

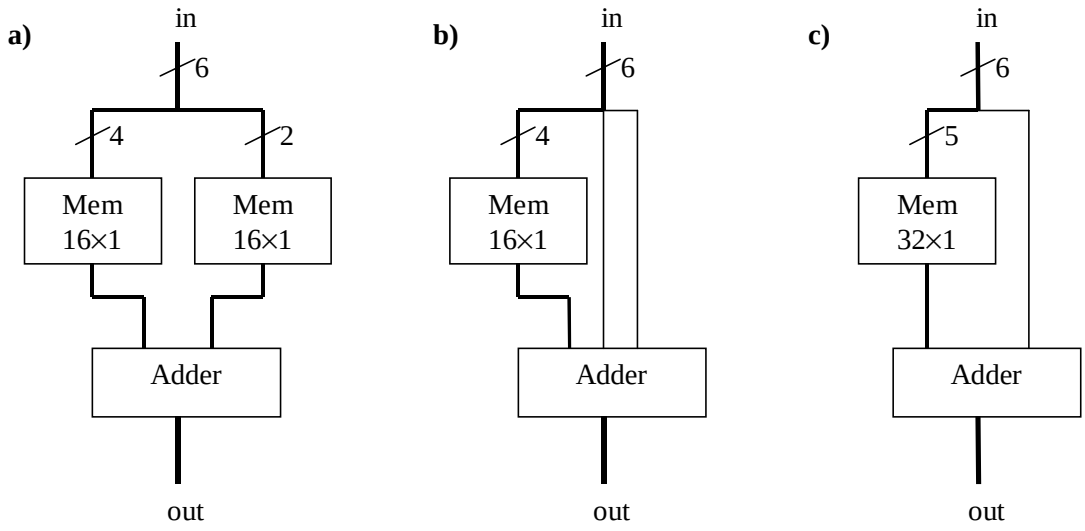

Figure 3-6. Different reasonable methods for implementing the multiplier with the input bus width equals 6

The design task is even more complicated for Virtex family. 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 [Xil99b]. The area in silicon, occupied by a BSR is equivalent to roughly 16 Virtex CLBs (64 LEs). However the actual cost of these memories may differ with respect to free FPGA resources, e.g. a design does not implement any BSRs but uses all CLBs. Consequently, trade-off for distributed RAMs and BSRs is design-dependent. However general conclusions can be derived from Figure 3-7. On

average, the equivalent cost of the BSR 256×16 is about 8÷11 Virtex CLBs (VCLBs) (=32÷44 LEs).

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 a small input and coefficient width (<8), the equivalent cost of the BSR is rather small (see Figure 3-7). For the width 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 an effect of quantization and distribution of the large BSR. 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.

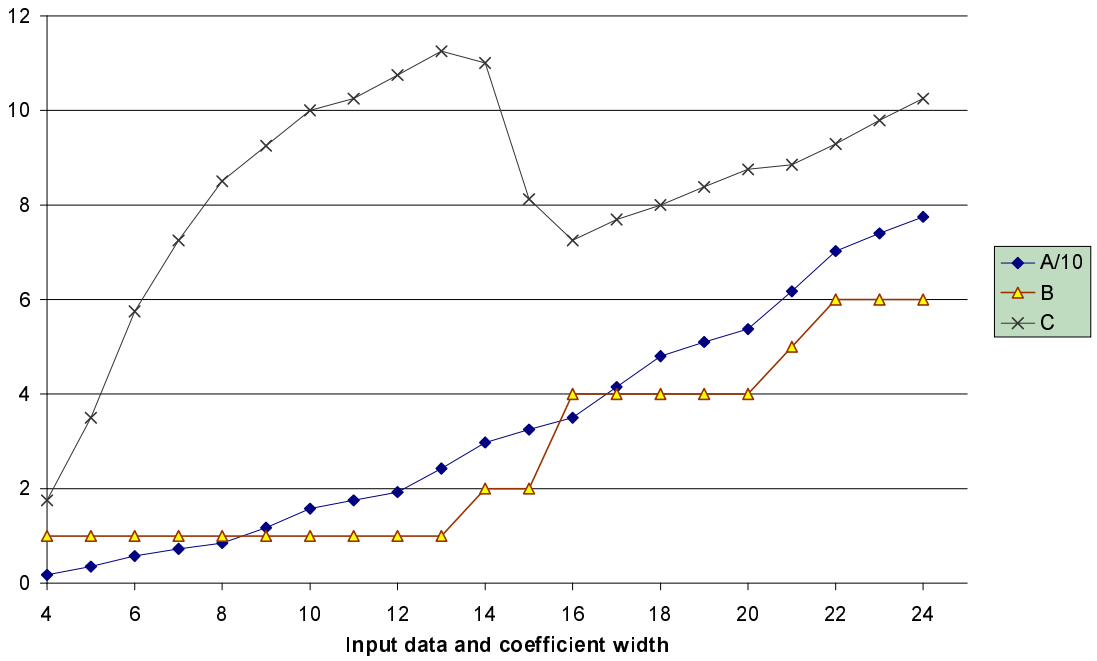

Figure 3-7. The area of the LM for the different input and coefficient width. A- area (in Virtex CLB) scaled of 1:10, for the LM using only distributed 16×1 and 32×1 RAMs, B- number of used 256×16 BSRs, C- equivalent cost (in Virtex CLB) of a BSR in comparison to distributed RAMs- only option

Figure 3-7A shows also the cost for multipliers using only small distributed RAMs. It can be seen a rapid grow of the cost for the width equal 9, 13, 17, 21, ... when the width surpasses the number divided by 4.

In this thesis to find an optimal solution of a multiplier, an exhaustive 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 illustrate considered architectures, an example of the LM for input and coefficient width equal 14 is shown in Figure 3-8. In this example a combination of BSRs and distributed RAMs is implemented. 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 the MS and DAWR optimisations have not been considered.

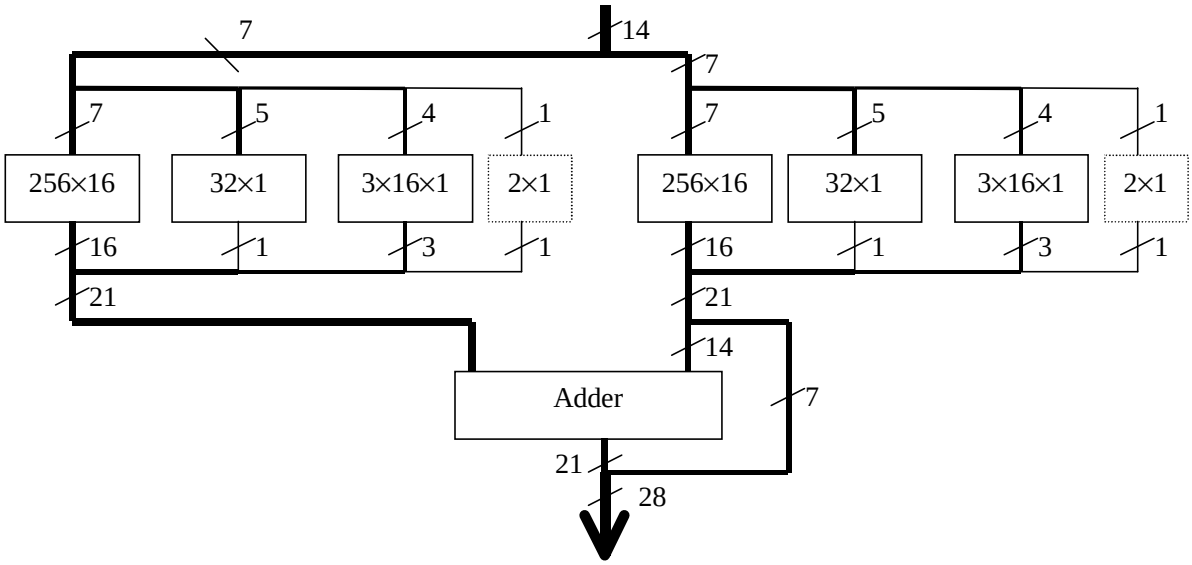

Figure 3-8. A LM for input and coefficient width equal 14

It should be noted that the AuToCon does not assume any initial relations between memory modules and adders' costs, therefore memory modules can be freely selected and the program can generate circuits for any FPGA family or even for ASICs. The input parameters to the program are adders and memories sizes and costs.

In this thesis 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 [Alt99] 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.

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, only 256×16 RAMs have been used. Even 256×16 memory modules are very seldom fully exploited. For example in Figure 3-8, one address line is not

used, which causes that only half of the memory is used. Therefore a question arises what optimal memory size is. A general answer is that memory data width should satisfy:

$$W_D = W_C + W_A - W_L \qquad (3\text{-}7)$$

where: W_D *– memory data width,* W_C *– width of the coefficient,* W_A*- memory address width,* W_L*- address width of smaller and less costy LUTs,* $W_L = 5$ *for Virtex (32×1).*

From eq. 3-7 it can be seen that for $W_C = 13$ and $W_A = 8$ the result is $W_D = 16$ which corresponds with the 256×16 memory module, therefore the maximum of the equivalent BSR cost is observed in Figure 3-7 for the input width equal 13.

It should be also noted that the number of used BSRs depends on the cost relations. For example, for a 16-bit wide multiplier, the number of BSRs gradually increases as the BSR cost decreases, see Table 3-4.

Cost BSR VCLBs	# BSRs	LUT RAMs Cost VCLBs	Adders Cost VCLBs	BSR eq. Cost VCLBs
≥7.75	0	19	16	-
≥6.5	2	9.5	10	7.75
≤6.25	4	0	6	7.25

Table 3-4. The BSRs cost and its influence on the best architecture; area in Vertex CLBs, 1VCLB= 4LEs

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.
- The 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 format. However the MSB LUT output can be either positive or negative therefore design optimisation cannot be implemented.
- The negative coefficient and positive input. All LUTs operate on two's complement numbers, but it can be seen that the LUT outputs are always negative. Therefore additions can be substituted by subtractions and in this way all LUTs will operate only on positive data. In consequence, the LUT sign bit coding is skipped and therefore the width of each LUT is one bit shorter. However the double subtraction ($s=-a-b$) cannot be implemented

in 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 substations for all but the LSB LUT. This implies that double subtraction chain is broken, and the copy of LSBs (see Section 3.1.1) is achieved.

- The negative coefficient, two's complement input data. In this case all but the MSBs LUT, operate on only negative numbers and should be implemented as in the previous region. The MSBs LUT operates on either positive or negative numbers, therefore might 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 with a subtraction therefore the circuit is implemented in the same way as in the previous region.

3.3. Comparison of the multipliers

3.3.1. Area

In this chapter 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 3-9. 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 3-9 shows only statistical relationship. Therefore both architectures should be considered and the best of them chosen. However, from Figure 3-9 it can be seen that the gain from considering best of the LM and MM is insignificant for K greater than 5.

The general conclusion can be drawn from Figure 3-9. 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 gain is on average 5÷20% depending on the input width K.

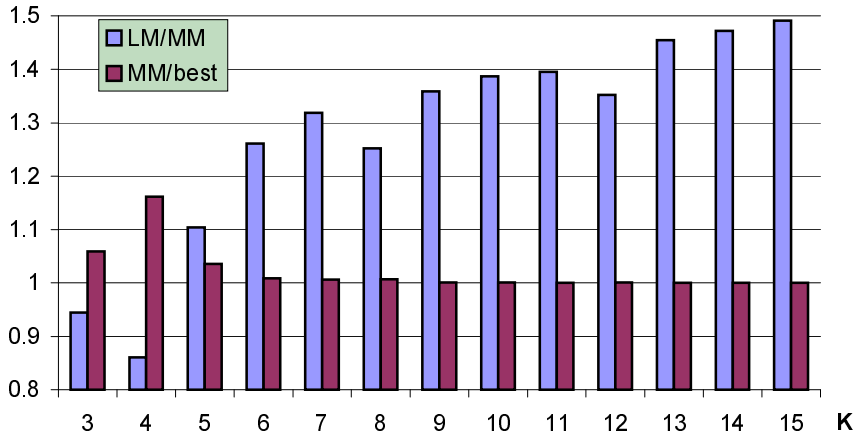

Figure 3-9. Relation between average area of XC4000 occupied by: LM/MM – using only LM and only MM; MM/best – using only MM and the best of LM and MM. Results for the different input width K (input range $0 \div 2^K-1$) and coefficient values $1 \div 2^K-1$

3.3.2. 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 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 flip-flops have to be inserted without associated logic (according to cut-set method [Pir98]). 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 [Pir98] 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.

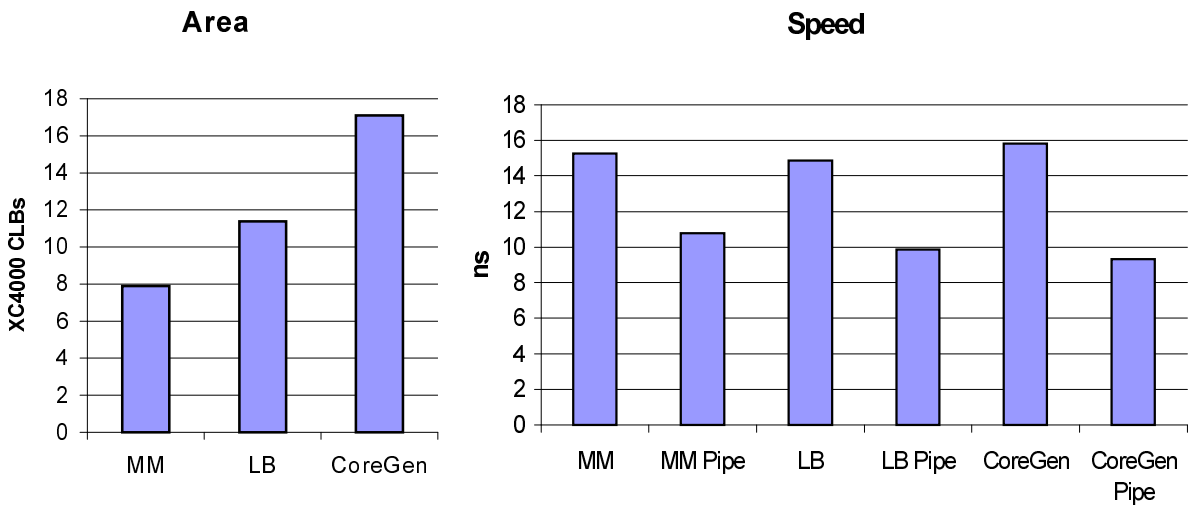

Figure 3-10. Average area without pipelining and system period without and with pipelining for the MM, LM and Core Generator [Xil99] multiplier. Implementation results for XC4000E-1 and for the 8-bit unsigned input and randomly chosen coefficients equal: 41, 108, 132, 190, 225

Figure 3-10 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 [Xil99a] – a commercial program.

3.4. Conclusions

This chapter 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 width increase, the MM is getting more and more attractive due to greater efficiency of the CSD and SS methods.

Furthermore, an improved algorithm for conversion from the two's complement to the CSD representation is 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 achieved for the subtraction. Consequently, a subtraction (CSD representation $\bar{1}$) is implemented only if the total number of operations decreases.

This chapter thoroughly studies the LM and presents different optimisation techniques which are rather intuitive but have never been presented. Firstly, the combination of different memory modules has been introduced. This aspect is very important as FPGAs incorporate

different memories and therefore finding the optimal memory configuration is a complex task that is tackled hereby. Furthermore, different optimisation methods are presented for the LM: LSB Address Width Reduction (LAWR), Don’t care Address Width Reduction (DAWR), Memory Sharing (MS) and negative number optimisation techniques. At the end, the cost/speed relationship between presented architectures are presented for Xilinx Virtex or XC4000 family.

4. Architectures of Multipliers

Bit-parallel multiplication can be carried out implementing three different methodologies. The first is a variable coefficient (fully functional) multiplier (VCM) which can be implemented using for example parallel-array multipliers [Omo94] or Wallace tree multipliers [Wal64]. For the VCM, a coefficient value can be freely changed but the disadvantage of this solution is a relatively high cost. The alternative solution is a Constant Coefficient Multiplier (KCM) which in comparison to the VCM has much lower hardware requirements [Cha96, Pet95], and therefore is recommended provided that the coefficient is constant during a calculation process. For ASIC designs the coefficient value once determined cannot be changed. Conversely for FPGAs, the change of a coefficient value can be implemented by reconfiguring the FPGA structure. The process of reconfiguration usually takes several ms [Xil99b]; therefore if the calculation process can be paused for that time and the coefficient is relatively constant during data processing [Wir97], the KCM solution should be considered. The reconfiguration time can be however reduced by the use of a partially reconfigurable FPGA, e.g. a Virtex FPGA [Xil99b]. The KCM solution has another drawback that the multiplier circuit has to be redesigned for a different coefficient value. Fortunately, by the use of an automated tool (e.g. AuToCon), the KCM can be redesigned within the time of seconds. However a new design has to re-employ a place and route program which fits the new design into the FPGA. The fitting process is usually time-consuming and takes approximately 1min ÷ 1hour. In conclusion, the change of a coefficient value for the KCM requires not only the FPGA to be reconfigured but also the whole design cycle to be re-implied. This causes that the change of a coefficient value for the KCM solution is onerous and therefore often the more-hardware-consuming VCM solution taken instead.

An alternative solution is a Dynamic Constant Coefficient Multiplier (DKCM). The DKCM is a Look up table based Multiplication (LM) for which the change of the coefficient can be achieved by a proper change of LUT memory contents. This solution can implement in-circuit coefficient reconfiguration therefore the multiplier configuration time is shorter and the design fitting into the FPGA need not be re-implied. A drawback of this solution is that the DKCM occupies more area in comparison with the KCM.

At the first part of this chapter the LUT based multiplication (LM) and its modification – the DKCM is presented. For the DKCM three different options: multiplexing in logic, multiplexing in tri-state buffers and dual port memories are studied. Then, a comparison of the KCM, DKCM and VCM and their implementation results are given.

4.1. Dynamic Constant Coefficient Multiplier (DKCM)

The DKCM [Xil99] (or self-configurable binary multiplier [Woj98, Woj99]) is the LUT based multiplier for which ROMs are replaced by RAMs. The idea behind the dynamic change of a coefficient value is to properly change the contents of the memories. This, however, requires an additional RAM programming interface and imposes constrains on the DKCM architecture in comparison to the KCM.

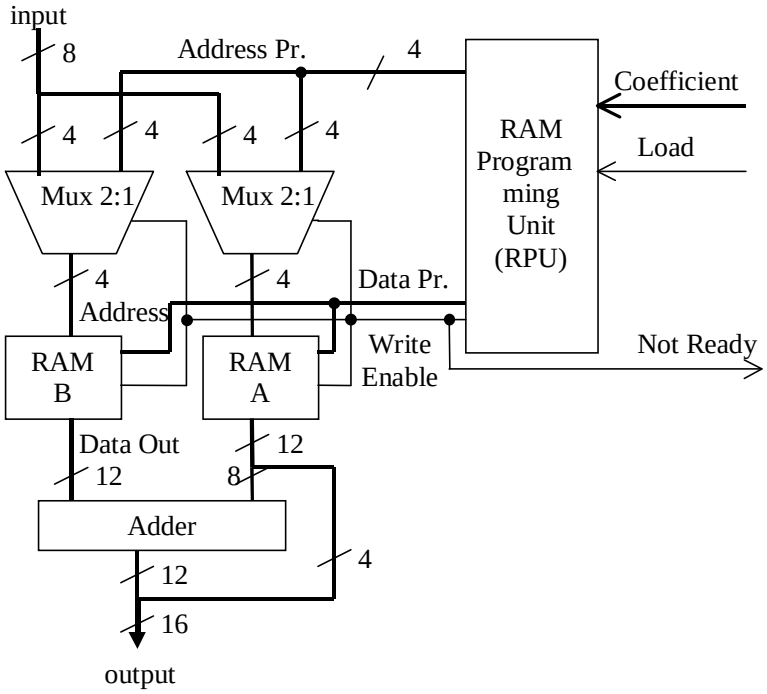

Figure 4-1. An example of the DKCM for input data and coefficient width equal 8

The additional RAM programming interface can be divided into two parts. The first part allows the RAMs to be programmed and usually consists of address and (rather seldom) data multiplexers. The second part of the additional circuit is RAM Programming Unit (RPU) which produces proper data and address sequences and control signals for RAM programming. An example of the DKCM is shown in Figure 4-1. It should be noted that this example is equivalent to the KCM given in Figure 3-5.

4.2. Memory Multiplexers

It can be seen from Figure 4-1 that RAM memories usually have separated paths for data reads and data writes [Xil99b], therefore data multiplexing is not required. Unfortunately, the address bus is the same for reads and writes to the RAMs, therefore additional multiplexers for switching between these addresses have to be implemented. The multiplexing process can be carried out using Logic Elements (LE) in Configurable Logic Blocks (CLBs) or tri-state buffers (TSBs) [Xil99b]. The latest solution consumes no logic area, though uses the programmable interconnect resources which are often limited and slower than multiplexing in LEs. The multiplexing process can be skipped by the use of dual-port (DP) RAMs. The DP-RAM solution is usually quicker (without multiplexers delay) but consumes more area. For example, for Virtex, a 16×1DP distributed RAM consumes the area of two corresponding single-port (SP) RAMs. However, Virtex incorporates a large 4kb Block SelectRAM (BSR) DP RAM therefore DP-RAM can be employed without any hardware overheads. Summing up, design optimisation should consider three different options:

- Multiplexing using logic (LEs) resources (denoted as DKCM-L)
- Multiplexing using programmable interconnect (TSB) resources (DKCM-T)
- Using dual-port RAMs (DKCM-D).

4.3. RAM programming unit (RPU)

The main task of the RPU is to provide the memory with write address and data. Let consider, at first, the case when input data is always positive and all memory modules have the same address width. In this case during programming, all memories are fed with the same address and data (like in Figure 4-1), therefore the RPU consists of address counter and the accumulator which starts from value zero and is incremented every clock cycle by the coefficient value [Woj98]. In consequence, the data sequence is as follows:

$$d_0 = 0$$
$$d_1 = d_0 + coeff = coeff$$
$$d_2 = d_1 + coeff = 2 \cdot coeff \qquad (4\text{-}1)$$
$$\ldots$$

where d_i- write data for address value i, coeff- the coefficient value

It should be noted that the number of memory writes (the number of the multiplier idle clock cycles) depends on the memory size, e.g. for RAM 16×1, sixteen memory writes are required. Therefore in some applications, it may be beneficial to use only a part of memory in order to

reduce the multiplier idle time. However, this causes that the multiplier consumes more hardware [Woj98].

The RPU becomes more complicated if memory sizes (address widths) are different because either different memory modules have been implemented or the input data width can not be evenly distributed into separate memories. In this case, each memory write-enable signal should be disasserted whenever the write address overflows the memory address width. This however may require additional write-enable logic to be implemented. The write-enable problem can be solved by programming RAMs from the highest address (all ones) down to zero. In this solution, all memories can be written disregarding address width because the latest memory writes are always proper and overwrite the previous (improper) writes. The data sequence for programming RAMs is therefore as follows:

$$d_{s-1} = (coeff << w) - coeff = (s-1) \cdot coeff;$$

$$d_{s-2} = d_{s-1} - coeff = (s-2) \cdot coeff$$

....

$$d_0 = d_1 - coeff = 0 \qquad (4\text{-}2)$$

where: coeff<<w - the coefficient shifted w bits to the left, w- maximum width of memory address, s- maximum size of memory $s = 2^w$.

The drawback of the above solution is that an multiplexer 2:1 is required (instead of the reset circuit for eq. 4-1) for feeding the subtractor either with *(coeff<<w)* or d_{i+1}.

The RPU is further complicated for negative (two's complement) inputs. In this case all RAMs except from the MSBs RAM, operate on positive inputs therefore can be programmed as above. For the MSB RAM and for the MSB (sign bit) equal zero, the MSB RAM is programmed as the rest of RAMs. Conversely, if the sign bit is asserted then the RAM has to be programmed with a different data sequence which can be generated by continuing eq. 4-2, as follows:

...

$$d_0 = d_1 - coeff = 0$$

$$d_{-1} = d_0 - coeff = -coeff$$

$$d_{-2} = d_{-1} - coeff = -2 \cdot coeff \qquad (4\text{-}3)$$

...

$$d_{-sn} = d_{-sn+1} - coeff = -sn \cdot coeff$$

where: sn- the size of the MSB RAM divided by 2.

It should be noted that eq. 4-3 does not require additional hardware, it uses the same address counter and subtractor as eq. 4-2. However, programming two's complement input multiplier requires additional control logic for write-enable signals. Consequently, the MSB

RAM write-enable is asserted during whole programming process; for the rest of the RAMs, the write-enable signal is asserted only for eq. 4-2 and disasserted for the rest of eq. 4-3. It should be noted that the two's complement input format causes that the multiplier programming (idle) time is longer.

4.4. Implementation results for the DKCM

The optimal architecture of the DKCM depends strongly on a given FPGA device; therefore at first implementation results for Xilinx XC4000 family [Xil99b] will be studied. The multiplication requires mainly 2:1 multiplexing, addition and RAM units, therefore only these modules will be considered here. The XC4000 incorporates single-port (SP) 16×1 and 32×1 and dual-port (DP) 16×1 distributed RAMs at the cost of 1, 2 and 2 Logic Elements (1LE ≈ 4-input LUT ≈ ½ XC4000 CLB) respectable, and an adder with dedicated ripple carry logic at the cost of 1 LE/bit. A 2:1 multiplexer consumes 1 LE if implemented in logic, or only programmable interconnects resources if implemented as a tri-state buffer.

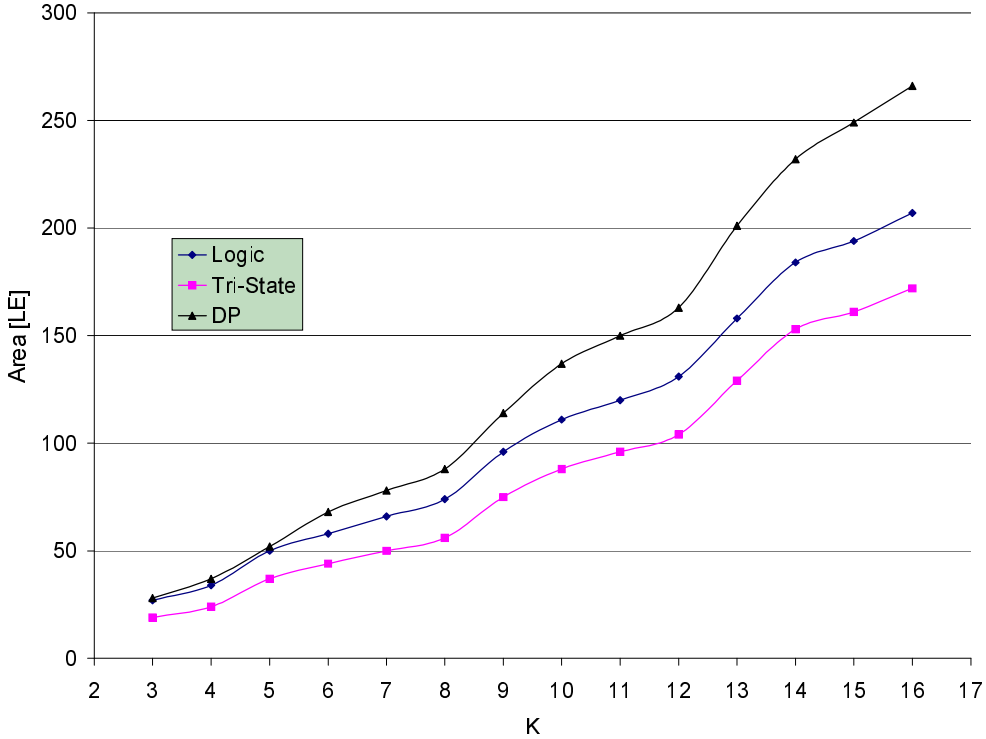

Figure 4-2. Area occupied by the DKCM for different input and maximum coefficient widths K. Implementation for XC4000 and unsigned coefficients and inputs

It can be also seen from Figure 4-2 that the best solution seems to be the DKCM-T for which multiplexers are implemented as tri-state buffers. However the drawback of the DKCM-T is that tri-state-buffer multiplexers are usually slower than logic multiplexers are

(see Figure 4-3). This may cause that commonly used A·T product [Sei84] is worse than for other multipliers. It should be noted from Figure 4-3 that the tri-state-buffers propagation delay is even less acceptable for pipeline architectures. Conversely, in order to speed up the multiplier the DP memories should be used. The DP RAM solution has however a drawback, the cost of DP RAM is twice of the SP RAM and the speed of the circuit is improved only for non-pipelined architectures. However, if the cost of SP and DP RAMs is the same the DP RAMs should be taken.

There is also another virtual DP 2×1 RAM which can be implemented as a 2-input AND gate ($a_0 \cdot b_0$). The cost of this module is 1 LE which is lower than 2 LE for 16×1 DP RAMs, this memory module should be therefore implemented to calculate every LSB of the LUT (for which address width is 1). Similarly, a virtual 4×1DP memory can be implemented using a 16×1 LUT ($a_0 \cdot b_1 \oplus a_1 \cdot b_0$) to calculate the next bit to the LSB of every LUT. Furthermore, the parallel-array multiplier [Omo94] can be obtained if only AND gates instead of RAMs are implemented.

Up to now, XC4000 family, which incorporates only small distributed RAMs, has been considered, but additional RAM resources are available in Virtex which incorporates large DP 4k×1, 2k×2, 1k×4, 512×8 and 256×16 BlockSelect RAMs (BSRs).

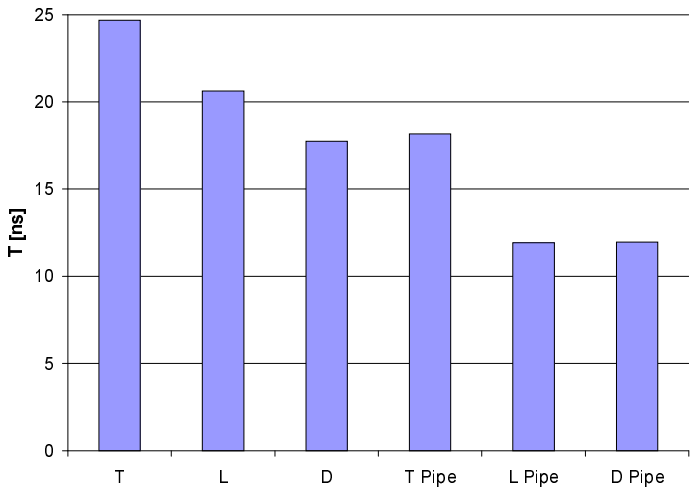

Figure 4-3. The maximum propagation delay for different architectures of the DPCM without and with pipelining. Results for XC4003E-1 and K=8

Constructing the optimal multiplier using large BSRs, distributed RAMs and adders is however a difficult task which involves many trade-offs:

- Cost relations between BSRs, distributed (small) SP and DP RAMs, multiplexers and adders. The chip area occupied by 1 BSR is equivalent to roughly 64 LEs, but the real cost-relation is application- and resources-dependent, as free BSRs can be implemented instead of fully used logic elements and vice-versa.
- The multiplier programming time is proportional to the memory size, therefore in applications where operation idle time is a critical factor, smaller memory blocks are preferable.
- Multiplication delay time tends to be lower with larger memory blocks as the number of arithmetic blocks decreases. Conversely, the memory access time usually increases with the memory size, and routing large RAMs with arithmetic modules is more difficult as the BSRs have fixed position in FPGAs and cannot be freely mixed with adders as it is the case for small distributed memories. The case is even more complicated for pipelined architectures where a cost of additional flip-flops and a frequency of the system clock have to be considered.

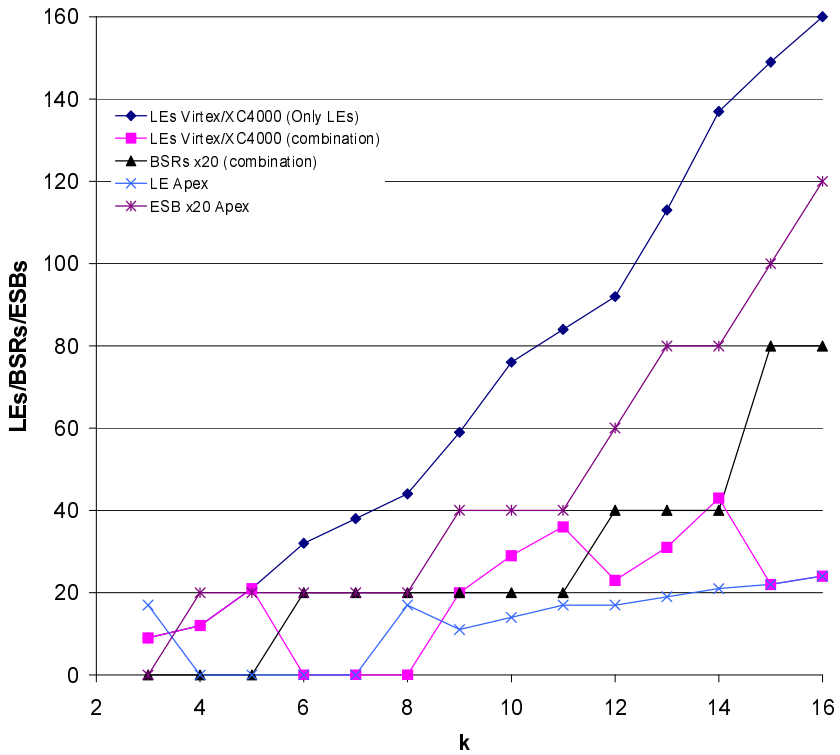

Figure 4-4. Number of LEs and BSRs (scaled 20:1) for Virtex (using large BSR), and number of LEs for XC4000/Virtex (using only small distributed RAMs), and number of LEs and ESBs (scaled 20:1) for Apex. The RPU is not considered, the equivalent costs: 1 BSR = 1 ESB= 20 LEs

The implementation results for combination of large BSRs and small distributed RAMs, and for only small distributed RAMs is shown in Figure 4-4. Note that the number of used

BSRs depends on the equivalent cost of the BSR; and for the equivalent cost greater than roughly 44 LEs- the BSRs are not used at all.

Altera Apex 20K [Alt99] family also incorporates dedicated ripple carry logic at the cost of roughly 1 LE/bit; but in comparison with the Xilinx FPGAs can implement only large DP RAMs: 2k×1, 1k×2, 512×4, 256×8 or 128×16, one in each Embedded System Block (ESB). Consequently, as it can be seen in Figure 4-4, the number of required ESBs in comparison with BSRs is greater, however the number of LEs is reduced. The next consequence of the lack of distributed RAMs in Apex FPGAs is the longer coefficient reprogramming time in comparison with the Xilinx FPGAs when only distributed RAMs are used.

The AuToCon uses advance full search algorithm which generates the best solution from the given input parameters: input data range, coefficient range and cost relations between adders, memories, multiplexers and flip-flops, etc. In order to illustrate architectures analysed by the AuToCon, an example of the optimum structure of the multiplier for K= 12 is given in Figure 4-5. Note that in this example only dual port memories are implemented therefore the input multiplexing is not required. However it might seem that it is better to use multiplexers instead of DP RAMs according to the results in Figure 4-2, nevertheless in Figure 4-5 only one 16×1DP RAM is needed, therefore the use of multiplexers cannot be justified.

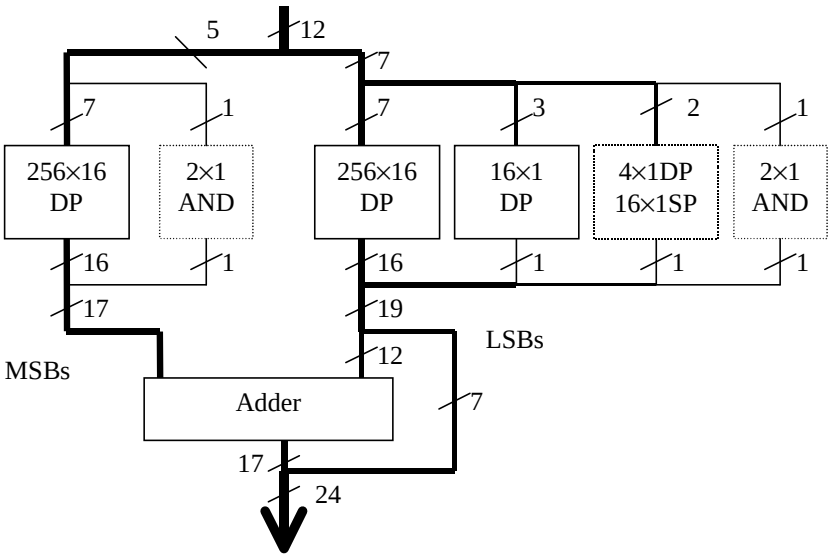

Figure 4-5. An example of the multiplier for input and coefficient range 0÷2^{12}-1. The optimal architecture for Virtex FPGAs and equivalent cost 1BSR= 20LEs = 5 CLBs. The RPU is not shown

4.5. Implementation of the DKCM versus the KCM

Initially, the architecture of the DKCM does not seem to be much different in comparison with the KCM, only the additional RAM Programming Unit (RPU) and address multiplexing are required (see Figure 3-5 vs. Figure 4-1). However, the KCM can be implemented using either the LM or MM (see Chapter 3). The MM is getting more and more attractive as the coefficient width increases because more efficient optimisation techniques such as Canonic Sign Digit CSD and / or Sub-structure Sharing (SS) are employed. Consequently, for Xilinx XC4000, and input and coefficient width greater than 5, the LM consumes on average 25÷50% more area in comparison with the corresponding MM, as it is shown in Figure 3-9.

Furthermore, even the LM can employ advance optimisation techniques which are suitable only for the KCM. These techniques are enumerated below:

- Simplification for even coefficients - in this case the LSB(s) of the product has (have) a fixed zero value. Furthermore, for coefficients: 1, 2, 4, 8, etc., the KCM can be replaced by the argument hardwired shift, which does not consume any hardware.
- Reduction of memory size –Don't care Address Width Reduction (DAWR), see Section 3.2.1.
- Memory sharing (MS)
- Skipping coding of the sign bit – for fixed coefficient, LUT outputs are usually either always positive or negative, and therefore sign bit need not be coded see Section 3.2.2.

The DKCM in comparison with the KCM can implement a great range of coefficient values, for which, conversely, different KCMs should be developed. Furthermore, a KCM architecture varies significantly for different coefficients, which causes a great difference in area occupied by the KCM. Therefore, to compare the DKCM with the corresponding KCMs, three different statistical costs of the KCM can be used:

1. Average area occupied by a KCM for a given coefficient range (usually $1 \div 2^K-1$). This cost is suitable for static configurable systems [San99], for which the cost of a static KCM and its equivalent static DKCM is compared. The average area of the KCM can also be used for dynamic configurable systems [San99] for which a great number of KCMs are considered at the time, therefore usage average statistical value is justified.
2. Maximum area for a given coefficient range – is recommended for dynamic configurable systems, for which the coefficient is changed by FPGA reconfiguration.

3. Maximum area for a given coefficient set - as in point 2, but in the case when the number of possible coefficients is relatively small. This value seems the best for defined designs, however may constrain further design changes. This solution however cannot be generalised and therefore is not further referred to.

The comparison of the KCM and DKCM is given in Figure 4-6. For small values of *K*, area occupied by the DKCM is much greater than for the KCMs due to the strong influence of the RPU; on Figure 4-6 the cost of the RPU is illustrated as the difference between DKCM-T and maximum cost of the KCM-LM. As *K* increases, the relative cost of the RPU decreases (for *K= 3* and *K=16*, the RPU occupies 63% and 23% of the whole DKCM-L area respectively), and additional cost of the DKCM over the KCM is related rather to the comparison strategy (the average or maximum cost of the KCM). For example, for *K*= 15 the DKCM overhead is 39% and 136% for the maximum and average KCM area respectively. It should be also noted that architectural (LM vs. MM) overhead increases with growing *K* as it is shown in Figure 4-6 (max KCM-LM vs. max KCM).

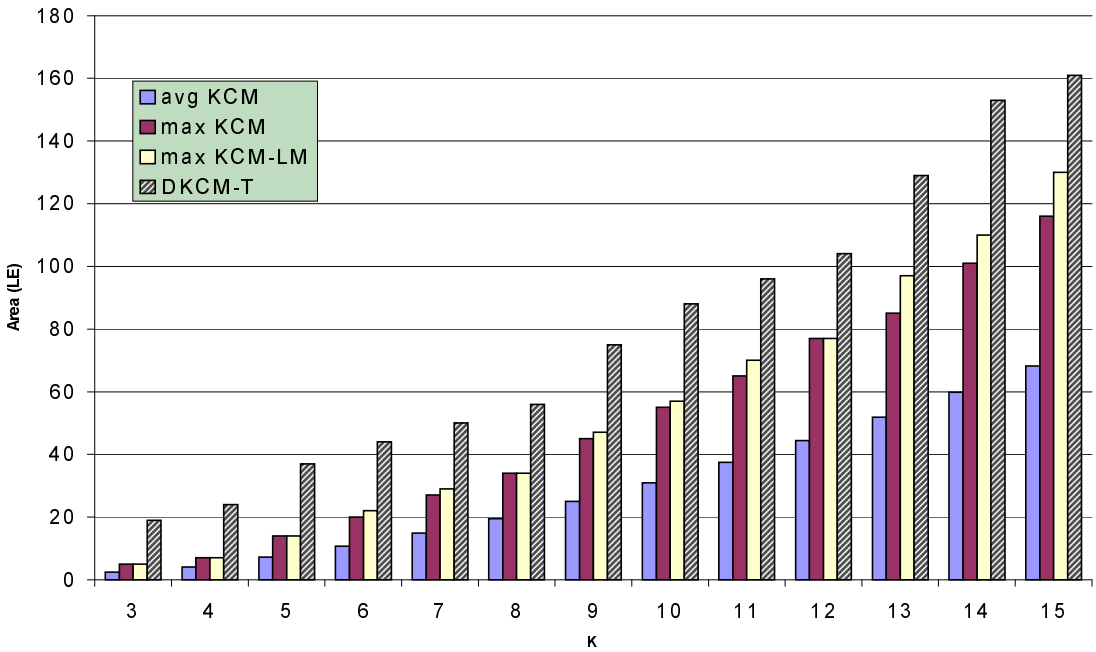

Figure 4-6. Area for Xilinx XC4000, occupied by: the DKCM-T, maximum area of the KCM-LM and KCM (the best architecture of the MM or LM) and average area for the KCM. The input range $0 \div 2^K-1$ and the coefficient range $1 \div 2^K-1$

4.6. Implementation of the DKCM versus the VCM

The VCM is a fully functional multiplier, usually implemented using AND-gates and adders [Omo94, Wal64], for which a coefficient-change penalty is not observed. The

drawback of the VCM, as can be seen from Figure 4-7, is its large cost in comparison with the DKCM. For small multiplier width *K*, however, the cost of the DKCM is dominated by the RPU, therefore the VCM is recommended.

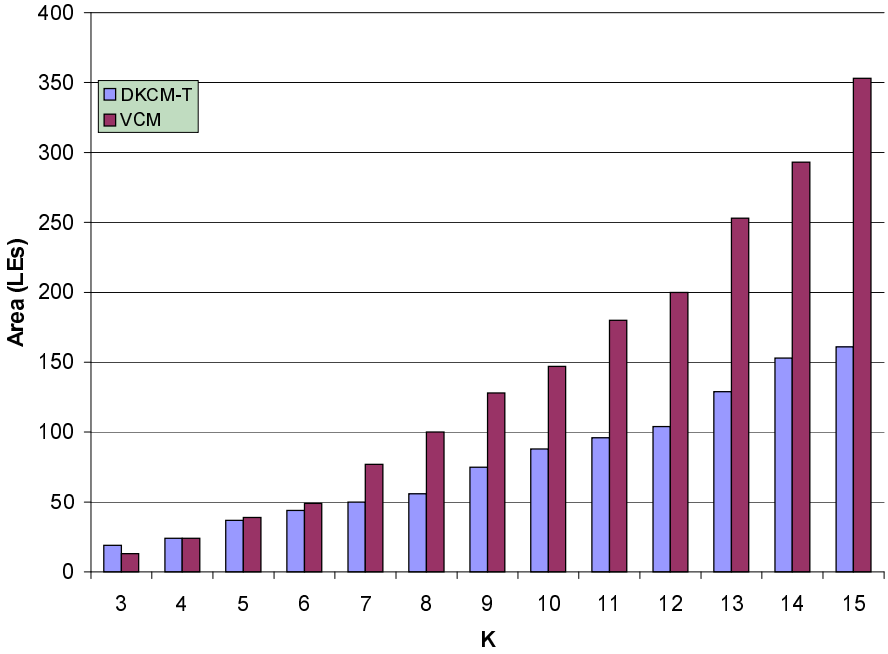

Figure 4-7. Area (Xilinx XC4000) for the DKCM-T and VCM for different K

According to Figure 4-7, the DKCM should be implemented for $K \geq 7$. Nevertheless Figure 4-7 presents the best results for the DKCM as in real applications the DKCM requires RAMs programming (idle) cycles which decrease the design throughput and may require design modifications (additional cost). In consequence, two different groups of application can be distinguished:

A) **Designs without reconfiguration overheads** – the change of the coefficient occurs very seldom and / or does not disrupt the system work. For example, a real time image processing system for which a change of the coefficient is carried out during image blank time, or so seldom that corrupted data is invisible. In this case, the DKCM can be implemented without additional overheads. In this case, the KCM and a (dynamic) reconfiguration system instead of DKCM should be also considered to allow additional savings.

It should be noted that for the DKCM-D (dual port DKCM) and for adaptive systems where the difference between the present and new coefficient is very slight, the product obtained while RAM programming is usually only slightly corrupted. Furthermore, for positive inputs (or sign and magnitude input data format) and RAM programming schedule

according to eq. 4-1, the multiplication result is in the range of the results calculated for the old and new coefficient. This is proved below for multiplication $M= A \cdot B$. To simplify the prove, multiplier consists of only two LUTs, however the prove can be easily extended for a greater number of LUTs.

The multiplication result is given as:

$$M= (B \cdot A_{MSB}) << k + B \cdot A_{LSB} \quad (4\text{-}4)$$

where: B- multiplication coefficient, A_{MSB}- MSBs part of the input, A_{LSB}- LSBs part of the input, k- address width of the LSBs LUT

During the LUTs programming, each address location of the LUTs stores either new $B_N \cdot A_{MSB}$, ($B_N \cdot A_{LSB}$ for the LSB LUT) or old $B_O \cdot A_{MSB}$ ($B_O \cdot A_{LSB}$) value depending if the address location selected by the input data has or has not been programmed. Therefore the result of the multiplication can be as follows:

$$M_O= (B_O \cdot A_{MSB}) << k + B_O \cdot A_{LSB} \quad (4\text{-}5)$$

$$M_{C0}= (B_O \cdot A_{MSB}) << k + B_N \cdot A_{LSB} \quad (4\text{-}6)$$

$$M_{C1}= (B_N \cdot A_{MSB}) << k + B_O \cdot A_{LSB} \quad (4\text{-}7)$$

$$M_N= (B_N \cdot A_{MSB}) << k + B_N \cdot A_{LSB} \quad (4\text{-}8)$$

In eq. 4-5 the multiplier calculates a proper value for the old coefficient. In eq. 4-8 a proper new result is calculated. Only eq. 4-6 and 4.7 give corrupted results, however it can be seen that for $A_{LSB} \geq 0$ (always true) and $A_{MSB} \geq 0$ (holds for positive inputs) the M_{C0} and M_{C1} are in the range of M_O and M_N (Consider two cases: $B_O < B_N$, which gives $M_O \leq M_C \leq M_N$, and $B_O > B_N$, which gives $M_N \leq M_C \leq M_O$).

Eqs. 4-5 ÷ 4-8 hold provided that a simultaneous read and write to the same DP RAM address location is allowed and well defined. However, this is usually the case, e.g. for Xilinx XC4000 16×1 DP.

B) **Designs with reconfiguration overheads** - the coefficient changes frequently or its change interferes with the system work. In this case, four different approaches can be implemented:

- DKCM-P - two parallel RAMs sets and additional multiplexers are used [Woj98], which allows a RAMs set to be programmed while another is operating and vice-versa.
- DKCM-D - as described in the point A (output data may be slightly corrupted!), but architectural overhead is considered as the DP-RAM solution is usually less hardware-efficient than the SP-RAM counterpart.

- VCM, which has no coefficient change penalty.
- DKCM - the multiplier for which multiplication process is stopped whenever RAM is programmed. In some cases, however, this solution cannot be implemented, as the multiplier cannot stop its operation without heavy influence on the other units.

To qualify the benefits from using the DKCM reconfiguration approach, let define a functional density *D* [Wir97, Wir98]:

$$D = \frac{1}{A \cdot T} \tag{4-9}$$

The functional density for the DKCM should consider the idle time needed to program the LUTs:

$$D_D = \frac{1}{A \cdot T \cdot (1 + \frac{r}{n})} \tag{4-10}$$

where: D, A, T – functional density, area and critical delay respectively; r- number of reconfiguration cycles; n- number of execution cycles between two consecutive reconfigurations.

For the DKCM, a reconfiguration penalty factor *r/n* has been introduced. The penalty can be decreased either by the increase of *n* - the number of execution cycles between two consequtive reconfigurations; or by a decrease of the number of reconfiguration cycles *r*.

It can be seen from Figure 4-8 and for the DKCM-T8, for which the number of reconfiguration cycles *r* is reduced from *r*= 16 to *r*= 8 (only half of the 16×1 RAMs is used), that the decrease of *r* causes only slight increase of system performance for small *n.* For small *r*, memory blocks are not fully used and therefore the multiplier occupies more area and has longer propagation time. In consequence, for small *n*, the VCM should be rather implemented. An alternative solution is the DKCM-D for *n*>16, for which the coefficient change penalty is not observed, provided that coefficient changes are very slight and the product can be slightly corrupted. For the DKCM-P, the additional RAM set and multiplexers increase the design area and propagation time, which causes that this multiplier is usually not recommended for XC4000.

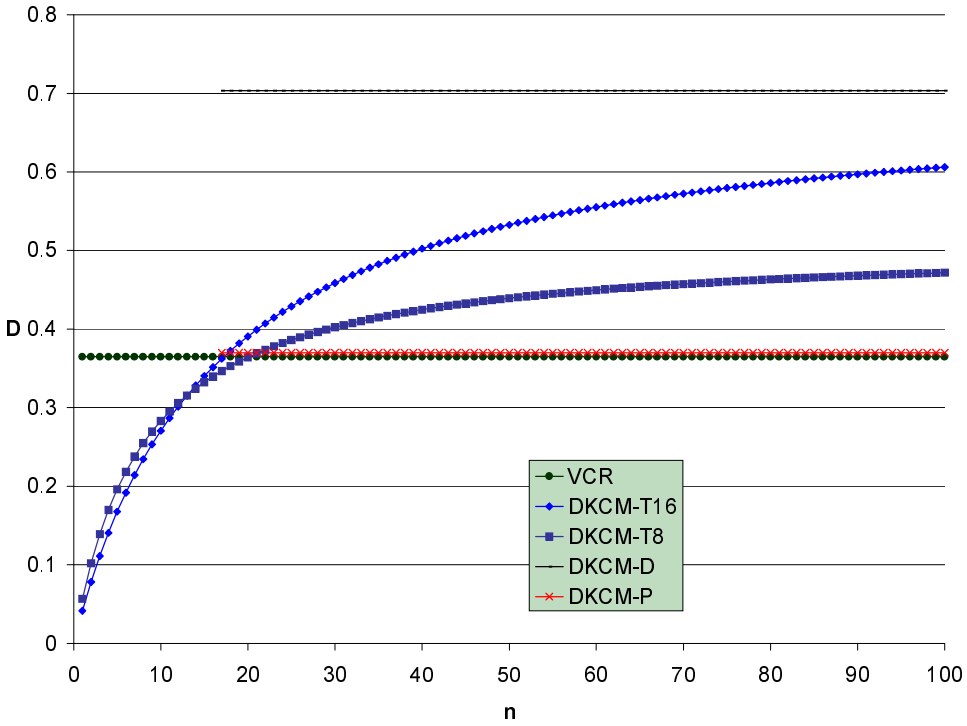

Figure 4-8. Functional density D [1/LE·µs] as a function of the number of n, results for K= 8 for XC4003E-1 (XC4005E for the DCKM-P), for following options (enumerating from the top for n=100): DKCM-D, DKCM-T16, DKCM-T8, DKCM-P (TSB option), VCM

4.7. Conclusions

A proper choice of the multiplier architecture in FPGAs is, as it is shown in this chapter, a difficult task. For ASICs, two choices: the VCM and KCM can be easily distinguished. However, for FPGAs the boarder between these two solutions cannot be smoothly defined as FPGAs can be quickly reconfigured. Therefore, implementation of the KCM instead of the VCM is strongly recommended as the KCM occupies 17÷23% on average or 29÷41% on maximum, area of the VCM for multiplier width K= 3÷15. Furthermore, lower area of the design causes usually shorter propagation time, and consequently a significant increase in design functionality *D*. Conversely, coefficient change for the KCM has a penalty of operation idle time which decreases design functionality according to eq. 4-10. Consequently, to decrease the idle, reconfiguration time there is a tendency to use a partial reconfiguration for which only the multiplier circuit is reconfigured.

The reconfiguration time can be further decreased by in-circuit reconfiguration, i.e. by the use of the DKCM. The DKCM offers much quicker reconfiguration but occupies more area in comparison with the KCM, therefore is a middle-way solution between the VCM and

KCM. The DKCM requires additional RAM Programming Unit (RPU) which significantly influences the DKCM cost for small multiplier widths.

For adaptive signal processing, the DKCM solution is even more attractive as the DKCM-D has no reconfiguration penalty provided that the product can be slightly corrupted and the number of execution cycles is greater than memory size. Furthermore, the process of coefficient change for the KCM requires not only FPGA reconfiguration but also redesigning and re-routing of the KCM which consumes significant amount of time (about 1min÷1hour) and therefore this solution is usually unacceptable for adaptive systems.

5. Convolution in FPGAs

The convolution basically consists of sum of (delayed) products, therefore multiplication is an essential operation. Consequently, each multiplier can be implemented separately, and then the addition applied. Nevertheless, disregarding the multiplier entities allows for further optimisations. For example, for the LM, instead of considering separately additions within the multipliers and then the final addition, a single adders block can be formed, which allows for better grouping the adders, and therefore for implementing a more hardware-efficient circuit. This design approach to the group of the LMs is further denoted as LUT based Convolution (LC).

In addition, a (parallel) Distributed Arithmetic Convoler (DAC) – a completely different architectural solution can be implemented. This solution is similar to the LM, nevertheless, the order of multiplications and additions is disregarded, which allows for memory data width reduction, in comparison to the LC. This chapter presents also a novel approach: an Irregular Distributed Arithmetic Convoler (IDAC) which is a combination of the DAC and LC.

Unlike for multiplierless multiplication (MM), for convolution common substructure is not considered separately within each multiplier, but substructure sharing is applied for all coefficient altogether. Therefore a term, Multiplierless Convolution (MC), instead of the MM, is introduced. This causes that trading-off between the (LUT based) IDAC versus the MC is more complex than it is the case for the multiplication and the LM vs. MM. Consequently a sophisticated algorithm has been developed to confront the problem.

For convolution interdependence between coefficients is often very strong, as symmetric filters are often implemented. Consequently, additional algorithm for automatic detecting and grouping similar coefficients is also implemented.

5.1. Previous Works

For ASICs, several FIR filter silicon compilers [Jai91, Las92, Haw96] have been developed. Nevertheless, FPGA designs differ significantly from the ASICs, as FPGAs usually incorporate dedicated ripple-carry logic, which makes that the different adders approach is adopted. Furthermore, FPGAs implement logic employing Look-Up Tables and therefore LUT-based multiplication or convolution is an alternative solution to be taken into account.

An automatic implementation of FIR filters on FPGAs has been presented in [Moh95]. This tool employs inverted form 1D FIR filters [Moh95] and techniques adopted for ASICs, such as power-of-two coefficient space [Lim83] (denoted hereby as the multiplierless multiplication), carry-save adders [Omo94] for XC3000 family [Xil93] (XC3000 family does not incorporate dedicated ripple-carry logic) and dedicated ripple-carry adders for XC4000 [Eva94]. Up to the author's knowledge, the SS has not been implemented in [Moh95], only CSD representation is used. Employing inverted form FIR filters instead of direct form filters (implemented by the AuToCon) excludes implementation of distributed arithmetic. Furthermore, adders operate on wider arguments in comparison to the direct-form filters. Besides inverted form filters are not recommended for 2D filters as wider line buffers are required. Nevertheless, a 2D filter can be constructed from several inverted form 1D filters. Conversely, the structure of inverted form filters is more modular. Furthermore, a lot of design effort in [Moh95] has been put into mapping (optimising placment and routing) to increment clock frequency. Pipelining is (somehow) built-in the structure of the inverted form filters, therefore, in comparison to the pipeline architecture of the direct-form filter, it might seem that less flip-flops are required. However inverted-form filters require wider pipelining registers. Summing up, direct-form filters allow more architectural solutions to be adopted, and more design parameters to be specified in comparison to the inverted-form filters. Therefore, the direct-form solution has been adopted in the AuToCon, nevertheless more thorough research is required to compare these two different architectural solutions. Besides direct form filters can be employed as a sum of products, etc.

Core Generator [Xil99], program distributed by Xilinx Inc., automatically generates FIR filters employing only (parallel) distributed arithmetic. Nevertheless, implementation results obtained for the AuToCon outperform the results obtained for Core Generator, as the AuToCon considers different architectural solutions and applies more sophisticated optimisation techniques. The Core Generator takes into account mapping of element into the FPGA. Conversely, the AuToCon generates circuits on higher level using VHDL-approach, therefore it might seem that the throughput for the Core Generator circuit is greater. Implementation results proved that this is not the case.

Xilinx Inc. also provides a VHDL-based FIR filter description employing inverted-form and KCM approach [Pas01]. Nevertheless, the input width is fixed and only number of taps and coefficient values can be changed. Besides, the KCM LUTs operate on 4 times the input clock frequency, therefore comparison with the AuToCon is impossible. It should be noted

that there is a tendency for FPGAs to describe systems on high level (e.g. VHDL) and disregard relative placement of elements. This allows for reducing design time and/or implementing more sophisticated optimisation techniques.

5.2. Symmetry of Convolution Coefficients

Values of coefficients, in general, can be selected without any restrictions, however filters with symmetry are usually implemented, e.g. to obtain linear phase filters [Vai93]. The filters given in Figure 1-1 are also with symmetry (or asymmetry in the case of Sobel gradient filter). Table 5-1 gives possible 3×3 convolution kernels for different symmetries.

a)

$W_{0,0}$	$W_{0,1}$	$W_{0,2}$
$W_{1,0}$	$W_{1,1}$	$W_{1,2}$
$W_{2,0}$	$W_{2,1}$	$W_{2,2}$

b)

$\mathbf{W_{0,0}}$	$\mathbf{W_{0,1}}$	$W_{0,0}$
$\mathbf{W_{1,0}}$	$\mathbf{W_{1,1}}$	$W_{1,0}$
$\mathbf{W_{2,0}}$	$\mathbf{W_{2,1}}$	$W_{2,0}$

c)

$\mathbf{W_{0,0}}$	$\mathbf{W_{0,1}}$	$\mathbf{W_{0,2}}$
$\mathbf{W_{1,0}}$	$\mathbf{W_{1,1}}$	$\mathbf{W_{1,2}}$
$W_{0,0}$	$W_{0,1}$	$W_{0,2}$

d)

$\mathbf{W_{0,0}}$	$\mathbf{W_{0,1}}$	$W_{0,0}$
$\mathbf{W_{1,0}}$	$\mathbf{W_{1,1}}$	$W_{1,0}$
$W_{0,0}$	$W_{0,1}$	$W_{0,0}$

e)

$\mathbf{W_{0,0}}$	$\mathbf{W_{0,1}}$	$W_{0,0}$
$W_{0,1}$	$\mathbf{W_{1,1}}$	$W_{0,1}$
$W_{0,0}$	$W_{0,1}$	$W_{0,0}$

Table 5-1. Different filter symmetries: a) without symmetry, b) horizontal, c) vertical, d) horizontal-vertical e) point symmetry

The symmetry of the filter allows further optimisation of the circuit. The same coefficient inputs should be at first added, and then the common multiplication performed. Figure 5-1 shows the circuit simplifications, and Table 5-2 number of adders and multipliers after symmetry has been taken into account. It can be seen that for horizontal and vertical symmetry the number of multipliers is the same. However the number of adders and pixel delay elements is reduced for vertical symmetry because for this symmetry, only a single adder is needed for every common line. For the point symmetry, the number of multipliers is further reduced. It should be noted that for 3×3 convolution kernels, savings are less significant than for large kernel sizes, for which the number of multipliers is halved for horizontal or vertical symmetry, quartered for horizontal-vertical symmetry and reduced to 1/8 for point symmetry. It should be noted that for some 2D filters with the point symmetry, it may be beneficial to implement two independent vertical and horizontal 1D filtering [Cas96]. This is often the case for wavelet transforms [Cas96].

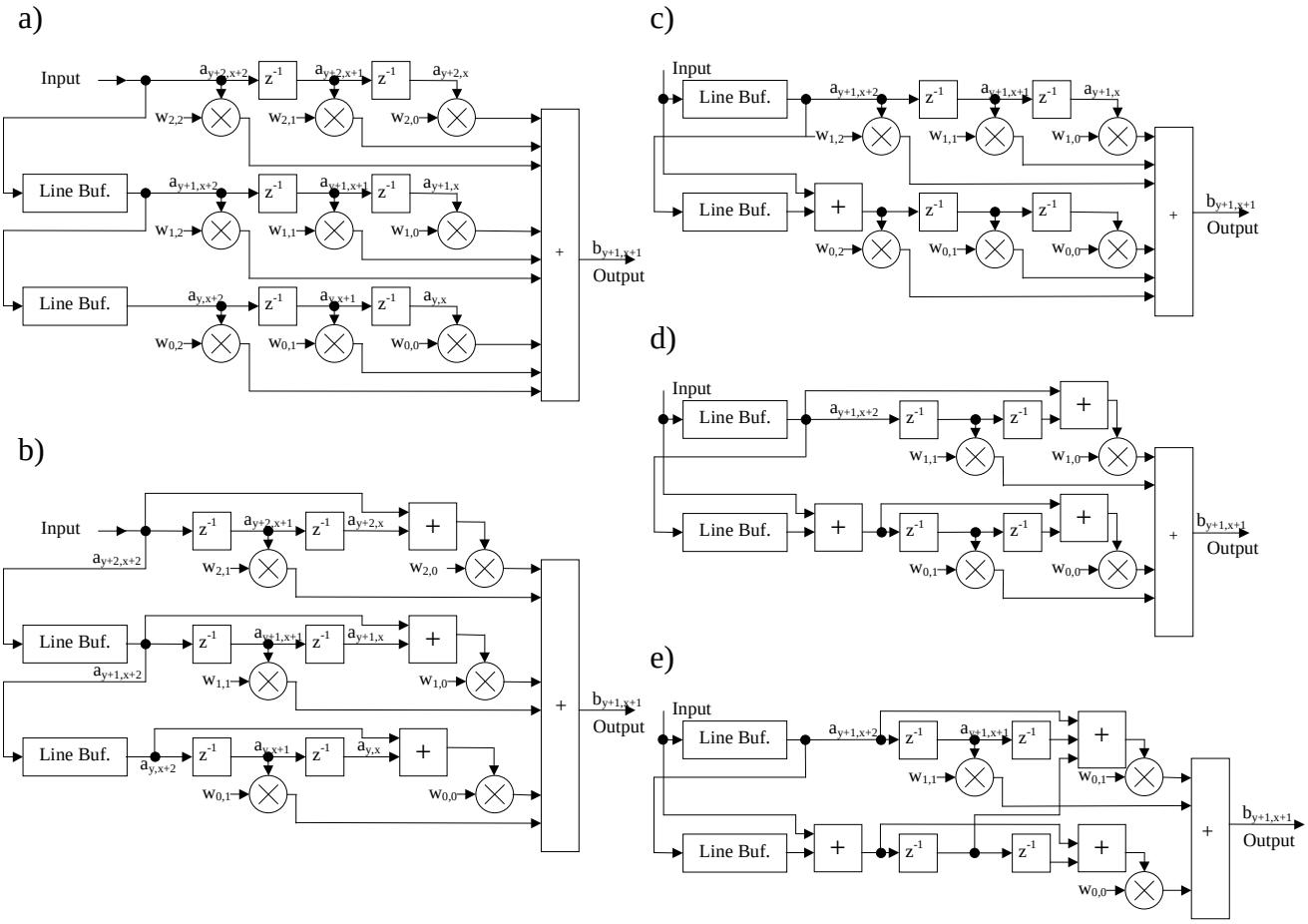

Figure 5-1. Convoler 3×3 architecture for different symmetry options: a) without symmetry, b) horizontal symmetry, c) vertical symmetry, d) horizontal-vertical e) point symmetry

Symmetry	# adders	# multipliers L	# ad. 3×3	# mul. 3×3
No-symmetry	N·M – 1	N·M	8	9
Horizontal	N·M – 1	N·⌈M/2⌉	8	6
Vertical	⌊N/2⌋ + ⌈N/2⌉·M - 1	⌈N/2⌉·M	6	6
Hor.-Vert.	⌊N/2⌋ + ⌈N/2⌉·M - 1	⌈N/2⌉ · ⌈M/2⌉	6	4
Point N=M	⌊N/2⌋ + ⌈N/2⌉·N - 1	(1+⌈N/2⌉)·⌈N/2⌉/2	6	3

Table 5-2. Number of adders and multipliers for different symmetry options and for a given convolution size 3×3. M- horizontal, N- vertical kernel size

5.3. LUT based Convoler (LC)

5.3.1. Concept

The structure of the LUT based Convoler (LC) is similar to the sum of products. 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 5-2, for convolution kernel size equal 1×2 and 8×8 multipliers. Let 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 a multiplier equals roughly w. The final adder width equals roughly $w+s$. Therefore total adders width for the sum of the LM is equal

$$w_{LM}= 3\cdot w+s. \tag{5-1}$$

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

$$w_{LC}= 3\cdot w. \tag{5-2}$$

Consequently, a penalty factor, a result of employing sum of LMs instead of the LC, is roughly

$$p = \frac{w_{LM} - w_{LC}}{w_{LC}} = \frac{s}{3\cdot w} \tag{5-3}$$

The above penalty factor is further employed for substructure sharing adders when two arguments are shifted by s. It should be also noted that employing the LC rather than the sum of LMs reduces the maximum width of the adder from roughly $w+s$ to w, and therefore reduces maximum propagation time.

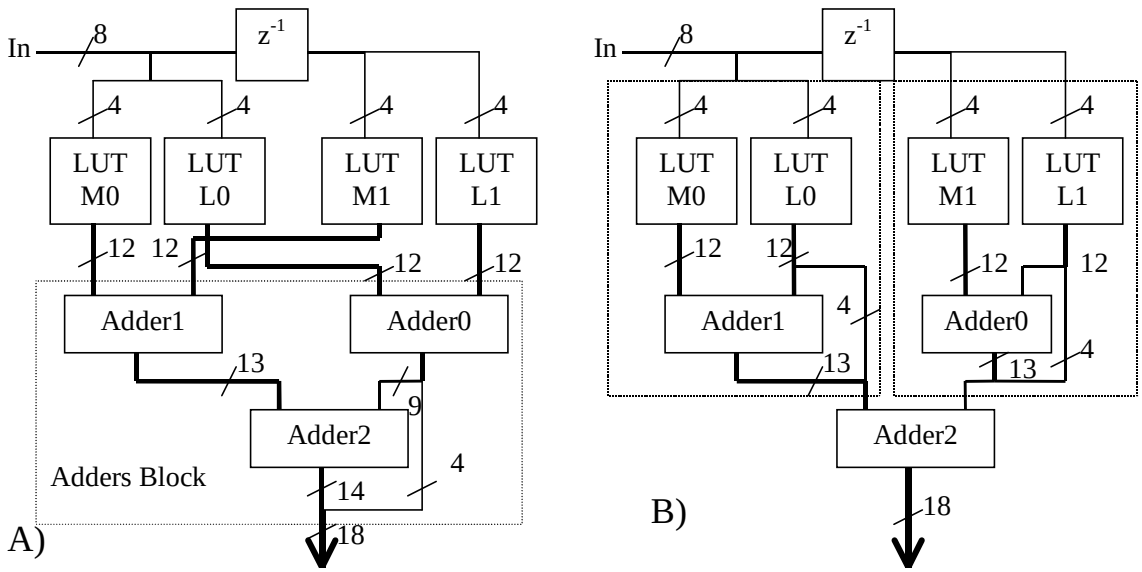

Figure 5-2. The structure of the convoler for $Y= A\cdot B_0 + z^{-1}\cdot A\cdot B_1$ for input and coefficient width K= 8. A) LC, B) sum of products

The LC is constructed in similar way as the LM, the same optimisation techniques are employed. However exhaustive search technique, which optimises all LUT memories and adders, is impractical to be implemented. Consequently for the LC, only local full search

optimisation is implemented, for which each multiplier is optimised separately using the full search technique and then all adders are merged into the adders block which is then separately optimised by techniques described in the next chapter.

5.3.2. Constant coefficients LUT based Convoler (KLC)

The KLC employs the same optimisation techniques as the KCM: LSB Address Width Reduction (LAWR), Don't Care Address Width Reduction (DAWR) and Memory Sharing (MS). In addition, optimisation techniques characteristic only for convolers are employed.

Similar Coefficients Optimisation (SCO)

Section 5.2 describes the symmetries of filters. However, also different symmetries and coefficient combinations can be used [Lu92]. Therefore, the AuToCon 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:

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

similar coefficient inputs are added:

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

and the final result is:

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

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

Pipelining Optimisation

Similarly like for the multipliers, the AuToCon generates a convoler with a sophisticated pipelining architecture, for which additional parameter *p* defines maximum number of logic elements between pipelining registers. Figure 5-3a shows an example of a convoler with straightforward pipelining architecture. For this method, however, additional pipelining registers are often required to compensate different pipelining delays. To reduce this drawback, pipelining optimisation is implemented, for which feeding points of arithmetic units are relocated in order to reduce unnecessary registers (similar optimisation is implemented in [Har96]). A result of the optimisation is shown in Figure 5-3b. It should be noted that the total convoler pipelining delay is often reduced in this method. This optimisation technique is implemented for every architecture described in this chapter.

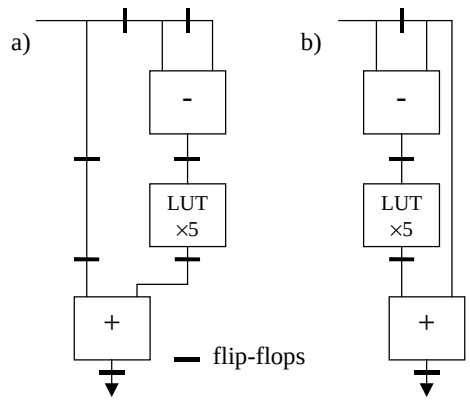

Figure 5-3. Implementation of $(2 + 5 \cdot z^{-1} - 5 \cdot z^{-2})$ filter for pipelining parameter p= 1 and a) without b) with pipelining optimisation

5.3.3. Dynamic Constant coefficients LUT based Convoler (DKLC)

For the DKLC, the value of coefficients can be changed in similar way, as it is in the case for the DKCM; rearranging the order of adders does not influence the LUTs programming schedule. For the DKCM, address multiplexing is performed on the input of the multiplier. Similarly for the DKLC, the multiplexer can be placed on the input of each multiplier. Let denote this option as DKLC-M. An alternative solution, denoted as DKLC-C, is to place the multiplexer on the convoler input and so the address sequence for programming LUTs will propagate through the convolution delay elements to the input of the LUTs. The drawback of this method is more sophisticated control logic. Besides, the number of programming cycles increases because of additional propagation time through the filter delay elements. In order to reduce this time the multiplexers should be rather placed at the beginning of each line. Therefore the programming sequence will propagate only through pixel delay elements. Summing up, $M \times N$ (M- horizontal; N- vertical kernel size) convoler requires N multiplexers and $M-1$ additional programming cycles for dynamic reconfiguration. In this option, however, similar coefficient adders (for symmetric filters, etc.) distract memory addressing and make the approach more complicated.

It should be noted that the LUTs can be programmed either in serial: a single multiplier is programmed at the time, or in parallel, when all multipliers are programmed simultaneously. The serial option has longer programming time but a single RAM Programming Unit (RPU) is required. The parallel option has short programming time but each multiplier requires its own RPU and therefore this option occupies more hardware. The choice between the serial and parallel option should be taken after considering the average time between coefficients changes in similar way as it was described in Section 4.6.

It should be noted that so far only self-programming architecture of the DKCM and DKLC has been considered. However, the LUTs can be programmed using an off-chip interface. In this case new LUT contents can be pre-calculated by a system processor and then written to the LUT memories. In this case the RPU is not required. Conversely, off-chip transfers are slower than internal ones and involve the system processor, which may not be accepted in some designs.

The DKLC can be implement with many different options. This is one of the reasons that the AuToCon cannot generate automatically any DKLC. Consequently including the DKLC to the AuToCon might be a suggestion for further work. Nevertheless Virtex II incorporates built-in fully functional multipliers, which makes the DKLC option less attractive.

5.4. Distributed Arithmetic Convoler (DAC)

5.4.1. Concept

The idea behind the DAC [Bur77, Min92, Do98] is to compute the convolution in different order than for the LC. The following mathematical transformation has been employed:

$$\sum_{i=0}^{N-1} h_i \cdot a_i = \sum_{i=0}^{N-1} h_i \cdot \sum_{j=0}^{L-1} 2^j \cdot a_{i,j} = \sum_{j=0}^{L-1} 2^j \cdot \sum_{i=0}^{N-1} h_i \cdot a_{i,j} \qquad (5\text{-}7)$$

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.*

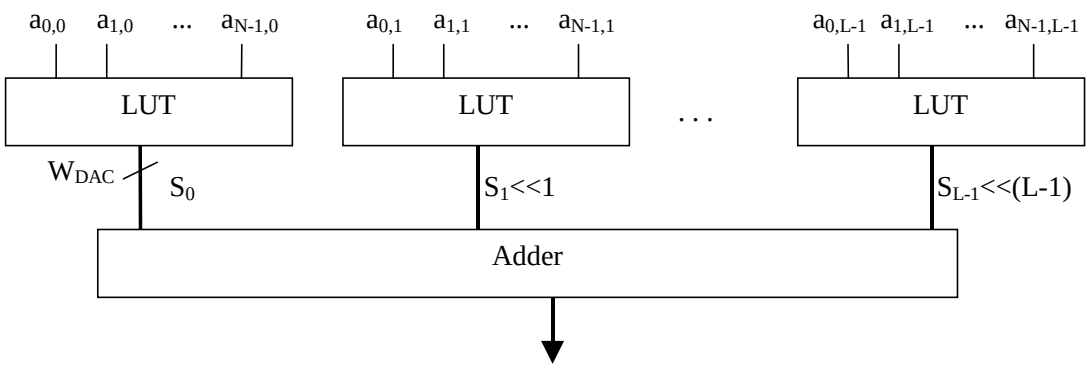

Figure 5-4. Diagram of the Distributed Arithmetic Convoler

In comparison with the LC, the LUT data bus width of the DAC is smaller, as it can be seen from eq. 5-8.

$$W_{DAC}=K+\lceil log_2(N+1) \rceil \qquad W_{LC}= K+W_{IN} \qquad (5\text{-}8)$$

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.

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. The drawback of the DAC solution is that the dynamic change of the coefficient is much more difficult in comparison to the LC, which makes this approach rather impractical for dynamic systems.

A diagram of the DAC is shown in Figure 5-4. 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 similarly like for the LM. The split of the memory should be implemented with respect to the cost relation between different memory modules and adders.

Consequently, in some cases the LUT based Hybrid Convoler (LHC) [Wia00c, Wia00d] - the hybrid of the LM and DAC, may be implemented, as the optimum memory split issue is concerned. For example, for the 3×3 convolution $N=9=3\cdot3$, 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. Therefore the alternative LHC may employ 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 [Wia00c]. Therefore 17 CLBs are saved by the use of the LHC.

5.4.2. Irregular Distributed Arithmetic Convoler (IDAC)

The previous solution assumes that the structure of the DAC is the same 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. Therefore more or less a combination of the LC and DAC is obtained. This novel, introduced by the author of this thesis, design approach is denoted as Irregular Distributed Arithmetic Convoler (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 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. 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 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 in Listing 5-1 at first assigns input bits with the lowest shift s_i (step S1). 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 returned 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 adders block (see the next chapter). 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 5.7.

The above algorithm is a novel algorithm which deals with the problem which has not been considered so far. Probably better optimisation techniques can be derived that employ better optimisation criteria in the greedy algorithm. Furthermore, optimisation techniques, which focus on global optimisation, should be implemented; some of these techniques are described in the next chapter.

5.5. Multiplierless Convolution (MC)

The MC employs similar optimisation methods as the Multiplierless Multiplication (MM) does. However for the MC, these methods are much more sophisticated as the convoler composes of many multipliers and optimisation is not constrained to a single multiplier as it is for the LC, but the whole convoler circuit is optimised all together. The design developments combines the following solutions:

1. Canonical Sign Digit optimisation (CSD) – the same method as described in Section 3.1.2, therefore this optimisation is not further described.
2. Substructure sharing (SS).
3. Pipeline Optimisation (PO) – described in Section 5.3.2.

5.5.1. Substructure Sharing (SS)

The SS has been already described for the KCM, however in the case of the convoler the optimisation techniques are much more complicated and some trade-offs have to be considered.

The choice of which substructure to choose at each iteration is a substantial problem as the selections in the early stages of optimisation influence the possible optimisations at the next steps. In the AuToCon, similarly like in the other similar systems [Pas99, Har96, Pot96, Cha93], the Greedy Algorithm (GrA) [Cor94] has been employed. The proposed algorithm takes the best partial solution which is found in the exhaustive search [Cor94], i.e. all possible two-input sub-expressions are tested and the best of them taken.

In the case of a multiplier, the best partial solution is selected according to the number of times *t* a common two-input sub-expression occurs, as the number of adders (subtractors) is reduced by *t-1* after the SS optimisation. In the case of a convoler, a curtain number *d* of delay elements (flip-flops) is often required, as the common sub-expression is often needed in different time slots. This is illustrated in an example of a convoler design in Section 5.7.2. The cost of these additional delay elements should be also considered when selecting the best partial solution.

To increase *t* by 1 (or more), a large number, denoted as d_{t+1}, of additional delay elements is often required. However, the cost of d_{t+1} delay elements may be greater than the cost of the common sub-expression adder. Therefore it is beneficial to implement the additional adder rather than the large number d_{t+1} of flip-flops. An alternative and more appropriate solution is to stop this optimisation step after *t-th* occurrence of the common sub-expressions and to consider *(t+1)-th* and next occurrences of the common sub-expression in the next optimisation steps. The latest solution allows optimisation program to select the best partial solution in more unconstrained way, which is beneficial to the algorithm results. For a 2D convoler, some sub-expressions may be even required after being delayed by a line buffer, which is impractical and should be rejected at the early stages of the optimisation. Consequently, the maximum number d_{max} ($d_{t+1} \leq d_{max}$) of delay elements between two successive time slots which are used in the next design stages, should be defined as:

$$d_{\max} \leq \frac{C_{Add}}{C_{FF}} \tag{5-9}$$

where: C_{Add}*- cost of the adder,* C_{FF}*- cost of the delay element (flip-flops).*

The above equation can be justified, as an increase of *t* by 1 decreases the number of adders by 1, therefore the additional cost of the delay elements must be lower than the cost of the adder. Nevertheless, *(t+1)-th* occurrence of the common sub-expression may be included in other sub-expressions in the next optimisation steps and including these inputs in this optimisation step usually makes the next optimisation steps less efficient, therefore eq. 5-9 should be considered as the inequality.

In order to reduce the output width of a common substructure adder, the inputs to the common sub-expressions adder should be at the same bit-significance. Consequently a penalty factor, given in eq. 5-3 and rewritten hereby,

$$p = \frac{s}{3 \cdot w} \tag{5-10}$$

is introduced. This factor takes into account the shift between arguments. To simplify the optimisation program, width *w* is assumed to be constant for all adders and equal the width of the input data to the convoler.

In the case when the subtrahend is shifted to the right in a SS adder, the LSBs cannot be copied directly to the output, therefore in this case, the common sub-expression should be negated. As a result, the minuend is now shifted to the right and therefore the LSBs can be directly copied to the output, see also Section 5.7.

In the case when two or more sub-expressions are found, for which savings are the same, some additional conditions are considered. In order to reduce the level of dependency between substructure sharing adders and so to allow better pipeline optimisation, it is beneficial to reduced the number of layers of the common substructure adders (an output of a common substructure adder is an input to the next common substructure adder, which forms an additional layer of adders). Consequently, a sub-expression which inputs are at lower layer (to simplify the algorithm the input with lower index, see Section 5.7 and Table 5-5) is selected first. In the case when inputs are at the same layer, which is the case only for the direct inputs (because the algorithm considers the index rather than the layer number) the sub-expression which inputs are closest to each other, is taken. This allows for reducing routing resources.

Summing up, in order to select the best partial solution, the following factors are considered:

1. How many times the common substructure occurs - *t*.
2. How many additional delay elements are required - *d*.
3. Bit-shift between inputs of the common substructure adder - *s*.
4. Maximum number of delay elements between two subsequent time slots- d_{max}.
5. Layer of the SS adders and position of the input.

The first three points conclude in the following equation:

$$Savings = (t - 1 - \frac{s}{3 \cdot w}) \cdot C_{Add} - d \cdot C_{FF} \tag{5-11}$$

and the fourth point concludes:

$$d_{t+1} \leq d_{max} \tag{5-12}$$

The sub-expression with the greatest *savings* is taken. The optimisation process is stopped when no sub-expression with *savings* greater than zero can be found. If the *savings* are the same for two or more sub-expressions, the 5th point is considered.

The AuToCon implements design in a similar way to [Har96]. However, [Har96] considers more sophisticated delay optimisation (together with pipelining registers and change in a register count). Conversely the AuToCon constrains the maximum number of delay elements between two subsequent feeding points (eq. 5-12) and takes into account shifts between inputs (eq. 5-11).

5.6. IDAC versus MC

The choice between the IDAC and the MC depends on the cost of the memories, adders and flip-flops within a selected FPGA device. Furthermore, the cost-relation between these elements differs for different designs, e.g. some designs may contain a lot of free memory resources but adders and flip-flops resources are already occupied, and therefore it is beneficial to employ more memory blocks than adders. This can be achieved by a decrease of memory costs. Even convoler parameters influence the cost of FPGA resources. For example, numbers of flip-flops and 16×1 LUTs are usually the same, as each LUT is associated with a flip-flop. Therefore, pipelining parameter p which defines the maximum number of logic elements between pipelining flip-flops, strongly influences relation between the number of incorporated LUTs and flip-flops. Consequently for $p=1$ the area is usually defined by the number of flip-flops and therefore the cost of flip-flops is relatively high in comparison to the cost of adders and memories. Conversely, for $p \geq 3$, the area is usually defined by the number of adders and memory modules, consequently the cost of flip-flops is relatively low. Summing up, parameters of the design strongly influence e.g. eq. 5-9 and eq. 5-11, and therefore the best solution differs significantly for different input parameters.

In conclusion, the choice between the IDAC and the MC depends strongly on the given FPGA device and inputs parameters. Besides, the area occupied by the MC depends strongly by the given coefficient values as some numbers require less non-zero bits and allows for better substructure optimisation than others. Conversely, for the IDAC, the width of the coefficients is the most important factor. In conclusion, for some coefficients, the IDAC is preferable, and the MC is a better choice for others.

As a result of the above conclusions, the AuToCon does not make any predefined assumptions and automatically trade-offs between these two architectures and implements some coefficients employing the IDAC, and other coefficients employing the MC. The trade-off between the IDAC and the MC cannot be considered separately for each coefficient (as for the KCM, see Section 3.3) as the SS depends strongly on relation between coefficients. Initially it might seem that the best solution can be obtained by checking all possible solutions, i.e. considering all possible combinations, for which each coefficient is implemented either as the MC or the IDAC. The number of combinations is equal $2^{N \cdot M}$ ($N \times M$-size of the kernel) and therefore this algorithm is impractical even for small convolution kernels.

MC vs. IDAC Algorithm

A novel implementation algorithm which trade-offs between the MC and the IDAC is introduced below:

Listing 5-2. MC vs. IDAC Algorithm

1. Determine the cost of every coefficient for the IDAC architecture (see Section 5.7).
2. Set optimisation *option* to *all*.
3. Mark all coefficients as (implemented employing) the MC.

(beginning of the optimisation loop)

4. Determine the MC circuit considering only coefficients which are marked as the MC.
5. Calculate the cost for each coefficient which is marked as the MC (see Section 5.7).
6. Find *total cost* of the circuit summing the cost of the IDAC coefficients and the cost of the MC coefficients.
7. **If** the *total cost* is equal or lower than the *best total cost* **then** accept changes
   else restore the best circuit (mark all coefficients as the IDAC or the MC as it was for the best circuit)
   and
   If *option=all* **then** set *option* to *one*.
   If *option=one* **then** finish the algorithm.

8a. **If** *option=all* **then for** every coefficient marked as the MC, compare the coefficient cost for the IDAC and the MC architectures and mark according to the better result.

8b. **If** *option=one* **then** find a coefficient for which difference of costs for the IDAC and the MC is the greatest and mark this coefficient as the IDAC.

9. **Go to** *step 4*.

The above algorithm assumes that the cost of the IDAC is rather not sensitive on whether the rest of the coefficients are implemented employing the IDAC or the MC. The cost of the IDAC circuits is the lowest when the coefficients are at same width and inputs at the same bit-significance (shift) are grouped in a single LUT. Therefore implementing a coefficient employing the MC rather than the IDAC may caused that the rest of the coefficients cannot be grouped so efficiently; i.e. the coefficients at different width are grouped to a single LUT, which increases the LUT data width and consequently the cost of

the IDAC. However the increase of the IDAC cost is rather insignificant, especially when coefficient widths are similar.

The above algorithm requires knowledge of each coefficient cost, however only the cost of the whole convoler is known, as the program disregards bounds between each multiplier. Consequently an algorithm that calculates the cost of each coefficient has been developed and is described in Section 5.7.

In the third point of the algorithm, initially all coefficients are assumed to be implemented as the MC. In the next iteration steps, more and more coefficients are implemented using the IDAC if the cost of individual coefficient employing the MC is greater than for the IDAC. Initially it seems that only one iteration step is required, however each coefficient marked as the IDAC, is not available in the next iterations of the SS optimisation and therefore the rest of the MC coefficients cannot be optimised so efficiently as they were in the previous iteration steps. Consequently, the cost of some additional MC coefficients may be greater than the cost of the corresponding IDAC coefficients and therefore these coefficients should be implemented employing IDAC, and so on.

Let denote coefficients, for which cost of the IDAC is lower than for the MC as trade-off coefficients (TOCs). Marking TOCs as IDAC causes that the TOCs are implemented more efficiently. Conversely, the total cost of the circuit may increase, as the rest of the coefficients are less efficiently optimised by the SS optimisation. In this case, better global result is obtained when the TOCs are implemented using the MC rather than the IDAC architecture, and therefore the TOCs should be marked as the MC, or equivalently, the circuit from the previous optimisation step should be restored.

In the previous paragraph all TOCs were considered altogether (optimisation *option= all*), however some of the TOCs when implemented as the IDAC may decrease the total cost of the circuit. Therefore two different optimisation options are considered in the MC vs. IDAC optimisation algorithm. Initially (*option= all*), all TOCs are marked as IDAC in every optimisation step until the total cost of the circuit decreases. Afterwards (*option= one*), only a single TOC, for which a local cost gain obtained by implementing the IDAC instead of the MC is the greatest, is implemented as the IDAC. This step is continued until the total circuit cost decreases.

It can be seen that in most cases the best solution is obtained when only *option= one* is implemented. However introducing *option= all* reduces the calculation time and usually influences the overall performance insignificantly.

5.7. Implementation Results

This section describes design steps for the given example of 1D filter:

$$H(z)= 59 + 183\cdot z^{-1} + 162\cdot z^{-2} - 7\cdot z^{-3} - 48\cdot z^{-4} + 12\cdot z^{-5} + 9\cdot z^{-6} + 2\cdot z^{-7} \qquad (5\text{-}13)$$

The filter can be illustrated as follows:

bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		1	1	-	-			
6		0	0	-	1			
5	1	1	1	-	0			
4	1	1	0	-	1			
3	1	0	0	1	0	1	1	
2	0	1	0	0	0	1	0	
1	1	1	1	0	0	0	0	1
0	1	1	0	1	0	0	1	0

Table 5-3. The filter representation, '-' – for negative numbers sign bit extension

5.7.1. Canonic Sign Digit (CSD) conversion

After CSD conversion the number of non-zero bits has decreased from 23 to 19. To better illustrate the optimisation techniques the zero bits are invisible.

bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		1	1					
6	1	1			-1			
5			1					
4					1			
3		-1		-1		1	1	
2	-1					1		
1			1					1
0	-1	-1		1			1	

Table 5-4. Filter representation after CSD conversion

5.7.2. Sub-structure sharing (SS)

In the first step of the SS optimisation, substructure $S_2= 1 + z^{-1}$ is introduced (Table 5-5, S_2 is denoted as 2 and 0_2). Substructure S_2 occurs three times in the following expressions: $-(1 + z^{-1}) + 2^6\cdot(1 + z^{-1}) + 2^7\cdot z^{-1}\cdot(1 + z^{-1})$, and only a single delay element (z^{-1}) is required to obtain expression: $2^7\cdot z^{-1}\cdot(1+z^{-1})$. Sub-expression $8\cdot z^{-5}\cdot(1 + z^{-1})$ can also be implemented using sub-expression S_2, however, this would require additional delay $d_{t+1}= 4$ (4 delay elements)

and $d_{t+1}>d_{max}= 1$ (see eq. 5-12), therefore this sub-expression is not included in this optimisation step.

Expression $S_2'= -1+z^{-5}$ also occurs three times: $4\cdot(-1+z^{-5}) + z^{-1}\cdot(-1+z^{-5}) + 8\cdot z^{-1}\cdot(-1+z^{-5})$ and requires also one delay element and shift between inputs is also equal zero. However, for S_2', input z^{-5} is delayed by 5 clocks in comparison to the fist input. For S_2, the second input z^{-1} is delayed by only 1 clock, and consequently to save routing resources, sub-expression S_2 is selected.

Bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		2	0_2					
6	2	0_2			-1			
5			1					
4					1			
3		3		-1		*1*	*0_3*	
2	3					0_3		
1			1					1
0	-2	0_2		1			1	

Table 5-5. Filter representation after two sub-expression optimisations (S_2 and S_3) have been implemented. In italic – expressions which cannot be shared in sub-expression 2 because $d_{t+1}>d_{max}$

In the next steps of optimisation the following sub-expressions are introduced:

$$S_3= -1 + z^{-5} \qquad S_4= S_3 + 2^{-2}\cdot z^{-6} \qquad (5\text{-}14)$$

$$S_5= z^{-2} + 2^3\cdot z^{-4} \qquad S_6= S_4 + 2^4\cdot S_2$$

Bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		$0_{6,2}$	0_2					
6	$0_{6,2}$	0_2			-1			
5			1					
4					0_5			
3		$6_{4,3}$		-1		0_5	0_3	
2	$6_{4,3}$					0_3		
1			5					0_4
0	-2	0_2		5			0_4	

Table 5-6. Filter representation after all sub-expression optimisation have been implemented. 0_i – zero inserted as a result of i-th sub-expression sharing

In the case of subtraction between two expressions, the order of subtraction (*a-b* or *b-a*) can be freely chosen, therefore should be selected to allow for direct copy of the LSBs (subtrahend should be shifted left). For example, for $S_3'= 1- 2^{-2}z^{-1}$, sub-expression should be negated to allow copping the LSB, i.e. sub-expression $S_3''= -1 + 2^{-2}z^{-1}$ should be taken.

5.7.3. Multiplierless Convoler

The circuit obtained from Table 5-6 is shown in Figure 5-5. It can be seen from Figure 5-5 that the additional number of flip-flops is significantly reduced in comparison to estimations made during the SS optimisation. Additional flip-flops after adder A_2, A_3, A_4 are not required, as all delayed sub-expressions are included to adder A_6. Even input expression $In{\cdot}z^{-7}$ is obtained in additional delay element after adder A_6.

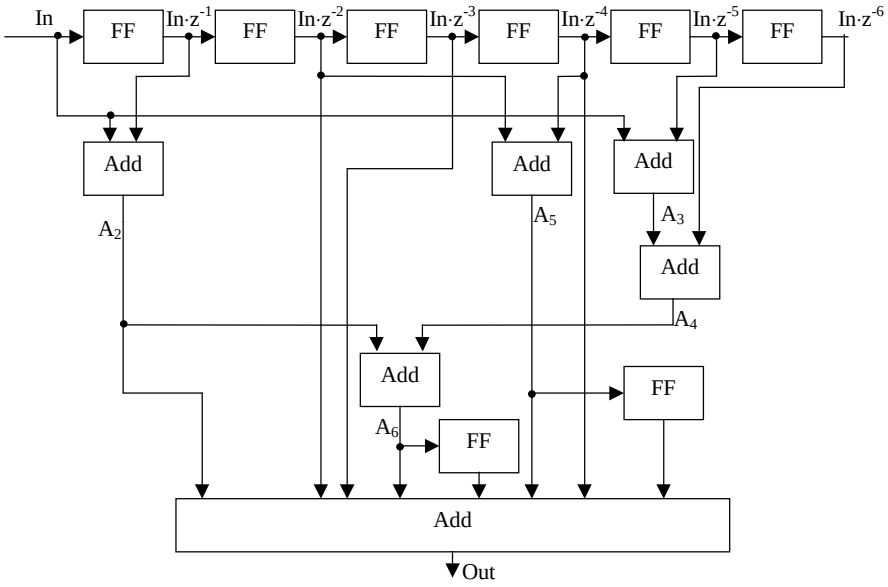

Figure 5-5. Block diagram of the MC circuit without pipelining. FF- flip-flops, Add- Adders

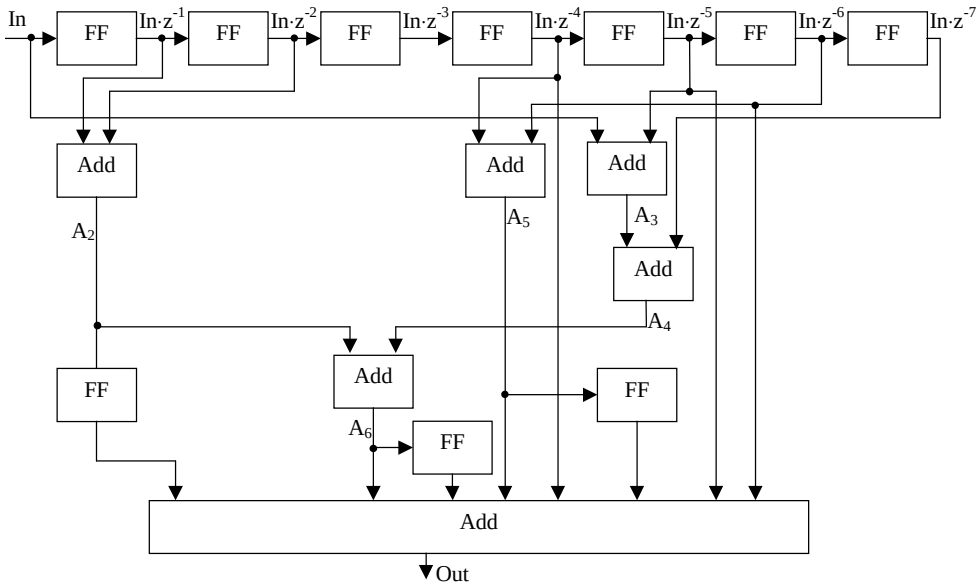

Figure 5-6. Block diagram of the MC circuit with pipelining. Pipelining flip-flops are inserted after every arithmetic unit

To speed up the above circuit, pipelining is introduced, i.e. flip-flops are inserted after every arithmetic unit (pipelining parameter $p= 1$). Additional pipeline optimisation is implemented for which in order to compensate different delays introduced by pipelining, the feeding points to the arithmetic units are reallocated rather than additional flip-flops inserted at the end of the arithmetic units. Figure 5-6 shows pipelined and optimised circuit corresponding to the circuit presented Figure 5-5.

Figure 5-6, in comparison to Figure 5-5, incorporates (disregarding pipelining flip-flops associated with every arithmetic unit) only 2 additional flip-flop sets: to generate $In \cdot z^{-7}$ and $A_2 \cdot z^{-1}$. It should be noted that the AuToCon allows for defining pipelining parameter p to be any integer from 1 to $+\infty$.

MC implementation results

Table 5-7 shows implementation results for different pipelining and synthesis options.

p	Area by AuToCon [LEs]	# FFs	XCV100CS144-6 (Synthesised Adders)		XCV100CS144-6 (Predefined Adders)		
			T [ns]	Area [LEs]	T [ns]	Area [LEs]	D [1/LE·µs]
1	83	183	7.4	92	7.4	71	0.738
2	83	138	8.9	91	8.5	68	0.853
3	83	102	12.7	88	11.4	72	0.860
4	83	100	14.1	92	13.9	71	0.719
5	83	85	17.4	90	17.8	72	0.661
$+\infty$	83	61	20.4	89	19.6	72	0.615

Table 5-7. Implementation results (area, minimum clock period T, functional density D= 1/A·T) for different pipelining and synthesis options

Two different results are presented hereby. The first one uses VHDL-synthesised adders. For example, the adder which input widths are 4-bit wide and output width is 5-bit wide when generated by FPGA Express (from Synopsys), a VHDL synthesis program, occupies 7 LEs (7 4-input LUTs). The estimated number of LEs for this adder is only 5 LEs. This adder does not use dedicated carry logic, which is the reason of the difference between estimated and reported number of LEs. This is only the case for Virtex family; for XC4000 family, adders are sinthesised properly.

An alternative and better solution, as it can be seen from Table 5-7, is to implement adders employing predefined adders which were previously generated by CORE Generator [Xil99a]. In the latest solution, dedicated carry logic is employed. For the predefined adders the reported number of LEs is smaller than the estimated number of LEs because the implementation report considers only usage of 4-input LUTs. Nevertheless, for dedicated

carry logic, the carry out signal (e.g. the 5-th MSB of the adder output for which inputs are 4-bit wide) does not use LUT logic, however the LUT associated with the carry out signal cannot be rather used by other logic. Therefore, the estimated number of LEs should also include the carry out signals.

The numbers of flip-flops estimated by the AuToCon and reported by Foundation 3.1 (distributed by Xilinx Inc.) are the same (they include additional flip-flops inserted at the input and output of the convoler, additional 18 flip-flops).

5.7.4. Irregular Distributed Arithmetic Convoler

The implementation results for the example given in eq. 5-13 and for the IDAC optimisation algorithm presented in Listing 5-1 is given in Figure 5-7 and Listing 5-3.

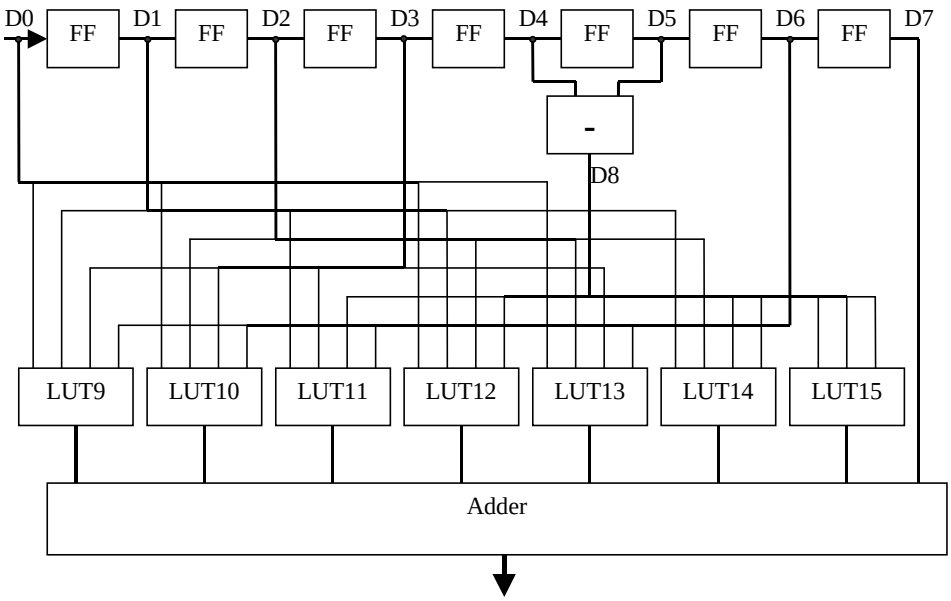

Figure 5-7. Block Diagram of the IDAC

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. In the given filter example, 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 5-7).

Listing 5-3. A fragment of VHDL code that describes IDAC LUT given in Figure B-3. data(i)(j) denotes signal d_i in Figure B-3 and j-th bit of this signal

```
d9: da4g generic map( -- LUT output shift= 0 – IDAC LUT number 9
        coeff0=> 9, coeff1=> -7, coeff2=> 59, coeff3=> 183, width_dout=> 9, insert_ff=> 0)
      port map (clk=>clk, ce=>ce,
        din0=> data(6)(0), din1=> data(3)(0), din2=> data(0)(0), din3=> data(1)(0), dout=> data(9)(8 downto 0));
```

```
d10: da4g generic map( -- LUT output shift= 1
          coeff0=> 9, coeff1=> -7, coeff2=> 59, coeff3=> 81, width_dout=> 9, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(6)(1), din1=> data(3)(1), din2=> data(0)(1), din3=> data(2)(0), dout=> data(10)(8 downto 0));
d11: da4g generic map( -- LUT output shift= 1
          coeff0=> 183, coeff1=> 6, coeff2=> 18, coeff3=> -14, width_dout=> 9, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(1)(1), din1=> data(8)(0), din2=> data(6)(2), din3=> data(3)(2), dout=> data(11)(8 downto 0));
d12: da4g generic map( -- LUT output shift= 2
          coeff0=> 59, coeff1=> 81, coeff2=> 183, coeff3=> 6, width_dout=> 9, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(0)(2), din1=> data(2)(1), din2=> data(1)(2), din3=> data(8)(1), dout=> data(12)(8 downto 0));
d13: da4g generic map( -- LUT output shift= 3
          coeff0=> 9, coeff1=> -7, coeff2=> 59, coeff3=> 81, width_dout=> 9, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(6)(3), din1=> data(3)(3), din2=> data(0)(3), din3=> data(2)(2), dout=> data(13)(8 downto 0));
d14: da4g generic map( -- LUT output shift= 3
          coeff0=> 183, coeff1=> 6, coeff2=> 162, coeff3=> 12, width_dout=> 9, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(1)(3), din1=> data(8)(2), din2=> data(2)(3), din3=> data(8)(3), dout=> data(14)(8 downto 0));
d15: da3g generic map( -- LUT output shift= 6
          coeff0=> 3, coeff1=> 6, coeff2=> -12, width_dout=> 5, insert_ff=> 0)
        port map (clk=>clk, ce=>ce,
          din0=> data(8)(4), din1=> data(8)(5), din2=> data(8)(6), dout=> data(15)(4 downto 0));
```

5.7.5. Approximated coefficients' cost for the MC and IDAC

In order to compare the cost of the MC and the IDAC, the cost of each individual coefficient must be found.

Multiplierless Convolution

At first, savings obtained by introducing common substructure are calculated. It can be seen that by introducing a sub-expression shared *t* times, the total number of adders is reduced by *t-1*, and *2·t* inputs are involved. Therefore an average saving (in number of adders) per SS input is:

$$S_A = \frac{t-1}{2 \cdot t} \tag{5-15}$$

Introducing each sub-expression often requires additional delay elements, which cost reduces the savings. Furthermore, inputs to the sub-expression adder are often shifted to each another, which increases the width of the adders in the next calculation stages. Therefore in this approach, the additional cost of delay elements and increased width of adders is evenly shared by all sub-expression inputs. Consequently, the approximated savings (in number of adders) introduced by a SS are as follows:

$$S = \frac{t - 1 - \frac{s}{3 \cdot w} - \frac{d \cdot w \cdot C_{FF}}{C_A}}{2 \cdot t} \tag{5-16}$$

where: d- the number of delay elements, C_{FF}- cost of a delay element (flip-flop), C_A – cost of the adder, s- shift between arguments, w- width of the arguments.

Savings for the circuit in Table 5-6, obtained by introducing sub-expressions S_5 and S_6 are given in Table 5-8. The cost of a delay element is $w \cdot C_{FF}/C_A = 0.4$, ($w = 4$). It should be noted that actual shift between arguments for S_6 is $s_6 = 6$ (initially it might seem to be equal 4); $s_5 = 3$.

Bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		**0.025**						
6	**0.025**							
5								
4					*0.088*			
3		**0.025**				*0.088*		
2	**0.025**							
1			*0.088*					
0				*0.088*				

Table 5-8. Saving obtained by introducing sub-expressions S_5 and $\mathbf{S_6}$

The savings obtained in the latest stages of the optimisation (the greatest indices) are evenly divided to two involved inputs. For example, for input $2^7 \cdot z^{-1}$, saving $S = 0.025$ is divided (by 2) into inputs $2^7 \cdot z^{-1}$ and $2^7 \cdot z^{-2}$ and then savings $S = 0.267$ (sub-expression S_2, $t=3$, $d=1$, $s=0$) added, in total $S = 0.279$. Further optimisation savings are added to the previous ones. As the result the following final savings are obtained.

Bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		0.279	0.279					
6	0.279	0.279			0			
5			0					
4					0.088			
3		0.210		0		0.088	0.210	
2	0.210					0.210		
1			0.088					0.121
0	0.267	0.267		0.088			0.121	

Table 5-9. Total savings obtained by the SS

The total cost of each coefficient is proportional to the number of non-zero CSD bits minus the total SS savings obtained for the coefficient, plus an approximated number of pipelining flip-flops

$$C_C = C_A \cdot (N_A - \sum S_i) + ((w+3) \cdot C_{FF} \cdot N_A) >> (p-1)) \qquad (5\text{-}17)$$

where: C_C – approximated cost of a coefficient, N_A – number of non-zero CSD bits, p – pipelining parameter, >> - a shift to the right.

Bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		0.721	0.721					
6	0.721	0.721			1			
5			1					
4					0.912			
3		0.790		1		0.912	0.790	
2	0.790					0.790		
1			0.912					0.879
0	0.733	0.733		0.912			0.879	
MC N_A	2.244	2.965	2.633	1.912	1.912	1.702	1.669	0.879
MC [LEs]	15.7	20.8	18.4	13.4	13.4	11.9	11.7	6.2
IDAC[LEs]	19	23	21	15	13	11	15	9

Table 5-10. Approximated cost (for p=∞) of non-zero CSD bits and coefficients, and the cost of the alternative IDAC solution, Cost RAM 16×1= 1, cost adder = 1LE/bit

It should be noted that for FPGAs the cost of adders depends on the adder width. Therefore to simplify the above considerations, all adders are considered to be at the same width equal width of the convolution input plus 3.

Irregular Distributed Arithmetic Convoler

For the IDAC, the cost of each coefficient depends on the width of the coefficient rather than number of non-zero bits and SS optimisation result. Consequently different circuit cost may be obtained for the IDAC and the MC architectures, and therefore these two architectures are compared with each other separately for every coefficient, and the better coefficient circuit is taken.

The previous section described the algorithm for calculating the cost of the MC for every coefficient. Now a similar algorithm for the IDAC is derived:

1. Calculate the widths of coefficient w_{coeff}. At this step, even coefficients are shifted to the right until the odd coefficients are obtained. The shifts are implemented because for the even coefficients the LSBs of the result are always zero.
2. For all different kind of memory modules:
a) Calculate data width of each IDAC block:

$$w_{data} = w_{coeff} + \lceil log_2(1+w_{ma}) \rceil \qquad (5\text{-}18)$$

where: w_{data}- data width of the considered IDAC memory, w_{coeff}- coefficient width calculated in the first point, w_{ma}- address width of the memory module.

The above equation is similar to eq. 5-8 and assumes that all coefficients grouped in a IDAC ROM are at the same width, and input bits are at the same bit-significance. Therefore the

actual width of the data may be greater. Conversely, the ceiling function is employed, which tends to increment the actual data width of the IDAC ROM.

b) Calculate the number of required memory modules and adders:

$$n_m = \left\lceil \frac{w_{data}}{w_{md}} \right\rceil \cdot \frac{w_{in}}{w_{ma}} \tag{5-19}$$

$$n_a = \frac{w_{in}}{w_{ma}} \tag{5-20}$$

where: n_m- number of required memory modules for a single coefficient, n_a- number of adders required for a single coefficient, w_{md}- address width of the considered memory module, w_{in}- width of the input data to the convoler.

c) Calculate the cost of a coefficient

$$C = n_m \cdot C_{Mem} + n_a \cdot C_{Add}(w_{data}+1) + C_{FFMem} + C_{FFAdd} \tag{5-21}$$

where: C_{Mem}- cost of the considered memory module, $C_{Add}(w_{data}+1)$- cost of the adder which is $(w_{data}+1)$-bit wide, C_{FFMem} – approximated cost of pipelining flip-flops inserted after memory blocks (only for asynchronous memories), C_{FFAdd} – approximated cost of pipelining flip-flops inserted after adders

$$C_{FFMem} = \begin{cases} 0 & for\ p \geq 3 \\ w_{data} \cdot n_a \cdot C_{FF} & for\ p \leq 2 \end{cases} \tag{5-22}$$

$$C_{FFAdd} = n_a \cdot (w_{data}+1) >> (p-1) \tag{5-23}$$

where: p – pipelining parameter, C_{FF} – cost of a flip-flop.

It should be noted that the average width of the adder is incremented by 1 for the IDAC, in comparison to the MC for which the adder width in incremented by 3. This assumption is made because the adder inputs are usually at better bit-alignment for the IDAC; and for the MC, adder width tends to increase after the SS.

d) Compare the current cost with the best obtained cost, and store the best of them.

In the considered example, for coefficient 162 (=2·81, w_{coeff}= 7):

- for 16×1 (w_{ma}= *4*, w_{md}= *1*) memory and w_{in}= *4*, C_{Mem}= *1 LE*, $C_{Add}(11)$= *11 LEs* the following values are obtained: w_{data}= *10*, n_m= *10*, n_a= *1*, *C= 21 LEs*.

5.7.6. MC vs. IDAC Algorithm

The MC vs. IDAC algorithm was described in Section 5.6, here only an example of the algorithm implementation is given. The cost of the IDAC in comparison to the MC is

given in Table 5-11. For coefficients z^{-4} and z^{-5} cost of the MC is higher than for the IDAC (see Table 5-10), therefore these coefficients are marked as the IDAC. Table 5-11 shows an intermediate result of the algorithm.

bit \ coeff.	1	z^{-1}	z^{-2}	z^{-3}	z^{-4}	z^{-5}	z^{-6}	z^{-7}
7		4_2	0_2					
6	4_2	0_2						
5			3					
4								
3		-3		0_3	I	I	1	
2	-1				D	D		
1			0_3		A	A		0_4
0	-2	0_2		1	C	C	0_4	
MC [LEs]	17.1	21.4	17.5	13.2			13.8	6.8
IDAC[LEs]	19.0	23.0	21.0	15.0	13.0	11.0	15.0	9.0

Table 5-11. MC representation of the convoler and relative cost (in number of MC adders) of non-zero bits

Excluding coefficients z^{-4} and z^{-5} from the SS optimisation causes that the SS optimisation is altered and therefore cost of other coefficients may be greater than the cost for the IDAC, etc. For this example, the cost of coefficient *1* has increased from 15.7 LEs to 17.1. Conversely, the cost of coefficient z^{-2} has decreased from 18.4 to 17.5 LEs. Nevertheless, by introducing the IDAC, the total cost of the circuit has increased from 111.4 LEs to 114.1 LEs. Therefore the old circuit (without the IDAC) is restored. In the next step of optimisation only a single coefficient for which the difference between the cost of the MC and the IDAC is the greatest, is marked as the IDAC. In the given example coefficient z^{-5} is implemented using the IDAC architecture. The overall cost of such a circuit is 114.2 LEs which is slightly more than for the MC architecture. In conclusion, the final circuit employs only the MC as shown in Tables 5-6 and 5-10.

If cost of a flip-flop is incremented from 0.4 to 1.0 LE then the cost of the MC increases (mostly because $d_{max}=0$, see eq. 5-9) and the final circuit is as shown in Figure 5-8 where only coefficients z^{-1} and coefficients z^{-4} and z^{-5} are implemented using the IDAC. It should be noted that 1-input LUT is implemented as a direct connection to the final adder block, therefore this LUT does not occupy any hardware.

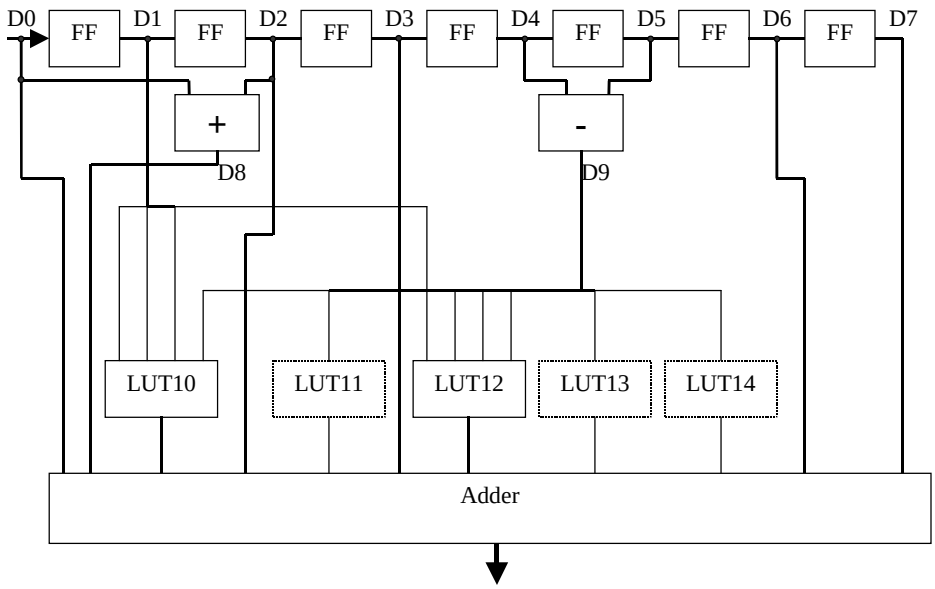

Figure 5-8. An example of the circuit when both the MC and IDAC architectures are employed

Implementation Results

Table 5-12 presents implementation results for Xilinx XC4005XLPC84-09. The AuToCon outperforms Core Generator, which generates only Distributed Arithmetic circuit for 1-D convolution. Furthermore in Core Generator, the pipelining option cannot be specified.

Circuits 2 and 4 in Table 5-12 are as in Figures 5-5 and 5-6 respectable. Circuit 3 is similar as in Figure 5-5, however some pipelining modifications are implemented. Circuit 5 is as in Figure 5-8. Circuit 6 is similar as in Figure 5-8. This circuit is generated when the pipelining flip-flops are not taken into account when trading off between the MC and IDAC. The estimated number of flip-flops should be smaller than for e.g. circuit 4. However this circuit requires more LUTs than the counterpart, and therefore more pipelining flip-flops are inserted. Consequently, pipelining flip-flops should be also taken into account when trading off between the IDAC and MC, which is the case for circuits 7 and 8. Therefore circuit 8 consumes less FFs in comparison to circuit 6. Summing up, the generated circuits are different not only for different cost relations between FPGA resources but also depends on the pipelining parameter.

Table 5-12 uncovers an inaccuracy of the AuToCon. For $p=\propto$ and $C_{FF}= 1$, $C_{LUT16\times1}= 1$ the total cost of the circuit 2 is C= 83 + 43= 126, and the total cost of circuit 5 is C= 134 + 28= 162, that is, the result for circuit 5 is worse than for circuit 2. The reason of this inaccuracy is that for the circuit 2, shown in Figure 5-5, additional flip-flops after *adder 2, 3, 4* are not required, as all delayed sub-expressions are included in *adder 6*. This, however, is

difficult to be estimated during design optimisation and therefore the reduction of the number of flip-flops is not considered by the architectural trade-off algorithm.

Circuit	# 16×1 LUTs	# FFs	T [ns]
1) Core Generator	169	201	12.1
2) AuToCon p=∝ C_{FF}=0.4	83	43	39.6
3) AuToCon p=2 C_{FF}=0.4	83	119	17
4) AuToCon p=1 C_{FF}=0.4	84 (83)	163 (159)	11
5) AuToCon p=∝ C_{FF}=1	134 (136)	28	36
6) AuToCon p=1 C_{FF}=1	135 (136)	208 (210)	11.8
7) AuToCon p=2 C_{FF}=1	107 (105)	116	16
8) AuToCon p=1 C_{FF}=1	107 (105)	178	12.2

Table 5-12. Implementation results for Core Generator [Xil99a] and the AuToCon for different pipelining parameter p and cost of flip-flops C_{FF}, ( )- values estimated by the AuToCon if different from implementation results

Consequently to improve the algorithm, the AuToCon should consider the actual (not estimated) circuit cost during searching for the optimal circuit. This however would significantly increase circuit generation time. However, for small designs, the AuToCon generates a circuit within a second, which is only a small fraction of the time required to implement the design in a FPGA. Consequently, this algorithm improvement might be a suggestion for a future work.

5.8. Conclusions

In this chapter a novel algorithm for the IDAC has been presented. This algorithm disregards regular structure of the DAC, and therefore allows for further optimisation of the circuit. Similarly like for the multipliers the MC proven to be more efficient architecture than the IDAC or LC. One of the most important features of the AuToCon is that cost-relation between FPGA resources is user-defined and no architectural assumption is made. Therefore a novel architectural algorithm has been presented which trade-off between the IDAC and the MC.

Implementation examples and results have been presented in Section 5.7, and as a result, the circuit generated by the AuToCon significantly outperforms the commercial counterpart.

6. Optimisation of the adders tree

Addition is a fundamental operation for the convolution, as for example, the MC incorporates only adders, and appropriate routing of the adders significantly influence the total cost of the circuit. Different optimisation techniques for carry-save-adders have been studied e.g. in [Kim98, Kim00]. According to the author knowledge, no research has been done in order to optimise the ripple-carry-adders network. For example, Thien-Toan Do et. al. [Do98] 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 automated tool has to be developed in order to find an optimal network of adders [Jam01a, Jam01b, Jam01c].

This chapter studies also sophisticated input parameters of the adders block. Basically, only input widths are required, however, to achieve hardware savings, the input ranges and even inputs correlation should be considered. Furthermore, correlation between inputs depends on the architecture: the MM, LM or DAC, which makes implementation more difficult. Further, different heuristics for finding the optimal adders tree are investigated. Implementation approaches and results are included to illustrate how the adders tree is optimised.

6.1. Implementation of adders in FPGAs

For ASIC designs, the classic problem of carry propagation is resolved by numerous techniques, e.g. carry-look-ahead, carry-select [Omo94], 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 carry-save addition [Omo94]. Consequently, in ASICs, carry-save addition is a substantial technique implemented in convoler designs [Haw96].

FPGAs incorporate a dedicated carry propagate circuit [Xil99b, Alt99] 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 [Xil99b, Xin98]. Furthermore the dedicated carry-propagate circuit is implemented outside the standard logic (LUT), which results in the adder

area reduction. Consequently, using only ripple-carry adders in FPGA designs is the best solution with respect to the propagation time and occupied area [Xin98].

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 6-1. 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 6-1).

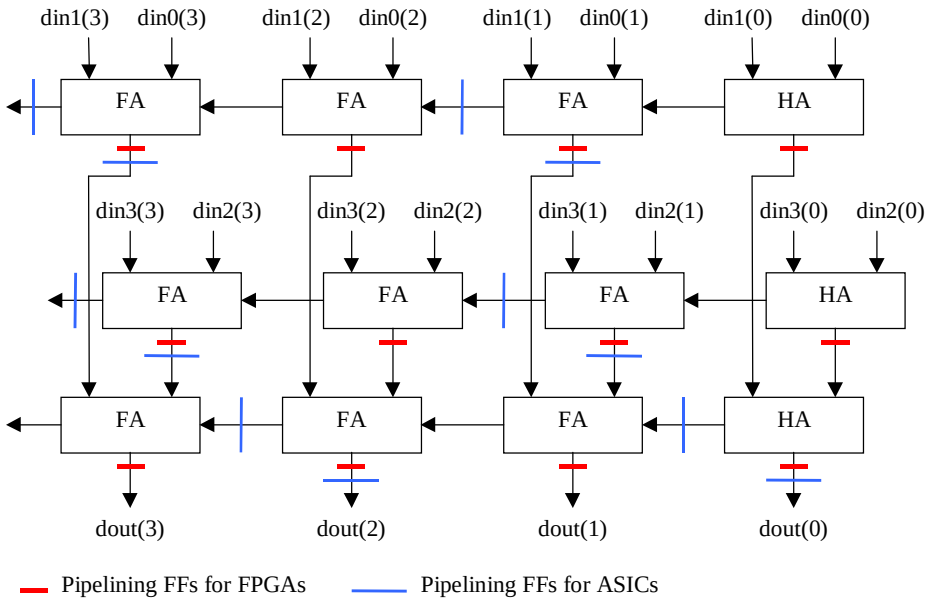

Figure 6-1. An example for different pipelining strategies for ASIC's and FPGA's additions for the equation: dout= din0 + din1 + din2 + din3. Pipelining parameters: N=2 (ASIC), K=1 (FPGA)

Nevertheless, the build-in carry logic cannot nullify the carry propagation time, and therefore for FPGAs, 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 delay 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 from the carry-out to the carry-in. Furthermore the propagation time through the programmable interconnects is usually comparable or even greater than the propagation time through LUT logic.

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 6-1) is employed. This, however, would complicate the system design and require additional flip-flops to be inserted according to the cut - set pipelining rule [Pir98]. 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 outperforms delayed addition technique [Luo98] 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 flip-flops are required for each 4-2 adder.

Summing up, for FPGAs the best solution seems to be the usage of dedicated adders and pipelining after every K additions as it is shown in Figure 6-1 and the FPGA option.

6.2. Addition parameters

6.2.1. Input parameters

The previous chapters describe methods of implementing convolers. Now, let consider the adders tree block alone. This block is independent of these methods and therefore let define input parameters which determine the adders 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:

$$y_{min}= a_{min} + b_{min} \qquad y_{max}= a_{max} + b_{max} \tag{6-1}$$

It should be noted that for a subtraction the above equation also holds. The use of minimum and maximum values instead of the bus widths can cause hardware savings, as some inputs might not use the full binary 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 will be 6-bit wide. Besides, using data range instead of the bus width allows for easy dealing with positive and negative numbers without additional hardware overheads – allows skipping sign-bit coding if possible.

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 is also included.

6.2.2. Correlation between inputs

To further decrease adder widths, correlation between inputs should be considered. Because an assumption is made that inputs a_i (see eq. 1-1, in order to simplify notation only 1-D convolution is hereby considered) are uncorrelated, the correlation is observed only

within the multiplication $h_k \cdot a_i$ when the addition of the partial products of the same multiplication takes place; e.g. addition of shifted a_i for the MM. Consequently, the correlation is considered separately for each multiplication, and therefore in this section rather a multiplication than a whole convoler is considered.

The correlation should be considered for every intermediate addition; e.g. for the addition: $y= a + b + c$, at first intermediate 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 intermediate addition; i.e. only inputs to the intermediate adders block should be taken into account, and therefore the actual adders connection network within the intermediate adders block is disregarded.

To describe correlation between inputs, different architectures will be further approached.

6.2.2.1. Multiplierless Multiplication

For the MM, no correlation is observed unless a subtraction $y= a - b$, where $a= 2^k b$, between the same argument takes place. In this case, the eq. 6-1 should be replaced by:

$$y_{min}= a_{min} - b_{min} \qquad y_{max}= a_{max} - b_{max} \qquad (6\text{-}2)$$

6.2.2.2. 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 (address) 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 adders block output. It can be seen that inputs to all LUTs but the MSBs LUT operate on the positive binary range:

$$I_{j\,max}= 2^{Wj}-1 \qquad I_{j\,min}= 0 \qquad \text{for } j= 0 \,..\, k-1. \qquad (6\text{-}3)$$

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

$$I_{k\,max}= I_{max}>>s_k \qquad I_{k\,min}= I_{min}>>s_k \qquad (6\text{-}4)$$

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:

$$O_{j\,max}= h \cdot I_{j\,max} \qquad O_{j\,min}= h \cdot I_{j\,min} \qquad \text{for } h \geq 0$$

$$O_{j\,max}= h \cdot I_{j\,min} \qquad O_{j\,min}= h \cdot I_{j\,max} \qquad \text{for } h < 0 \qquad (6\text{-}5)$$

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. 6-5 – only index j disappears. The relation between the LUT output ranges is specified:

$$O_{\max} \le [\sum_{j=0}^{k}(O_{j\max} << s_j)] >> s \qquad O_{\min} \ge [\sum_{j=0}^{k}(O_{j\min} << s_j] >> s \qquad (6\text{-}6)$$

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 which determines the correlated maximum and minimum of an intermediate adder A (the set A contains indices of all inputs to the intermediate 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 intermediate 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 and only if the index $j+1$ also belongs to. Consequently, C contains successive elements: $j, j+1, ..., k-1, k$, where $j-1$ is the greatest index of the LUT which is not included in the intermediate addition block, i.e. $(j-1) \notin A$. The input range of the of set C is calculated in a similar way as input range of the MSB LUT (eq. 6-4) and can be expressed as follows:

$$I_{Cmax} = I_{max} >> s_C \qquad I_{Cmin} = I_{min} >> s_C \qquad (6\text{-}7)$$

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 set C can be calculated in the following equation which is similar to eq. 6-5.

$$C_{A\,max} = h \cdot I_{C\,max} \qquad C_{A\,min} = h \cdot I_{C\,min} \qquad \text{for } h \ge 0$$

$$C_{A\,max} = h \cdot I_{C\,min} \qquad C_{A\,min} = h \cdot I_{C\,max} \qquad \text{for } h < 0 \qquad (6\text{-}8)$$

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

$$O_{A\min} = [(C_{A\min} << s_C) + \sum_{i \in A-C}(O_{i\min} << s_i)] >> s_A$$

$$O_{A\max} = [(C_{A\max} << s_C) + \sum_{i \in A-C}(O_{i\max} << s_i)] >> s_A \qquad (6\text{-}9)$$

where: s_A *- the shift of the intermediate 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 an intermediate 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 6-1

Let consider the example from Figure 3-5, for input range: *–99* to *99* (*8*-bit wide input) and coefficient $h=100$. Consequently, from eq. 6-3 and 6-4, the input range is $I_0= 0 \text{ to } 15$ and $I_1= -7 \text{ to } 6$. Employing eq. 6-5, 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. 6-6) $O_A= -11\,200 \text{ to } 11\,100$. Otherwise (from eq. 6-9), $O_A= -9\,900 \text{ to } 9\,900$.

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 divided into 4-bit wide LUT and a 1-bit wide LUT), $I_0= 0 \text{ to } 15$ and $I_1= -1 \text{ to } 0$. Consequently, uncorrelated (from eq. 6-6) 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 DKCM. In this case, instead of multiplication coefficient h, coefficient range h_{min} and h_{max} should be used. Consequently, eq. 6-5 should be replace by:

$$O_{j\,max}= Max\,(h_{max}\cdot I_{j\,max},\ h_{mim}\cdot I_{j\,min})$$

$$O_{j\,min}= Min\,(h_{max}\cdot I_{j\,min},\ h_{min}\cdot I_{j\,max}) \qquad (6\text{-}10)$$

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

Example 6-2

Let consider the example from Figure 4-1 for input range *–99* to *99* (*8*-bit wide input) and the coefficient values $h_{min}= -100$ and $h_{max}= 100$. Consequently, $I_0 = 0 \text{ to } 15$ and $I_1= -7 \text{ to } 6$. The output ranges are: $O_{0min}= -1500$, $O_{0max}= 1500$ and $O_{1min}= -700$, $O_{1max}= 700$. Output range of the addition, when the correlation is not taken into account is $O_A= -12\,700 \text{ to } 12\,700$. Otherwise, $O_A= -9\,900 \text{ to } 9\,900$.

6.2.2.3. Distributed Arithmetic

For the DAC (or IDAC), the correlation should be considered in a similar way as for the LM. However inputs to the LUT are from different filter inputs $a_{,j}$ (see eq. 1-1 and 5-7). Therefore each input to the LUT should have a separate correlation entry k_j, C_{jmin}, C_{jmax} (see Section 6.2.3) 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 a_i. 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 miss input is disregarded. Therefore, the simplified approach may result in the hardware overheads. However, the miss 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.

6.2.3. Summary of the input parameters

In our approach for every input *j*, the following 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. 6-2 instead of eq. 6-1 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: $C_{i\,min}$, $C_{i\,max}$, these values are calculated in eq. 6-8 for the correlation set $C= \{j, j+1, ..., k\}$. If all inputs from *j* to *k* are involved in a intermediate addition *A* then variables C_{Amin}, C_{Amax} rather then O_{jmin}, Oj_{max} and eq. 6-1 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.

6.2.4. Addition tree structure

Additional assumptions and rules for constructing the addition tree must also be specified. A binary tree is employed for which the number of inputs to the next adder layer is halved. An example of a binary adder tree for 6 inputs is given in Figure 6-2. 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.

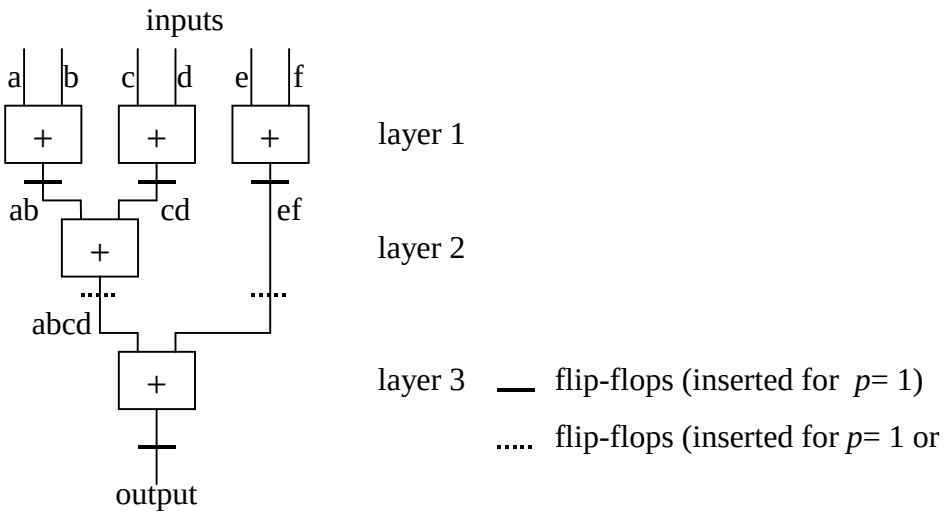

Figure 6-2. An example of the adders tree for 6 inputs

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) 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, as it is the case for e.g. carry look ahead adders [Omo94]. 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 an adder may be smaller than the width of the addition result in the case when the one argument is shifted to another. An example of shifted arguments is given in Figure 3-5, for which 4 LSBs are directly copied to the output. For subtraction, however, the LSBs of the subtrahend cannot be copied because conversion to two's complement has to be carried out on the subtrahend before the addition is implemented.

Consequently, subtraction and addition have to be treated in a slightly different way [Wia00b] 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 6-2. For *p*= 1, flip-flops are inserted after every addition 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 6-2). Therefore, in these cases no logic cell is required and consequently pipelining flip-flops are not 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, for adders blocks, 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.

6.2.5. Filters Example

Implementation results of different algorithm have been given for different filters. Table 6-1 shows parameters of these filters.

Filter	# inputs to adders bl.	Input range	# taps	Cost [# FA] for the GrA
a	16	0-15	5×5	111
b	11	0-15	5×5	74
c	13	-99-99	8	128
d	39	0-199	41	413
e	85	0-9999	41	1358
f	290	0-9999	41	3730

Table 6-1. Parameters of the implemented FIR filters

6.3. Greedy algorithm

A Greedy Algorithm (GrA) [Cor94] 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. The GrA is the quickest algorithm from all algorithms 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 this 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 $m_2 = s_2 + w_2$. 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 smallest 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 of the addition (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 tend to be of a greater shift and therefore easier to be grouped in the next iterations. Rule 1.2 tries to optimise partial solution by taking the smallest input width. This rule also supports searching for a good overall solution as wider inputs can be easier grouped with inputs with a greater shift, which are left for the next iterations. The second input is taken rather to optimise at first the partial and then overall result. 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 selects an 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 might 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, and this causes that a different priority queue should be specified, etc.

6.4. Exhaustive search

6.4.1. Concept

The best possible result can be always found by search through all possible solutions. 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 (letters: *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 find out 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*:

$$S_1(n) = \begin{cases} n \cdot (n-2) \cdot (n-4) \cdot \ldots \cdot 3 \cdot 1 & for\, odd\, n \\ S_1(n-1) & for\, even\, n \end{cases} \tag{6-11}$$

The total number of possible solutions *S(n)* is defined in an iterative way and is a product of the number of combinations on this layer and the total number of combinations on

the upper (closer to the output) layers, i.e. for the adders block for which number of inputs is halved:

$$S(n)= S_1(n)\cdot S(\lceil n/2 \rceil) \qquad (6\text{-}12)$$

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 6-2. The number of possible combinations for a given number of inputs n to the adder block

It can be seen from Table 6-2 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.

6.4.2. Constrained Search (CS)

As the number of possible solutions is growing rapidly with the growing number of adder inputs, a modification of the ES method is hereby 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 a method, similar to the ES, is implemented. This method, however, stops calculating a group of solutions in its early stages (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 comparison procedure is executed after every layer of the adders tree is completed.

The CS technique saves the calculation time, as solutions which are unlikely to give the best solution are skipped on a low layer and therefore upper layers and their combinations are not calculated for the given partially routed adders tree. Conversely, it is possible that an adder block has a very high cost on the bottom layer(s), however the upper layers are much less costy, and therefore the best solution is not found. 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 6-3. Theoretical numbers of considered solutions for different number of adder inputs N

Table 6-3 shows the theoretical number of possible solutions for the CS assuming that the calculation process is constrained only to layer 1, or layer 1 and 2. It can be seen that the total number of considered solutions has decreased significantly, however it is still unacceptable for the number of inputs *N* greater than 18.

6.4.3. Implementation Results

In this section results for the greedy algorithm (GrA), exhaustive search (ES) and constrained search (CS) algorithms are given. Table 6-4 shows the cost of the generated circuits by the GrA, ES and CS (for different thresholds *t*). Table 6-5 shows the calculation cost – the number of iterations needed to find the circuit.

Filter	No 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 6-4. The implementation costs (number of full or half adders) for different filters (see Table 6-1) and techniques

Filter	layer 1	layer 1,2	ES	CS (t=5)	CS (t=2)	CS (t=0)	CS (t=-1)
a	2 027 025	212 837 625	638 512 875	9 556 259	4 881 543	2 963 651	2 327 927
b	945	14 175	467 775	444 927	278 735	80 051	13 173
c	135 135	14 189 175	42 567 525	3 079 789	915 375	369 357	193 057

Table 6-5. The number of iterations for different filters and techniques

It can be seen from Table 6-4 and Table 6-5 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). For the threshold $t=-1$, only partial solution which is better than the best found is taken into consideration. This makes the CS ($t=-1$) similar to a GrA for which the step is constrained not only to a single adder (like for the GrA) but for all adders in the layer. Besides for the CS, it is always possible to undo a selection if the upper layers cost is high and therefore the overall cost of a new solution is higher than the best previously found cost. By the increase of threshold t, the number of considered solutions is growing. For $t=0$, not only the best but also all solutions on the same partial cost are also considered. This however increases the number of iterations but very slightly influences the overall results. Similar results are obtained for $t=2$ and $t=5$.

It should be noted that the GrA behaves more poorly for filters for which subtraction is implemented (for negative coefficients, examples a, c) as this algorithm deals with subtraction and addition in the same way.

6.5. Simulated Annealing (SA)

6.5.1. Principle

The principle behind the SA [Aar89, Kir83] is analogous to what happens when metals are cooled at a controlled rate. The slowly falling temperature allows atoms in the molten metal to line themselves up and form a regular crystalline structure that has high density and low energy. In the SA, the value of an objective function which we want to minimise, is analogous to the energy in a thermodynamic system. At high temperatures, SA allows function evaluations at faraway points and it is likely to accept a new point at higher energy. At low temperatures, SA evaluates the objective function only at local points and the likelihood of it accepting a new point with higher energy is much lower.

The SA algorithm, implemented for optimising adders structures, employs the following steps:

Objective function calculates the cost C of the circuit for a given adders tree.

Annealing Schedule regulates how rapidly the temperature T goes from high to low values, as a function of iteration counts. In the considered case, the starting temperature T_1 equals the cost of a 2-bit wide adder C_{A2}, the stopping temperature T_S equals ¼ of the cost of a 1-bit wide adder $C_{A1}/4$. In every iteration, the temperature T_i is decreased according to the following equation:

$$T_{i+1} = \eta \cdot T_i \qquad (6\text{-}13)$$

where $\eta = (\frac{T_1}{T_S})^{1/S}$, *S- the number of iterations.*

Generating a new adders structure – obtained by randomly selecting two adders on the same layer; i.e. randomly selecting a first adder (or input to the adders block) from all adders and randomly selecting a second adder from adders at the same layer as the first adder. Examples of possible modification are given in Figure 6-3.

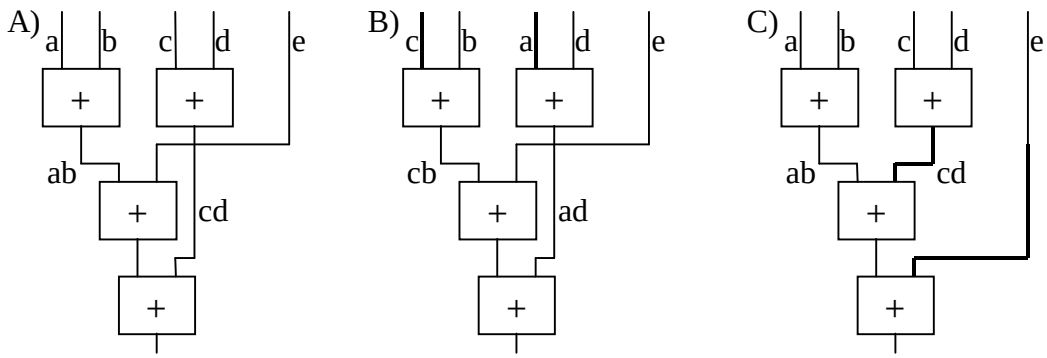

Figure 6-3. Examples of possible one-step modifications: A) an initial circuit, B) C) the modified circuit A.

Modifications of the circuit are constrained by temperature T_i. In the conventional SA, also known as the Boltzmann machine, the generating function which specifies the change of the input vector, is a Gaussian probability density function [Jan97]. In our approach, the number of possible solutions is finite therefore the Gaussian probability function is useless. An alternative solution is defining a move set [Mau84], denoted by M(x), as a set of legal points available for exploration. However, constructing the move set is rather computationally demanding task thus not implemented. In this approach, two adders are selected randomly (but at the same adders layer) and then a local acceptance function (LAF), which is further described in the next paragraph, is calculated. The LAF differs from the (global) acceptance function as it takes under consideration only the cost of the two involved adders before and after the modification. If the modification is not accepted locally, the change is rejected and the next modification is randomly generated (the iteration counter and temperature are not affected in this step).

Acceptance function. After a new network of adders has been evaluated, the SA decides whether to accept or reject it based on the value of an acceptance function *h( )*. The acceptance function is the Boltzmann probability distribution:

$$h(\Delta C, T) = \frac{1}{1 + \exp(\Delta C / T)} \tag{6-14}$$

where: $\Delta C = C_{i+1} - C_i$ *- the difference of the adders block cost for the previous and current adders tree.*

The new circuit is accepted with probability equal the value of the acceptance function.

6.5.2. Implementation results

The result for the SA, for different circuits are given in Table 6-6. For filter *a*, the result equals *103* (FAs/HAs), and is the best possible – the same as for the ES. This result is obtained already for 1000 iterations. For filter *c*, the cost equals *123*, the same as for the ES, and was obtained already for 30k iterations; for the CS, the result is *126* even for more than 3M iterations. It should be, however, noted that the computation cost of a single iteration is lower for the CS than for the SA. This holds as for the CS and ES, the change in the circuit is well-defined and usually constrained only to the upper layers of the adders and therefore only a part of the circuit has usually to be re-calculated. For the SA, the change is done randomly and on every part of the circuit, therefore cost of the whole circuit has to be recalculated. The lower calculation cost for the CS and ES, does not, however, compensate much greater number of iterations required to obtained the same result. Consequently, the overall calculation cost of the CS is usually greater than for the SA, however for small circuits for which the calculation cost is very low, the CS and ES are good alternatives to the SA.

Ex.	GrA	SA 1k	SA 30k	SA 1M	ES
a	111	93±0	93±0	93±0	93
c	128	126.9 ±0.3	125 ±1.4	123 ±0	123
d	413	394 ±3	385 ±1	382 ±1	-
d (wlaf)	413	398 ±4	382 ±1	380 ±1	-
e	1358	1346 ±10	1299 ±3	1292 ±4	-
e (wlaf)	1358	1341 ±9	1293 ±4	1283 ±4	-
f	3730	3702 ±20	3338 ±14	3245 ±6	-
f (wlaf)	3730	3706 ±13	3419 ±13	3296 ±8	-

Table 6-6. The circuit costs for the GrA, ES, and the SA for different number of iterations; wlaf – without local acceptance function

The final circuit (obtained in the lowest temperature) is often not the best one. Therefore, the best-obtained circuit is every time stored as the best result; this increases calculation cost insignificantly but allows for substantial algorithm improvements.

Table 6-6 shows also the results when local acceptance function (LAF) is not implemented (option: wlaf). Calculating local cost before and after the modification, insignificantly influences the total calculation cost and the LAF usually rejects solutions which are unlikely to generate a good global result. Conversely, the LAF constrains search space and therefore may cause some good solutions to be omitted. This is often the case for relatively small adders circuits and for large number of iterations. For example, it can be seen from Table 6-6 that not implementing the LAF gives better results for circuit *d* and for a small number of iterations, however spoils results for more complicated circuit *f*.

6.6. Genetic Programming (GP)

Genetic Programming (GP) [Koz92, Gol89] is an optimisation method based loosely on the concept of natural selection and evolutionary process. Major components of the GP include: encoding scheme, fitness evaluation, parent selection, crossover operation and mutation operators, these are approached next.

6.6.1. Encoding scheme

Encoding scheme transforms gene representation into a problem specific representation. In this approach, the adders tree is represented directly using two vectors of integers, which is typical rather for the Genetic Algorithm [Mic92]. Each adder occupies one entry in each vector. The entry specifies the index of the adder or input (from the lower layer) which is connected to the considered adder. For example, *parent 0* in Figure 6-4 is represented in the following two vectors of integers:

considered adder	*24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13*
vector 0	*22, 19, 18, 13, 14, 16, 0, 3, 2, 4, 9, 7*
vector 1	*23, 20, 21, 17, 15, 11, 6, 8, 1, 5, 12, 10*

Initially, it might seem that the structure of adders can be defined giving only the order of adders block inputs (the bottom layer order), as the rest of the structure can be built straightforward by connecting two neighbour adders. This, however, is a special case when the number of inputs to the adder block is a power of two. Otherwise, there is an alone signal which cannot be paired and therefore must be fed directly (without addition) to the upper layer of the adder block. In the given example for *parent 0*, it is the case for the input 11 for the first layer and signal 18 for the second layer. These alone signals complicate the adders structure, and cause that the structure of upper layers must be also included into gene coding.

6.6.2. Fitness evaluation

Fitness evaluation is based on evaluation of the cost (area) of a given adders tree.

6.6.3. Selection

After fitness evaluation, a new generation is produced from the current generation. The selection operation determines which adder will survive, based on the fitness value – the lowest cost of the adder, the greatest survival probability. It this approach, the modGA algorithm [Mic92] has been implemented as it has proved to surpass the classical genetic algorithm [Mic92]. In the modGA, in every generation we select independently *(p-r)* chromosomes to survive unchanged with the probability proportional to the scaled fitness f_i' which is obtained as a linear scaling of the area f_i occupied by the adder block:

$$f_i' = a \cdot f_i + b. \qquad (6\text{-}15)$$

Parameters *a* and *b* are calculated independently in every generation to satisfy the following equations:

$$a \cdot f_{\min} + b = 1 \qquad (6\text{-}16)$$

$$\sum_{i=1}^{p} (a \cdot f_i + b) = p - r \qquad (6\text{-}17)$$

where: f_{min} – the fittest (minimum cost) chromosome, *p* –population size, *r* – number of chromosomes determined to die $r<p/2$;

The eq. 6-16 preserves the fittest individual with the probability equal *1*. Eq. 6-17 causes that on average *(p-r)* chromosomes are selected to survive in a single wheel spin (a single wheel spin - every chromosome is picked to survive with probability f_i' only once). In this approach, the wheel spins until *(p-r)* chromosomes are selected and on average, a single wheel spin is required.

The *r* chromosomes selected to die are replaced by new ones, which are produced in either crossover or mutation. Consequently, the following equation holds:

$$r = c + m \qquad (6\text{-}18)$$

where: c- number of new chromosomes produced in the crossover operation, m- number of new chromosomes produced in the non-overwriting mutation operation (see mutation operation). In this approach p= 12, r= 5, c= 4, m= 1.

6.6.4. Crossover

Crossover is applied to randomly selected pairs of parents. The structure of the adders tree seems very similar to the commonly used tree graph structure used for scheduling and

partitioning [Mic92] or finding the optimal operation tree [Ayt95]. However for the adders block, the structure of the tree is strongly constrained. Therefore, the crossover operation implements a procedure, which generates only a valid structure of the adders tree (no repair procedure is required).

Two different options of crossover have been implemented. The first one (option A) attempts to copy as much as possible from the parents (considering actual parents structure and therefore disregarding the adder indices) and then employs a greedy algorithm (simplified version of the GrA described in Section 6.3) to route unconnected adders. In the second option (option B), the offspring copies the structure of the first parent and implements changes similar as for the SA, however changes are applied according to the structure of the second parent. In this option only indices are considered. These options are described below.

Crossover option A

In this option, an offspring inherits one (or all but the one) branch of adders from the first parent. For an example given in Figure 6-4, *offspring 0* inherits from *parent 0*, the adder structure: 20, 14, 15, 9, 12, 4, 5. Then, *offspring 0* inherits as much as possible from the second parent. In the given example, *offspring 0* can copy only adders 13, 2, 11; 16, 6, 7 and 17, 8, 0 from *parent 1*. Unfortunately, the whole adder structure may not be obtained directly from the parents; as some connections copied from the first parent, conflict with connections in the second parent. For the given example for *offspring 0*; from *parent 0*, the adder structure: 14, 9, 12 is copied; this makes impossible to copy the adder structure 14, 1, 9 from *parent 1* as *input 9* is already connected.

It should be noted that the crossover algorithm considers only indices of inputs on the bottom layer (in the given example: inputs 0-12) and how they are connected going up to the top layer (the indices of the adders on the upper layers are disregarded). Consequently, pairing the upper layers adders can be achieved if all adders on the lower layers can also be paired. Therefore, a single connection that cannot be achieved on a bottom layer, prevents the adders on the upper layers from being connected. This causes that the structure of adders on the upper layers is seldom inherited from the parents. Nevertheless, the offspring inherits only connections which existed in either of the parents.

This approach causes that the effectiveness of the algorithm strongly depends on crossover points; that is how many adders are copied from the first parent. Consequently, a crossover parameter is included into the gene-coding scheme, which allows this parameter to be optimised together with the adders structure during the evolutionary process. The crossover parameter defines from which adders layer a randomly chosen branch of adders is copied

from the first parent to the offspring. For example in Figure 6-5, *offspring 0* inherits adders structure: 20, 14, 15, 9, 12, 4, 5; i.e. the branch of adders beginning from the adder 20 (adder 20 is on the layer 2). Besides, the crossover parameter defines if one branch of the adder is inherited or all but one branches are inherited. For the given example, for the offspring 0, only one branch of adders is inherited from *parent 0*; for *offspring 1*, all but the one branches of adders are inherited from *parent 1*, i.e. except adders: 20, 14, 1, 9, 3.

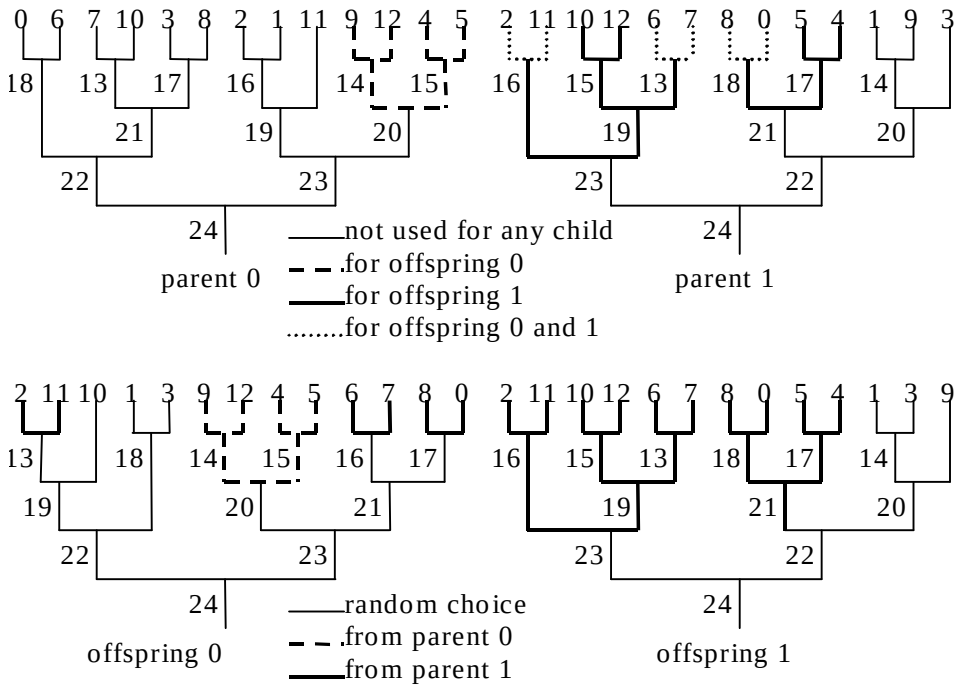

Figure 6-4. An example of the crossover operation for option A

The implementation results showed that in old generations, the crossover parameter is the same for all chromosomes and is equal: copy one branch beginning from the next to the last adder, i.e. copy half of the adders structure. Consequently, the crossover parameter is not longer included into gene-coding scheme and the crossover point is fixed to the half-adders-block copping.

Crossover option B

In this option, the crossover operation is very similar as for the Simulated Annealing presented in the previous section. The difference is that for the SA a modification is made in a random way, however for the GP, the modification is carried out with respect to the structure of the second parent. An example of the crossover operation is given in Figure 6-5. The crossover operation consists of three steps:

1. Randomly select a common crossover signal (in the given example: *signal 0*).

2. Find swapped signals which are paired with the common signal (*signal 1* for *parent 0* and *signal 2* for *parent 1*). In the case when the common signal is an alone signal (is not paired), the alone signal is chosen.
3. Swap signals found in the previous point.

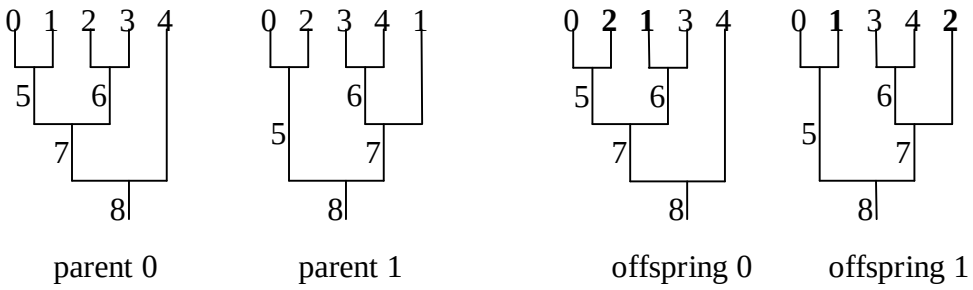

Figure 6-5. An example of the crossover operation for option B

It should be noted that the common crossover signal can be selected on any layer of the adder (except the top layer, which is a trivial case and therefore skipped). Besides, indices of signals on the upper layers for different parents may not correspond to each other, in the sense of the real adders structure. For example in Figure 6-5, *signal 5* in *parent 0* ($S_5 = S_0 + S_1$) is different from *signal 5* in *parent 1* ($S_5 = S_0 + S_2$). This means that swapping the upper layers adders often disregards the real connections of the parents as indices of these adders are assigned more or less in a random way. Therefore, to improve the algorithm the indices of the upper layer adders are assigned (sorted) according to the increase of the input index (the lower index of two inputs). For example, for *parent 1* in Figure 6-5, *adder 5* has the lowest index on the layer 1 because *input 0* is the lowest input index on the bottom layer. Sorting adder indices improves correlation between parents, nevertheless, the index of the upper layers adder in one parent often represents different addition than in the second parent. This means that swapping is often achieved in a random way, especially when structure of parents differs significantly. It should be noted that for large adders structure the relationship between index number and its structure is decreasing, therefore for a large adders tree this crossover method is not recommended (see Table 6-8).

The change made by a single swapping is rather insignificant therefore, usually 1-3 similar swapping operations are performed to obtain an offspring.

The idea behind the modGA is that the algorithm avoids leaving the exact copies of the same chromosomes in the new population, which may still happen accidentally by other means but is very unlikely [Mic92]. However, experiments proved that both option A and B can produce an offspring identical to its parents especially if the parents are very similar. This

causes that several copies of the same parents exist in the population, which deteriorates the result. Therefore, to improve the modGA results, in the case when an offspring is an exact copy of the parent(s) (or differs insignificantly), the mutation is performed on the offspring. Therefore, in this approach, the additional mutation operation prevents from obtaining the exact copy of the parent during crossover operation and prohibits super-individuals from dominating the population. It should be noted that in the nature, a specimen avoids to mate with its relatives in order not to produce similar gene offsprings. Moreover, similar solution has been proposed by Maudlin [Mau84], where the mutation rate is changed according to the degree of homogeneity of the chromosomes. The disadvantage of Maudlin's approach is that it requires additional computation time to evaluate the degree of the homogeneity. In this approach, however, detecting crossover diversity increases computation time insignificantly as it is associated directly with the crossover operation.

6.6.5. Mutation

Crossover operation can only explore the current gene potential therefore, a mutation operation is included to spontaneously generate new chromosomes. In our approach, mutation is carried out in a similar way as for the SA, i.e. by swapping two adders on the same layer.

Two different mutation options has been implemented:

1. parent non-overwriting mutation (NOM)
2. parent overwriting mutation (OM)

The NOM is associated with the modGA selection operation as the number of new chromosomes r generated in each generation, includes the number of new chromosomes created during mutation m. Therefore, randomly picked chromosomes (from the surviving chromosomes) are copied and the mutation is performed on the copy of the chromosomes.

The OM is carried out in the standard way, i.e. every unchanged chromosome is mutated with probability p_m.

Two different mutation options have been implemented to allow proper population development. In the case when only the OM is implemented, the high mutation ratio prohibits super-individuals to grow as often probability of generating an offspring which fitness is comparable to the parent is very low – lower than mutation rate. Therefore, the best solution is often generated rather in a random way then based on the genetic algorithm properties and the fitness of the latest generations is very often far from the best solution fitness. Conversely, low mutation ratio causes mutation to have insignificant influence on the result and therefore deteriorates the result. Employing only NOM causes that super-individuals are always copied to the new generation without any change (the fittest chromosome is selected with probability

equal 1) and therefore the population is dominated by the super-individuals which may be in a local minimum. Consequently, the best solution is a combination of the NOM and OM. Implementation results, given in Table 6-7, confirm this assumption.

circuit	p_m= 0, m= 1	p_m= 0.2%, m= 1	p_m= 10%, m= 0
d) iter= 6k	393 ±2	392 ±5	391 ±2
d) iter= 200k	392 ±4	388 ±5	381 ±1
e) iter= 6k	1302 ±3	1303 ±2	1317 ±4
e) iter= 200k	1297 ±3	1292 ±2	1297 ±2
f) iter= 6k	3580 ±7	3580 ±6	3616 ±8
f) iter= 200k	3454 ±5	3455 ±4	3508 ±16

Table 6-7. Implementation results for different mutation solutions: only NOM, combination of the NOM and OM, and only OM; for crossover: option A

6.7. Implementation results

Table 6-8 shows implementation results for the different algorithms. The number of iterations for the GP and SA is selected so that the calculation cost was roughly the same. It can be seen from Table 6-8 that the SA solution gives usually the best results and the crossover option A is a better solution in comparison to option B, especially for more complicated circuits. Furthermore for option B and circuit f, the implementation results are even so poor that the GrA initial solution is the best-found solution for up to 6k iterations.

Circuit) technique	# iterations (GP/SA)		
	200/1k	6k/30k	200k/1M
d) option A	399 ±3	392 ±5	388 ±5
d) option B	412 ±2	392 ±4	387 ±5
d) SA	394 ±3	385 ±1	382 ±1
e) option A	1341 ±6	1303 ±2	1292 ±2
e) option B	1358 ±0	1357 ±1	1297 ±4
e) SA	1346 ±10	1299 ±3	1292 ±4
f) option A	3713 ±7	3580 ±6	3455 ±4
f) option B	3730 ±0	3730 ±0	3645 ±26
f) SA	3702 ±20	3338 ±14	3245 ±6

Table 6-8. Implementation results for different options of the GP, and for the SA, for different number of iterations

6.8. Conclusions

In this chapter, thorough analysis of adders tree as a part of the FIR filters has been presented. Complex input parameters of the adders block have been considered: inputs range (not only input width), input shifts and even the correlation between inputs. Consequently, finding an optimal network of the adders tree is a complex task which has been investigated. Different approaches: greedy algorithm, exhaustive search, simulated annealing and genetic programming have been implemented and the results given. Consequently, the 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. For the CS, each layer of the addition is considered, in some degree, separately. The CS checks a lower number of solutions however the number 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. Further, the Simulated Annealing has been implemented. 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. Therefore for $n \leq 8$ the ES solution should be implemented. For $n \geq 9$, the ES goes through at least 42 525 solutions therefore the SA should be rather used.

The Genetic Programming is another design-approach which has been implemented. The structure of the adder block is strictly defined and therefore the crossover procedure has to copy parts of the parents in such a way that the child has a proper structure. This, however, is difficult to be achieved and therefore some parts of the offspring has to be routed to satisfy the adder block constrains rather then to copy a structure of the parents. Two different crossover procedures have been implemented. Nevertheless, at the same computation time, the GP usually gives worse results than the SA. This conclusion is similar as presented by McMahon [McM95] for scheduling problems and shows that for some problems the SA is beneficial in comparison to the GP.

7. Conclusions

The design complexity of nowadays systems stimulates a strong demand for design automated tools which are able to generate a wide range of implementations and support multiple parameters. The AuToCon is an example of such a system. The AuToCon takes different input bit-width, convolution kernel size and coefficient values, and generates a circuit searching through multiple architectural solutions in order to find the best circuit. Different speed options can be defined by the value of the pipeline parameter. This thesis presents a novel synthesis approach for which FPGA resources are user-defined, which allows for smooth migration from one device to another. Furthermore, by changing cost-relations between the FPGA resources, different architectural solutions are obtained, as the AuToCon searches for the lowest cost architecture. For variable coefficient systems, reconfiguration time is also a crucial factor that influences the architecture and design cost, which has been thoroughly studied.

The AuToCon explores FPGA device-specific features, e.g. for ASICs LUT-based multiplication or convolution is very seldom adopted, addition is carried out employing carry-save adders, while for FPGAs, ripple-carry adders are the best solution and the LM may give the best result. The AuToCon minimise the need for knowledge of low-level details, provided that the FPGA resources and their structural, VHDL models have been once supplied for the FPGA family. The system user has to only enter convolution kernel size, coefficient values, reconfiguration option and the pipelining parameter. Nevertheless in some projects (e.g. when almost all CLBs are occupied and still lot of large memory blocks are available), changing cost-relation between FPGA resources (which requires rather the low-level knowledge) may improve usage of FPGA resources. It should be noted that generated circuit is produced automatically mostly within the time of a second. Summing up, the thesis has been proved.

This thesis presents the AuToCon, however detailed description of the whole AuToCon system is outside the scope of this thesis. Only the most attractive fragments have been hereby characterised. It should be however noted that most of the author research time has been spent on development of the tangled and complicated automated system.

As a part of the research several novel architectural solutions have been developed, such as modified CSD conversion algorithm, usage of different memory modules for the LM and

IDAC, advance optimisation techniques for LMs, the Dual Port DKCM, extensive usage of Multiplierless Multiplication in FPGAs, Irregular Distributed Arithmetic Convoler, and trade-off between the IDAC and Multiplierless Convoler.

Chapter 2 reviews different computing machines implementing the convolution operation. General-purpose processors and DSPs are the best solution for relatively low computationally demanding processes. However, e.g. for real time high-resolution image convolution and large kernels, the microprocessor approach is inadequate. Furthermore, nowadays architectures of microprocessors are very sophisticated and further development of their complexity results in less and less significant computation speed-up. ASICs are an alternative solution, however, they suffer from long development time, high cost for prototyping and low volume production, and at last but not least low design flexibility. Conversely, FPGAs are more and more commonly implemented in regions originally reserved for DSPs or ASICs. Furthermore, FPGAs' density grow surpasses the counterparts' grow. As a result, there is a strong demand for design automation tools that speed-up design process for FPGAs. The presented automated tool is, therefore, a proposition for such a system.

Chapter 3 approaches the Constant Coefficient Multiplier (KCM) and different architectures performing the KCM, such as Multiplierless Multiplication (MM) and LUT-based Multiplication (LM); a comparison of these two techniques is also presented. Novel architectural solutions have been introduced, such as the Modified CSD conversion algorithm; usage of different memory modules for the LM together with the advance optimisation techniques: the LSBs Address Width Reduction, the Don't Care Address Width Reduction, and the Memory Sharing. Furthermore, the full search algorithm compares all possible solutions and the best solution for the given input parameters is taken.

Chapter 4 investigates multiplier architectures for which change of the coefficient is a feasible design factor. Furthermore, a novel architecture of Dynamic Constant Coefficient Multiplier (DKCM) with Dual Port memories is studied. For this multiplier the multiplication result is corrupted during a change of the coefficient, however, the corruption is well defined and may be acceptable in adaptive systems. Similarly like in Chapter 3, usage of different memory modules is included into the full search algorithm. However the search space is enlarged by additional DP memories, or single port memories and multiplexers trade off.

A convolution, basically, can be carried out as a sum of products, however, its modification, LUT-based Convolution, for which adders within LMs and the final adder are combine together, gives better results. Chapter 5 presents also the (parallel) Distributed Arithmetic Convoler, and its novel modification, the Irregular Distributed Arithmetic Convoler. The IDAC is an architectural solution which combines both the DAC and LC in

such a way that the resultant circuit is at the lowest cost. A novel optimisation algorithm, which finds the optimal IDAC circuit, has been also presented. The Multiplierless Convolution is an alternative solution to the IDAC. The MC is similar to the Multiplierless Multiplication, however more sophisticated methods are involved, such as substructure sharing between different coefficients or pipeline optimisation. Furthermore, the novel algorithm trading-off between the MC and IDAC has been developed. As a result, each individual coefficient can be implemented employing either the MC or IDAC.

Chapter 6 describes optimisation techniques for adders tree. Different techniques such as: Greedy Algorithm, Exhaustive Search, Simulated Annealing and Genetic Programming have been implemented. As a result, Greedy Algorithm is the quickest, however better results can be obtained for more complex algorithms. The ES is, therefore, recommended for number of inputs $n \leq 8$ because computation requirements for such an adders block are rather low, and the SA is recommended for $n \geq 9$.

In conclusion, the implementation results proved that the AuToCon outperforms comparable automated tools.

Suggestions for further work

The AuToCon takes into account a great number of parameters. However, additional parameters can be still defined. For example, the AuToCon implements only bit-parallel arithmetic, and in some applications a bit-serial [Hes96], or middle-way between bit-parallel and bit-serial architecture [Pas01] can handle the design requirements. The later solution may be used instead of the lowly pipeline (large value of the pipeline parameter p) architectures.

Conversely, in some cases even fully pipelined bit-parallel circuit cannot cope with high frequency requirements. In this case two or more parallel filters should be implemented. However additional area-reduction is obtained by employing reduced-complexity parallel FIR filters [Par97].

For multipliers, the best-possible circuit (under the given design conditions) is generated by the exhaustive search algorithm. However, a convolution circuit is much more complex and requires heuristics. In this thesis mostly greedy algorithm has been implemented. However better priority queues for the greedy algorithms might be found. Besides more sophisticated optimisation techniques should be also considered (similar like for the adders tree). In order to keep AuToCon computation requirements at low level, trade-off between the IDAC and MC is based on the estimated cost for a convoler. However, the AuToCon is

relatively quick (in most cases the convoler is generated within the time of a second), and therefore the trade-off algorithm might be based on the actual circuit cost.

FPGAs evolve and new FPGA resources are introduced. For example, recently introduced Virtex II family incorporates larger BSRs, and greater variety of small (distributed) memory modules. Because FPGA resources are the input parameters to the AuToCon, these new resources can be quickly included while searching for the optimal architecture. Nevertheless, Virtex II incorporates also new dedicated 18×18 fully functional multipliers, which makes DKCMs less attractive than for Virtex. In the case of the constant coefficient option, the KCM occupies less chip area, and therefore the standard KCM can be still employed especially when the width of the multiplier is short. The AuToCon might also map the 18×18 multiplier as a virtual $2^{18}\times 36$ memory, consequently more complex coefficients (containing large number of non-zero CSD bits) might be implemented in the built-in multipliers and the rest of the coefficient in the CLBs or BSRs. However this requires additional changes in the AuToCon and analysing implementation results.

The AuToCon takes little attention of design routing. An assumption is made that a place and route program can do the job. However routing optimisation should be considered in the next step especially for the substructure sharing and optimisation of the adders block.

Realisation of the FIR filters uses inputs and coefficients values directly, which requires full-length multipliers and adders. However, differential coefficients and inputs method [Cha00] might be implemented. This method uses differential coefficients to multiply with inputs and compensates the effect of differential coefficients by adding the product of the previous computation. Since differential coefficients have shorter word length, the resulting design can use shorter word length. Similar effect is obtained for differential input, when the range of the difference between two consecutive inputs is smaller than the original input.

For standard pipelining designs, the clock frequency is constrained by the maximum delay time between consecutive flip-flops. However wave pipelining [Boe98] might be implemented for which minimum clock period is limited by the difference between the maximum and minimum path delay plus the clock skew, the rise/fall time and the setup time of the registers. The difference between maximum and minimum delay can be further reduced by a place and route program.

Ripple-carry adders are assumed to be the best solution. Nevertheless for long adders the carry propagation delay can significantly slow down the design throughput. Consequently, a splitting of long adders into several parts (applying the hybrid pipelining as for FPGAs and ASICs) is suggested. The next design step might be automatic optimisation of the adders tree for which long adder splitting is implemented.

The AuToCon might be included to a hardware/software co-design system [Sta97] e.g. in the RACE [Smi96]. Consequently, the system might automatically detect convolution loops in e.g. C-language, and implement the most computationally demanding task in a FPGA rather than in a microprocessor. In the course of this work, different architectures have been compared, and FPGAs are the most promising architectural solution for high-speed convolution. Therefore such a system may be a fundamental solution for tomorrow's systems. Furthermore the AuToCon might be included into an adaptive system for which dynamic change of the coefficients or even the kernel size is allowed.

The author has designed and developed a general-purpose FPGA board with three XC4010E up to XC4025E chips, on board SRAM memory and the PCI interface. A more detailed description of this board is outside the scope of this thesis. Consequently, a range of real time image convolutions has been implemented. Nevertheless, the AuToCon allows for implementation of different algorithms: e.g. part of artificial neural networks, etc. Furthermore, the next design step might be development of a system that will automatically generate neural networks. The most essential advantage of such a system (in comparison to the ASIC solutions) might be a dynamic change applied to not only the weights but also to the network structure while learning process is in progress. The system may exploit the fact that most of weights might be equal zero and therefore need not be implemented.

Appendix A.
Brief description of the AuToCon

The Automated Tool for generation 2-dimentional Convolers implemented in FPGAs (AuToCon) basically consists of two segments:

- C++ program which reads input parameters from file ‘param.txt’ and VHDL-like template files and generates the final VHDL (text) files.
- Predefined VHDL files which describe FPGA implementation of the fundamental elements used in the design, such as RAMs, adders, etc. Besides discription of regular blocks is included, e.g. RAM programming unit.

Design flow is illustrated in Figure A-1. The AuToCon generates also a VHDL test bench, therefore the generated circuit can be automatically simulated, and a design error detected and reported.

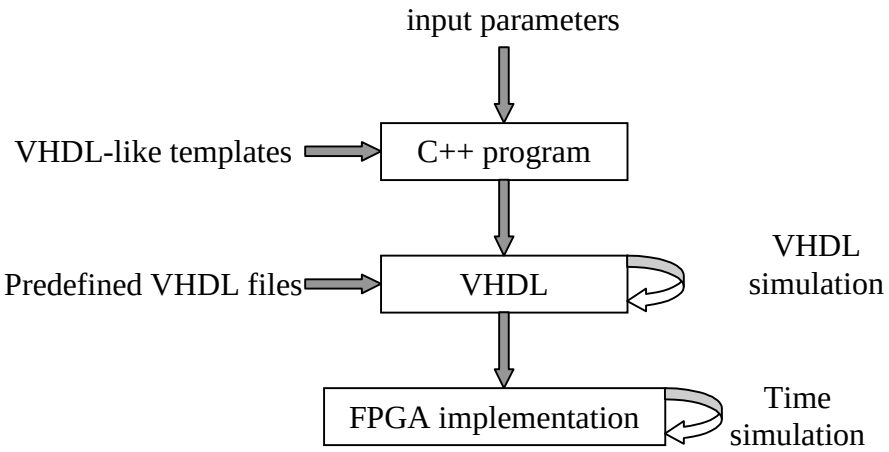

Figure A-1. Typical design flow

Input parameters

The following input parameters can be defined in ‘param.txt’ file:

min_din – minimum input value

max_din – maximum input value

ram – defines whether constant (ram=0) or dynamic (ram=1) circuit is implemented

coeff – coefficient value(s)

min_coeff – minimum coefficient value (applicable only for ram=1)

max_coeff – maximum coefficient value (applicable only for ram=1)

SizeCoeffX – horizontal kernel size

SizeCoeffY – vertical kernel size

SymmetryX – filter horizontal symmetry: 1- symmetry, 0 - no symmetry, -1 – asymmetry

SymmetryY - filter vertical symmetry: 1- symmetry, 0 - no symmetry, -1 – asymmetry

SymmetryP – filter point symmetry: 1- point symmetry, 0- no symmetry

pipeline – defines maximum number of logic elements between two subsequent pipelining flip-flops

InsertRegistersIn – insert flip-flips at the input of the convoler/multiplier

InsertRegisterOut – insert flip-flops at the output of the convoler/multiplier

InsertCe – insert clock enable signal to make the circuit inactive for clock cycles, this signal might be required e.g. during blank cycles when the image is inactive

SimulationLength – the number of simulation cycles. The smaller number, the shorter simulation time but an design error is less likely to be detected

clock – period of the clock (needed for time simulation only)

LineWidth – external line buffers length, needed for simulation only; during implementation line buffers are external blocks inserted by the designer (line buffers of any length can be automatically generated by e.g. Core Generator [Xil99a]). This allows to perform e.g. sum-of-products operation when *SizeCoeffX=1* and *SizeCoeffY* defines the number of products.

FPGA resources declaration

Flip-flops

CostFF – defines cost of a single D-type flip-flop

Adders

CostAdd – defines cost of the adder separately for different adders widths

Memory

Mem initialisation of memory entity

No memory identification number (the same as used during VHDL entity declaration)

= separator

MemorySize memory size declaration (k - 1024 and M= 1024k is accepted)

× separator

DataWidth data bus width declaration (must be power of 2 for multipliers)

DP/SP type of memory Dual Port/ Single Port

DataWidthDP data width of the second port (for DP, should be the same as *DataWidth*)

S/A (a)synchronous memory reading (no flip flops need to be inserted) (writing is always synchronous - or the RAM programming unit has to be modified)

cost cost of the single block

Example:

Mem1= 16x1SPA1 - 16×1 distributed memory for XC4000/Virtex, cost of memory equals 1 LE.

Communication between C++ and predefined VHDL files

C++ program generates VHDL files (C-VHDL) which describe a convoler mostly at structural domain. However, the C-VHDL files do not refer to any low-level structure of a FPGA, these files refers to the entities defined in the predefined VHDL files. Consequently the user can define the final structure of the adders or memory blocks, etc. in the predefined VHDL files. Furthermore, the pre-defined VHDL files can refer to pre-synthesised modules or modules which have been generated using different design entry methodology, e.g. modules generated by Core Generator [Xil99a], schematic elements.

Example of convoler design given in Figure 5-8.

Input parameters (param.txt file)

```
CostMux2= 0 CostFF= 10
CostAdd=
10      20      30      40      50      60      70      80
90      100     110     120     130     140     150     160
170     180     190     200     210     220     230     240
250     260     270     280     290     300     310     320
330     340     350     360     370     380     390     400
410     420     430     440     450     460     470     480

Mem0= 2x1SPA0
Mem1= 16x1SPA10
Mem2= 32x1SPA20
Mem3= 16x1DP1A20
LineWidth= 16 InsertCe= 0
clock= 50ns SimulationLength= 50
InsertRegistersIn= 1  InsertRegistersOut= 1
pipeline= 100 ram= 0
SymmetryX=0 SymmetryY=0 SymmetryP= 0
SizeCoeffX= 8     SizeCoeffY= 1
max_din= 15 min_din= 0
coeff=
59      183     162     -7      -48     12      9       2
```

A fragment of the predefined VHDL file, the description of the adder block.

This VHDL code can be edited to define a different adder structure.

```
-- Addition dout<= din1 + din0;
library IEEE;
use IEEE.std_logic_arith.all;
```

```
use IEEE.std_logic_1164.all;

entity adder is
   generic(width_din0: integer:= 4; -- input width of the first input
           width_din1: integer:= 4; -- input width of the second input
           width_dout: integer:= 5; -- input of the output
           sign: integer:= 0; -- encoded sign of the first and second input and operation (adder, subtractor)
           shift_din0: integer:= 0); -- shifting of  the first input to the left
   port (din0: in unsigned(width_din0-1 downto 0); -- the first input
      din1: in unsigned(width_din1-1 downto 0); -- the second input
      dout: out unsigned(width_dout-1 downto 0)); -- the output
end adder;

architecture adder_arch of adder is
           constant width_din0sh: integer:= width_din0 + shift_din0; -- width of the first input after shifting
           signal din0sh: unsigned(width_din0+shift_din0-1 downto 0); -- declaration of the shifted din0 signal
           signal din0u, din1u: unsigned(width_dout-1 downto 0); -- input signals for which MSBs are fill width 0s or 1s
begin
           -- shifting din0 signal if shift_din0
 din0sh(width_din0sh-1 downto shift_din0)<= din0;
 shift1:
 if shift_din0 > 0 generate
           din0sh(shift_din0-1 downto 0)<= conv_unsigned(0, shift_din0); -- fill with zeros
 end generate;


 i0g: if width_din0sh > width_dout generate -- the width of din0 is greater than the width of dout (unusual case)
           din0u(width_dout-1 downto 0)<= din0sh(width_dout-1 downto 0);
 end generate;
 -- filling MSBs of din0 with either 0s or sign
 i0a: if width_din0sh <= width_dout generate
  din0u(width_din0sh-1 downto 0)<= din0sh;
  i0: if width_din0sh<width_dout generate -- fill MSBs with either zeros or sign
    i0f: for i in width_din0sh to width_dout-1 generate -- for every not assigned MSB
           i0u: if sign=0 or sign=2 or sign=4 or sign=6 generate -- din0 is unsigned
                   din0u(i)<= '0';
           end generate;
           i0s: if sign=1 or sign=3 or sign=5 or sign=7 generate -- din0 is signed
                   din0u(i)<= din0sh(width_din0sh-1); -- sign bit
           end generate;
  end generate; end generate; end generate;


 i1g: if width_din1 > width_dout generate -- unsual case - the width of the adder is reduced
           din1u(width_dout-1 downto 0)<= din1(width_dout-1 downto 0);
 end generate;
 -- filling MSBs of din1 with either 0s or sign
 i1a: if width_din1 <= width_dout generate
  din1u(width_din1-1 downto 0)<= din1;
  i1: if width_din1<width_dout generate -- fill MSBs with either zeros or sign
   i1f: for i in width_din1 to width_dout-1 generate -- for every not assigned MSB
           i1u: if sign=0 or sign=1 or sign=4 or sign=5 generate -- din1 is unsigned
                   din1u(i)<= '0';
           end generate;
           i1s: if sign=2 or sign=3 or sign=6 or sign=7 generate -- din1 is signed
                   din1u(i)<= din1(width_din1-1); -- sign bit
           end generate;
  end generate; end generate; end generate;

 -- insert adder
 a: if sign<=3 generate
           dout<= din0u + din1u;
 end generate;

 -- insert subtractor
 s: if sign>=4 generate
           dout<= din0u - din1u;
 end generate;
end adder_arch;
```

An example of VHDL-like templates

The C program reads the following file in order to generate an adders tree description (e.g. entity ass2, ac3 or al). The final adders tree description is combination of the VHDL-like template and text inserted by the C program. The procedure is as follows. The C program reads the VHDL template file and copies directly fragments of between '#' symbol. Whenever '#' symbol is found, the C program inserts variable (depending on the AuToCon input parameters) text.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;

entity # is
  port(clk, ce: in std_logic;
#            dout: out unsigned(# downto 0));
end #;

architecture #_arch of # is
  component adder
    generic(width_din0: integer;
            width_din1: integer;
            width_dout: integer;
            sign: integer;
            shift_din0: integer);
    port (din0: in unsigned(width_din0-1 downto 0);
        din1: in unsigned(width_din1-1 downto 0);
        dout: out unsigned(width_dout-1 downto 0));
            end component;
  component  ffg – flip flops
            generic(width: integer);
            port(clk, ce: in std_logic;
                    din: in unsigned(width-1 downto 0);
                    dout: out unsigned(width-1 downto 0));
            end component;

  type arr is array (# downto 0) of unsigned(# downto 0);
  signal ff_out, add_out, alone: arr;

begin
#
end #_arch;
```

The final VHDL description generated by the C program

Adder D9 in Figure 5-8.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;

entity ass2 is
  port(clk, ce: in std_logic;
    din0: in unsigned(3 downto 0);
    din1: in unsigned(3 downto 0);
    dout: out unsigned(9 downto 0));
end ass2;

architecture ass2_arch of ass2 is
 [...] components declaration

type arr is array (3 downto 0) of unsigned(9 downto 0);
  signal ff_out, add_out, alone: arr;

begin
```

```
  ff_out(0)(3 downto 0)<= din0; -- shift= 0, min= 0, max= 15, correl= -1, add
  ff_out(1)(3 downto 0)<= din1; -- shift= 5, min= -15, max= 0, correl= -1, subtract
   --level of logic= 0
  ff_out(0)(4)<= '0'; -- din1 is only negated
  add_out(2)(4 downto 4)<= conv_unsigned(0, 1);
  add2: adder
   generic map (width_din0=>1, width_din1=>4, width_dout=>5, sign=>4, shift_din0=>0)
   port map(din0=>ff_out(0)(4 downto 4), din1=>ff_out(1)(3 downto 0), dout=>add_out(2)(9 downto 5));
  add_out(2)(4 downto 0)<= ff_out(0)(4 downto 0);
  ff_out(2)(9 downto 0)<= add_out(2)(9 downto 0);

  dout<= ff_out(2);
end ass2_arch;
```

Adder D8 in Figure 5-8.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;

entity ac3 is
  port(clk, ce: in std_logic;
   din0: in unsigned(3 downto 0);
   din1: in unsigned(3 downto 0);
   dout: out unsigned(6 downto 0));
end ac3;

architecture ac3_arch of ac3 is
  [...] components declaration
  type arr is array (3 downto 0) of unsigned(6 downto 0);
  signal ff_out, add_out, alone: arr;

begin
  ff_out(0)(3 downto 0)<= din0; -- shift= 2, min= -15, max= 0, correl= -1, subtract
  ff_out(1)(3 downto 0)<= din1; -- shift= 0, min= 0, max= 15, correl= -1, add
   --level of logic= 0
  add2: adder
   generic map (width_din0=>2, width_din1=>4, width_dout=>5, sign=>4, shift_din0=>0)
   port map(din0=>ff_out(1)(3 downto 2), din1=>ff_out(0)(3 downto 0), dout=>add_out(2)(6 downto 2));
  add_out(2)(1 downto 0)<= ff_out(1)(1 downto 0);
  ff_out(2)(6 downto 0)<= add_out(2)(6 downto 0);

  dout<= ff_out(2);
end ac3_arch;
```

The final adder

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;

entity al is
  port(clk, ce: in std_logic;
   din0: in unsigned(9 downto 0);
   din1: in unsigned(9 downto 0);
   din2: in unsigned(3 downto 0);
   din3: in unsigned(3 downto 0);
   din4: in unsigned(3 downto 0);
   din5: in unsigned(3 downto 0);
   din6: in unsigned(3 downto 0);
   din7: in unsigned(3 downto 0);
   din8: in unsigned(3 downto 0);
   din9: in unsigned(10 downto 0);
   din10: in unsigned(1 downto 0);
   din11: in unsigned(7 downto 0);
   din12: in unsigned(1 downto 0);
   din13: in unsigned(2 downto 0);
   dout: out unsigned(13 downto 0));
end al;

architecture al_arch of al is
```

```
 [...] components declaration
type arr is array (27 downto 0) of unsigned(13 downto 0);
 signal ff_out, add_out, alone: arr;

begin
 ff_out(0)(9 downto 0)<= din0; -- shift= 0, min= -15, max= 480, correl= -2, subtract
 ff_out(1)(9 downto 0)<= din1; -- shift= 2, min= -15, max= 480, correl= -2, subtract
 ff_out(2)(3 downto 0)<= din2; -- shift= 6, min= 0, max= 15, correl= -1 (no correlation), add
 ff_out(3)(3 downto 0)<= din3; -- shift= 1, min= 0, max= 15, correl= -1, add
 ff_out(4)(3 downto 0)<= din4; -- shift= 0, min= 0, max= 15, correl= -3, add
 ff_out(5)(3 downto 0)<= din5; -- shift= 3, min= -15, max= 0, correl= -3, subtract
 ff_out(6)(3 downto 0)<= din6; -- shift= 0, min= 0, max= 15, correl= -4, add
 ff_out(7)(3 downto 0)<= din7; -- shift= 3, min= 0, max= 15, correl= -4, add
 ff_out(8)(3 downto 0)<= din8; -- shift= 1, min= 0, max= 15, correl= -1, add
 ff_out(9)(10 downto 0)<= din9; -- shift= 0, min= 0, max= 1293, correl= 13, add, min_cor= -720, max_cor= 2925
 ff_out(10)(1 downto 0)<= din10; -- shift= 3, min= 0, max= 3, correl= 13, add, min_cor= -720, max_cor= 1632
 ff_out(11)(7 downto 0)<= din11; -- shift= 3, min= 0, max= 225, correl= 13, add, min_cor= -720, max_cor= 1608
 ff_out(12)(1 downto 0)<= din12; -- shift= 7, min= 0, max= 3, correl= 13, add, min_cor= -768, max_cor= 0
 ff_out(13)(2 downto 0)<= din13; -- shift= 8, min= -3, max= 0, correl= -1, add
  --level of logic= 0
 add14: adder
  generic map (width_din0=>4, width_din1=>4, width_dout=>5, sign=>0, shift_din0=>0)
  port map(din0=>ff_out(4)(3 downto 0), din1=>ff_out(6)(3 downto 0), dout=>add_out(14)(4 downto 0));
 ff_out(14)(4 downto 0)<= add_out(14)(4 downto 0);

 add15: adder
  generic map (width_din0=>4, width_din1=>10, width_dout=>10, sign=>6, shift_din0=>1)
  port map(din0=>ff_out(3)(3 downto 0), din1=>ff_out(0)(9 downto 0), dout=>add_out(15)(9 downto 0));
 ff_out(15)(9 downto 0)<= add_out(15)(9 downto 0);

 add16: adder
  generic map (width_din0=>10, width_din1=>4, width_dout=>10, sign=>0, shift_din0=>0)
  port map(din0=>ff_out(9)(10 downto 1), din1=>ff_out(8)(3 downto 0), dout=>add_out(16)(10 downto 1));
 add_out(16)(0 downto 0)<= ff_out(9)(0 downto 0);
 ff_out(16)(10 downto 0)<= add_out(16)(10 downto 0);

 add17: adder
  generic map (width_din0=>2, width_din1=>10, width_dout=>10, sign=>6, shift_din0=>1)
  port map(din0=>ff_out(10)(1 downto 0), din1=>ff_out(1)(9 downto 0), dout=>add_out(17)(9 downto 0));
 ff_out(17)(9 downto 0)<= add_out(17)(9 downto 0);

 add18: adder
  generic map (width_din0=>4, width_din1=>4, width_dout=>5, sign=>4, shift_din0=>0)
  port map(din0=>ff_out(7)(3 downto 0), din1=>ff_out(5)(3 downto 0), dout=>add_out(18)(4 downto 0));
 ff_out(18)(4 downto 0)<= add_out(18)(4 downto 0);

 add19: adder
  generic map (width_din0=>4, width_din1=>2, width_dout=>5, sign=>0, shift_din0=>0)
  port map(din0=>ff_out(11)(7 downto 4), din1=>ff_out(12)(1 downto 0), dout=>add_out(19)(8 downto 4));
 add_out(19)(3 downto 0)<= ff_out(11)(3 downto 0);
 ff_out(19)(8 downto 0)<= add_out(19)(8 downto 0);

 add20: adder
  generic map (width_din0=>2, width_din1=>3, width_dout=>3, sign=>2, shift_din0=>0)
  port map(din0=>ff_out(2)(3 downto 2), din1=>ff_out(13)(2 downto 0), dout=>add_out(20)(4 downto 2));
 add_out(20)(1 downto 0)<= ff_out(2)(1 downto 0);
 ff_out(20)(4 downto 0)<= add_out(20)(4 downto 0);

  --level of logic= 1
 add21: adder
  generic map (width_din0=>5, width_din1=>11, width_dout=>11, sign=>0, shift_din0=>0)
  port map(din0=>ff_out(14)(4 downto 0), din1=>ff_out(16)(10 downto 0), dout=>add_out(21)(10 downto 0));
 ff_out(21)(10 downto 0)<= add_out(21)(10 downto 0);

 add22: adder
  generic map (width_din0=>7, width_din1=>5, width_dout=>8, sign=>3, shift_din0=>0)
  port map(din0=>ff_out(15)(9 downto 3), din1=>ff_out(18)(4 downto 0), dout=>add_out(22)(10 downto 3));
 add_out(22)(2 downto 0)<= ff_out(15)(2 downto 0);
 ff_out(22)(10 downto 0)<= add_out(22)(10 downto 0);
```

```
  add23: adder
   generic map (width_din0=>6, width_din1=>5, width_dout=>7, sign=>3, shift_din0=>0)
   port map(din0=>ff_out(17)(9 downto 4), din1=>ff_out(20)(4 downto 0), dout=>add_out(23)(10 downto 4));
  add_out(23)(3 downto 0)<= ff_out(17)(3 downto 0);
  ff_out(23)(10 downto 0)<= add_out(23)(10 downto 0);

   --level of logic= 2
  add24: adder
   generic map (width_din0=>11, width_din1=>11, width_dout=>12, sign=>2, shift_din0=>0)
   port map(din0=>ff_out(21)(10 downto 0), din1=>ff_out(22)(10 downto 0), dout=>add_out(24)(11 downto 0));
  ff_out(24)(11 downto 0)<= add_out(24)(11 downto 0);

  add25: adder
   generic map (width_din0=>10, width_din1=>9, width_dout=>11, sign=>1, shift_din0=>0)
   port map(din0=>ff_out(23)(10 downto 1), din1=>ff_out(19)(8 downto 0), dout=>add_out(25)(11 downto 1));
  add_out(25)(0 downto 0)<= ff_out(23)(0 downto 0);
  ff_out(25)(11 downto 0)<= add_out(25)(11 downto 0);

   --level of logic= 3
  add26: adder
   generic map (width_din0=>10, width_din1=>12, width_dout=>12, sign=>3, shift_din0=>0)
   port map(din0=>ff_out(24)(11 downto 2), din1=>ff_out(25)(11 downto 0), dout=>add_out(26)(13 downto 2));
  add_out(26)(1 downto 0)<= ff_out(24)(1 downto 0);
  ff_out(26)(13 downto 0)<= add_out(26)(13 downto 0);

  dout<= ff_out(26);

end al_arch;
```

The final circuit description

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;

entity conv is
   port(clk: in std_logic;
   din0: in std_logic_vector(3 downto 0); -- input data (for different lines)
   dout: out std_logic_vector (13 downto 0) );  --output
end conv;

architecture conv_arch of conv is
  constant width_din: integer:= 4; -- width of din
  constant width_dout: integer:= 14; -- width of dout

  [...] components declaration
function my_conv_unsigned(din: std_logic_vector) return unsigned is – convert std_logic to unsigned
                variable dout: unsigned(din'range);
    begin
   for i in din'range loop
                dout(i):= din(i);
   end loop;
      return dout;
  end;

  type dinu_arr is array(0 downto 0) of unsigned(width_din-1 downto 0); -- input converted to unsigned
  signal dinu: dinu_arr;
  type data_arr is array(14 downto 0) of unsigned(width_dout-1 downto 0);-- data for adders and DAs
  signal data: data_arr;
  signal douta, doutff: unsigned(width_dout-1 downto 0);
  signal zero, one: std_logic;
  signal ce: std_logic;

begin
  zero<= '0';  one<= '1';
  dout<= conv_std_logic_vector(doutff, width_dout);
  ce<= '1';

-- line 0
  dinu(0)(3 downto 0)<= my_conv_unsigned(din0);
```

```
        ffd0: ffg generic map (width_din)
         port map (clk, ce, dinu(0), data(0)(width_din-1 downto 0));
        ff1: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(0)(3 downto 0), dout=> data(1)(3 downto 0));
        ff2: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(1)(3 downto 0), dout=> data(2)(3 downto 0));
        ff3: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(2)(3 downto 0), dout=> data(3)(3 downto 0));
        ff4: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(3)(3 downto 0), dout=> data(4)(3 downto 0));
        ff5: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(4)(3 downto 0), dout=> data(5)(3 downto 0));
        ff6: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(5)(3 downto 0), dout=> data(6)(3 downto 0));
        ff7: ffg generic map (width=>4)
         port map(clk=>clk, ce=>ce, din=>data(6)(3 downto 0), dout=> data(7)(3 downto 0));

       -- Adders and DAs
        a8: ass2
         port map(clk, ce, data(0)(3 downto 0), data(2)(3 downto 0), data(8)(9 downto 0) );

        a9: ac3
         port map(clk, ce, data(4)(3 downto 0), data(5)(3 downto 0), data(9)(6 downto 0) );

        d10: da4g generic map( -- output shift= 0
         coeff0=> 183, coeff1=> 366, coeff2=> 12, coeff3=> 732, width_dout=> 11, insert_ff=> 0)
         port map (clk=>clk, ce=>ce,
         din0=> data(1)(0), din1=> data(1)(1), din2=> data(9)(0), din3=> data(1)(2), dout=> data(10)(10 downto 0));
        d11: da1g generic map( -- output shift= 3
         coeff0=> 3, width_dout=> 2, insert_ff=> 0)
         port map (clk=>clk, ce=>ce,
         din0=> data(9)(1), dout=> data(11)(1 downto 0));
        d12: da4g generic map( -- output shift= 3
         coeff0=> 183, coeff1=> 6, coeff2=> 12, coeff3=> 24, width_dout=> 8, insert_ff=> 0)
         port map (clk=>clk, ce=>ce,
         din0=> data(1)(3), din1=> data(9)(2), din2=> data(9)(3), din3=> data(9)(4), dout=> data(12)(7 downto 0));
        d13: da1g generic map( -- output shift= 7
         coeff0=> 3, width_dout=> 2, insert_ff=> 0)
         port map (clk=>clk, ce=>ce,
         din0=> data(9)(5), dout=> data(13)(1 downto 0));
        d14: da1g generic map( -- output shift= 8
         coeff0=> -3, width_dout=> 3, insert_ff=> 0)
         port map (clk=>clk, ce=>ce,
         din0=> data(9)(6), dout=> data(14)(2 downto 0));

        a: al
         port map(clk, ce, data(8)(9 downto 0), data(8)(9 downto 0), data(0)(3 downto 0), data(2)(3 downto 0),
         data(3)(3 downto 0), data(3)(3 downto 0), data(6)(3 downto 0), data(6)(3 downto 0),
         data(7)(3 downto 0), data(10)(10 downto 0), data(11)(1 downto 0), data(12)(7 downto 0),
         data(13)(1 downto 0), data(14)(2 downto 0), douta(13 downto 0) );
        ffout: ffg generic map (14)
         port map (clk, ce, douta(13 downto 0), doutff(13 downto 0));

end conv_arch;
```

References

[Aar89] Aarts, E.H., Korst, J. *Simulated Annealing and Boltzman Machines*, Wiley, Chichester, UK, 1989

[Alt99] Altera Co. *Apex 20K Programmable Logic Device Family*, Data Sheet, ver 2.05, Nov. 1999.

[Ana99] Analog Devices *TigerSHARC DSP Microcomputer, Preliminary Technical Data ADSP-TS001*, Analog Devices, http://www.analog.com, Dec. 1999.

[And95] Anderson D., Shanley T. *Pentium Processor System Architecture.* Addision-Wesley 1995

[Ayt95] Aytekin T., Korkmaz E.E. Guvernir H.A. *An Application of Genetic Programming to the 4-Op Problem using Map-Trees*, pp.28-40 in Xin Yao *Progress in Evolutionary Computation, Selected Papers on AI'93 nad AI'94 Workshops on Evolutionary Computation*, Springer, Berlin, 1995.

[Boe98] Boemo E.I., Lopez-Buedo S., Meneses J.M., *Some Experiments About Wave Pipelining on FPGA's,* IEEE Trans. on VLSI Systems, vol. 6, no. 2, June 1998.

[Bos99] Bosi B., Bois G., Savaria Y., *Reconfigurable Pipelined 2-D Convolers for Fast Digital Signal Processing*, IEEE Trans. on VLSI Systems, vol. 7, no. 3, pp. 299-308, Sep. 1999.

[Bra97] Bramer B., Chauhan D., Ibrahim M.K., Aggoun A. *Virtual Radix Array Processors (V-RaAP),* 7-th International Workshop, FPL'97, London, UK, September 1-3, 1997, pp. 354-363 in [Luk97].

[Bre97] Brey B. B. *The Intel Microprocessors.* Prentice-Hall 1997

[Bur77] Burrus C.S.: *Digital filter structure described by arithmetic*, IEEE transaction on Circuits and systems, pp. 674-680, 1977

[Cas96] Castleman, K. R.: *Digital Image Processing.* Prentice Hall 1996

[Cha93] Chatterjee A., Roy R.K., d'Abreu M.A., *Greedy hardware optimisation for linear digital circuits using number splitting and refactorization* IEEE Trans. VLSI Syst., vol. 1, pp. 423-431, Dec 1993.

[Cha94] Chapman K. *Fast Integer Multiplier fit in FPGA's,* EDN 1993 Design Idea Winner, END May 12th 1994.

[Cha96] Chapman K. *Constant Coefficient Multipliers for the XC4000E*. Xilinx Application Note, XAPP 054 December 1996.

[Cha00] Chang T.S., Chu Y.H., Jen C.W., *Low-Power Filter Realisation with Differential Coefficients and Inputs*, IEEE Trans. on Circuits and Systems II, vol. 47, no. 2, Feb 2000.

[Cho93] Chou C.J., Mohanakrishnan S., Evans J.B., *FPGA Implementation of digital filters* Proc. Int. Conf. Signal Proc. Appl. & Tech. 1993.

[Cor94] Cormen T.H., Leiserson C.E., Rivest R.L. *Intoduction to Algorithms* Massachusetts Institute of Technology, 1994

[DeH98] DeHon A. *Comparing Computing Machines*, SPIE Conference on Configurable Computing, Technology and Applications, Boston, Massachusetts, Nov 1998.

[Dic98] Dick R.P., Jha N.K., *MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pp. 920-935, Oct. 1998.

[Do98] 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.

[Eva94] Evans J.B., *Effiecient FIR Filter Architectures Suitable for FPGA Implementation*, IEEE Trans. Circuit & Systems, July 1994.

[Ele98] Eles P., Kuchcinski K., Peng Z. *System Synthesis with VHDL* Kluwer Academic Publ. 1998.

[Eve98] Evers M., Patel S. J., Chappell R. S., and Patt Y.N, *Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work*, Proceedings of the 25th International Symposium on Computer Architecture, Barcelona, June 1998

[Fly66] Flynn M. J. Very *High-Speed Computing Systems. Proc.* IEEE, Vol 54, New York 1966, Nr 12.

[Gar65] Garner H. *Number Systems and Arithmetic*, Advances in Computing, vol. 6, pp. 131-194, 1965

[Gol89] Goldberg D.E. *Genetic Algorithms in Search Optimisation & Machine Learning*, Addison-Wesley, Massachusetts, 1989.

[Gon87] Gonzalez, R. , Wintz P., *Digital Image Processing*, Addision-Wesley 1987.

[Gre97] Greenfield D., Crome C., Won M.S., Amos D., *Anhancing Fixed Point DSP Processor Performance by adding CPLD's as Coprocessing Elements*, 7-th International Workshop, FPL'97, London, UK, September 1-3, 1997, pp. 354-363 in [Luk97].

[Gup97] Gupta R.K., Zorian Y., *Introducing Core-Based System Design*, IEEE Design & Test of Computers, pp. 15-25, Oct.-Dec. 1997.

[Har90] Harley R., Corbet P., *Digit-serial processing techniques* IEEE Trans. On Circuits and Systems, Vol. 37, no. 6, pp. 707-719, June 1990.

[Har94] Harris Inc., *Digital Signal Processing - Databook*, Harris Semicondactor Inc., 1994

[Har96] 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.

[Harte96] Hartenstein R. W. Glesner M. *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, 6th International Workshop on Field-Programmable Logic and Applications, FPL'96, Darmstadt, Germany, September 1996.

[Haw96] 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

[Hes96] Hesener A., *Implementing Reconfigurable Datapath in FPGAs for Adaptive Filter Design*,Proc. 6th International Workshop on Field-Programmable Logic and Applications, FPL'96, Darmstadt, Germany, Sep. 1996, pp. 220-229 in [Harte96].

[Hut97] Hutchings B.L. *Exploiting Reconfigurability Through Domain-Specific Systems*, 7th International Workshop, FPL'97, pp. 193-202 in [Luk97].

[Hwa93] Hwang K. *Advanced Computer Architecture. Parallelism, Scalability, Programmability* McGraw-Hill Singapore, 1993

[Int97a] Intel Co.: *Intel Architecture Software Developer's. Manual Volume 3: System Programming Guide.* (Order Number 243192). Intel Corporation 1997

[Int97b] Intel Co.: *Intel Architecture Software Developer's. Manual Volume 1: Basic Architecture. (Order Number 243190).* Intel Corporation 1997

[Int00] Intel Co. *A Detailed Look Inside the Intel NetBurst Micro-Architecture of the Intel Pentium 4 Processor,* Intel Corporation 2000

[Jai91] Jain R., Yang P.T. Yoshino T., *FIRGEN, a computer-aided design system for high performance FIR filter integrated circuits,* IEEE Trans. Signal Processing, vol. 39, no. 7, pp. 1655-1667, 1991.

[Jam97] Jamro E. *The design of a VHDL synthesis tool of BCH codes,* Master of Philosophy Thesis, University of Huddersfield U.K. September 1997.

[Jam99] Jamro E. *Synteza układów z parametrem w języku VHDL na przykładzie kodeka kodu BCH.* II Krajowa Konferencja Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, Kraków 25-17 X 1999, pp. 39-43.

[Jam00] Jamro E. *Automatyczna synteza układów z parametrem w języku VHDL na przykładzie kodeka kodu BCH,* Krajowe Sympozjum Telekomunikacji' 2000, 6-8 wrzesień 2000, ATR Bydgoszcz. Vol D. p. 59-64.

[Jam01a] Jamro E., Wiatr K., *FPGA Implementation of Addition as a part of the convolution,* Proc. of the IEEE Int. Conf. Digital System Design, Warszawa, Poland, IEEE Computer Society Press, 4-6 Sep 2001.

[Jam01b] 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, IEEE Computer Society Press, 4-6 Sep. 2001.

[Jam01c] Jamro E., Wiatr K., *Convolution Operation Implemented in FPGA Structures for Real Time Image Processing,* Proc. of the IEEE Int. Symposium on Image Processing and Analysis, Pula, Croatia, 19-21 June 2001.

[Jam01d] Jamro E., Wiatr K., *Implementation of real time image convolution in FPGA structures,* Image Processing & Communications, An International Journal, Bydgoszcz 2001.

[Jan97] Jang J.S.R., Sun C.T., Mizutani E., *Neuro-Fuzzy and Soft Computing,* Prentice-Hall, London, UK, 1997.

[Kea98] Keating M., Bricaud P. *Reuse Methodology Manual for System-On-A-Chip Designs,* Kluwer Academic Publishers, Boston, 1996.

[Kim98] Kim T., Jao W., Tjiang S., *Circuit Optimisation Using Carry-Save-Adder Cells,* IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pp. 974-984, Oct. 1998.

[Kim00] Kim T., Um J., *A Practical Approach to the Synthesis of Arithmetic Circuits Using Carry-Save-Adders,* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 5, pp. 615-623. May 2000.

[Kir83] Kirkpatrick, S. Gelatt, C.D., Vecchi, M.P. *Optimisation by simulated Annealing,* Science, 220 (4598): 671-680, May 1983.

[Kla00] Klaiber A.: *The Technology behind Crusoe Processors.* Transmeta Co. Jun 2000 http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf

[Kos97] Kostarnov I., Morley S., Osmany J., Solomon C., *A Reconfigurable Approach to Low Cost Media Processing* 7t^{h} International Workshop, FPL'97, pp. 79-90 in [Luk97].

[Koz92] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press, 1992.

[Kur00] Kurdahi F.J.K., Bagherzadeh N., Athanas P., Munoz J.L., *Guest Editors' Introduction: Configurable Computing,* IEEE Design & Test of Computers, Jan-Mar 2000, pp. 17-19.

[Las92] Laskowski J., Samueli H., *A 150 MHz 43-tap half-band FIR digital filter in 1.2 μm CMOS generated by silicon compiler,* in Proc. Custom Integrated Circuits Conference, May 1992, pp. 11.4.1-11.4-4.

[Lim83] Lim Y.C., Parker S.R., *FIR filter design over discrete power-of-two coefficient space*, IEEE Trans. Acoust., Speech Signal Processing, ASSP-31: 583-591, Nov. 1983.

[Lis97] Lisa F., Cuadrado F., Rexachs D., Carravina J., *A Reconfigurable Coprocessor for a PCI-based Real Time Computer Vision System*, 7th International Workshop, FPL'97, London UK, Sep. 1997, pp. 392-399 in [Luk97].

[Lu92] Lu W.-S., *Two-Dimensional Digital Filters*, Marcel Dekker, New York, 1992.

[Luk96] Luk W., Guo S., Shirazi N., Zhuang N. *A Framework for Developing Parameterised Libraries* in [Hartle96] pp. 24-33.

[Luk97] Luk W., Cheung P.Y.K., Glesner M., *Field-Programmable Logic and Applications*, Proceedings 7th International Workshop, FPL'97, London, UK, Sep. 1-3, 1997. Springer, Berlin,

[Luo98] 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.

[Mad95] Madisetti V. K. *VLSI Digital Signal Processors.* Butterworth Heinemann 1995

[Matrox] MATROX: *Matrox MIL-32 Library.* Matrox Electronic Systems Ltd, http://www.matrox.com /imgweb/

[Mau84] Maudlin, M.L. Maintaining Diversity in Genetic Search, AAAI Proc. National Conference on Artificial Inteligence, 1984, pp. 247-250.

[McM95] McMahon G. Hadinoto D. *Comparison of Heuristic Search Algorithms for Single Machine Scheduling Problems*, pp. 293-304 in Xin Yao *Progress in Evolutionary Computation, Selected Papers on AI'93 nad AI'94 Workshops on Evolutionary Computation*, Springer, Berlin, 1995.

[Meh97] Mehrotra K., Mohan C. K., Ranka S. *Elements of artificial neural networks*, MIT Press, Cambridge, London, 1997.

[Mic92] Michalewicz Z. *Genetic Algorithm + Data Structure = Evolutions Programs*, Spinger-Verlag, Berlin, 1992.

[Min92] Mintzer, L. *FIR filters with the Xilinx FPGA*, FPGA '92 ACM/SIGDA Workshop on FPGAs pp. 129-134

[Moh95] Mohanakrishnan S., Evans J.B. *Automatic Implementation of FIR filters on Field Programmable gate Arrays*, IEEE, Signal Processing Letters, March 1995.

[Nag98] Nag S.K., Rutenbar R.A., *Performance-Driven Simultaneous Placement and Routing for FPGA's*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 6, pp. 499-518, June 1998.

[Omo94] Omondi A.R *Computer Arithmetic Systems. Algorithms Architecture and Implementations*, Prentice Hall, UK, 1994

[Par97] Parker D.A. Parhi K.K. *Low-Area/Power Parallel FIR Digital Filter Implementations*, Journal of VLSI Signal Processings 17, 75-92, Kluwer 1997.

[Pas99] Pasko R., Schaumont P., Derudder V., Vernalde S., Durackova D., *A New Algorithm for Elimination of Common Subexpressions*, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 1, Jan 1999.

[Pas01] Pasham V., Miller A., Chapman K., *Transposed Form FIR Filters*, Xilinx Application Note, XAPP219, Jan 2001.

[Pei99] Peiro M.M., Valls J., Sansoloni T., Pascuual A.P., Boemo E.I., *A comparison between Lattice, Cascade and Direct-form FIR filter structures by using a FPGA bit-serial Distributed Arithmetic implementation,* Proc. IEEE ICECS Vol. I pp. 241-144, Paphos, Cyprus, Sep. 1999.

[Pet95] 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.

[Pir98] Pirsch P., *Architectures for Digital Signal Processing*, Chichester UK, Wiley 1998.

[Ple90] Plessey: *Digital Signal Processing - IC Handbook.* GEC Plessey Semiconductor 1990.

[Por97] Porat B., *A Course in Digital Signal Processing,* John Wiley & Sons, 1997.

[Pot96] Potkonjak M., Srivastava M.S., Chandrakasan A.P., *Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination,* IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol.15, no. 2, Feb 1996.

[Rox00] Roxby P.J., Prada E.C., Charlwood S., *Core-based design methodology for reconfigurable computing applications,* IEE Proc. Comput. Digit. Tech. Vol. 147, no. 3, pp. 142-146, May 2000.

[Sam89] 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.

[San99] Sanchez E., Sipper M., Haenni J., Beuchat J., Perez-Uribe A. *Static and Dynamic Configurable Systems,* IEEE Transactions on Computers, col. 48, no. 6, pp. 556-563, June 1999.

[Sei84] Seitz C.L. *Concurrent VLSI architectures,* IEEE Trans. on Computers, Vol. C-33, No. 12, pp. 1247-1265, 1984.

[Ser01] Sergyienko A., Vasylienko V., Maslennikow O., *FIR Filter Soft Core Generator*, Repregrogramowalne Układy Cyfrowe Szczecin 7-8 May 2001.

[Slo00] Slomka F., Dorfel M., Munzenberger R., Hofmann R., *Hardware/Software Codesign and Rapid Protopyping of Embedded Systems,* IEEE Design & Test of Computers, pp. 28-38, Apr.-June 2000.

[Smi96] Smith D., Bhatia D., *RACE: Reconfigurable and Adaptive Computing Enviroment,* Proc. 6th International Workshop on Field Programmable Logic and Applications, FPL'96, Darmstadt, Germany, Sep. 1996; pp. 87-95 in [Harte96].

[Smt81] Smith, J. E. *A study of branch prediction strategies,* in Proceedings of the 8th Annual International Symposium on Computer Architecture, pp. 135-148, 1981.

[Sta97] Staunstrup J., Wolf W. *Hardware/Software Co-design: Principles and Practise* Kluwer, Boston, 1997.

[Tad92] Tadeusiewicz R. *Systemy wizyjne robotów przemysłowych*, Warszawa WNT 1992.

[Tad93] Tadeusiewicz R., *Sieci Neuronowe,* Warszawa, Akademicka Oficyna Wydaw., 1993.

[Tex95] Texas Instruments, *TMS320C80 (MVP), User's Guide.* Texas Instruments Inc., 1995, http://www.ti.com/

[Tex97] Texas Instruments, *Implementation of an Image Processing Library for TMS320C8x (MVP).* Texas Instruments Inc. 1997.

[Tho90] SGS Thomson: *Image Processing - Databook.* SGS Thomson Microelectronics 1990

[Tul95] Tullsen D., Eggers S., Levy H. *Simultaneous Multithreading: Maximizing On-Chip Parallelism,* Proceedings of the 22rd Annual International Symposium on Computer Architecture, June 1995, pages 392-403.

[Vai93] Vaidyanathan P.P. *Multirate Systems and Filters Banks,* Prentice-Hall, New Jersey, 1993.

[Val98] Valls J., Peiro M.M., Sansaloni T., Boemo E., *A study about FPGA-based digital filters*, Proc. IEEE SIPS (IEEE Workshop on VLSI Signal Processing Design and Implementation) pp. 191-201, Boston, Oct. 1998.

[Wal64] Wallace C.S. *A suggestion for a fast multiplier*, IEEE Trans. On Electron. Comput., Vol. EC-13, pp. 14-17, 1964.

[Was78] Waser S., *High-speed monolithic multipliers for real-time digital signal processing*, IEEE Computer Magazine, Vol. 11, No. 10, pp.19-29, 1978.

[Wia98] Wiatr K., *Architektura potokowa specjalizowanych procesorów sprzętowych do wstępnego przetwarzania obrazów w systemach wizyjnych czasu rzeczywistego*, Wydawnictwo AGH, Kraków 1998.

[Wia99a] Wiatr K., Jamro E., *Implementacja algorytmu konwolucji 2D dla potrzeb przetwarzania obrazów w czasie rzeczywistym.* Kwartalnik Elektrotechnika i Elektronika, Tom 18, Zeszyt 4, 1999, pp. 157-169.

[Wia99b] Wiatr K., Jamro E., *Obliczanie algorytmu konwolucji dla potrzeb przetwarzania obrazów w czasie rzeczywistym*, II Krajowa Konferencja Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, Kraków 25-17 X 1999, pp. 459-464.

[Wia00a] Wiatr K., Jamro E., *Implementacja algorytmu konwolucji 2D w procesorach ogólnego przeznaczenia i w układach specjalizowanych VLSI.* Kwartalnik Elektronika i Telekomunikacja PAN, Tom 46, Zeszyt 4, 2000, s 553-587.

[Wia00b] 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.

[Wia00c] 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.

[Wia00d] Wiatr K., Jamro E., *Implementacja arytmetyki rozproszonej w układach programowalnych FPGA na przykładzie operacji konwolucji 2D*, Kwartalnik Elektrotechnika i Elektronika, Tom 19, Zeszyt 2, Kraków 2000, s. 98-104.

[Wia01a] Wiatr K., Jamro E., *Implementation of Multipliers in FPGA Structures*, Proc. of the IEEE International Symposium on Quality Electronic Design, San Jose, California, 26-28 March 2001, pp. 415-420, IEEE Computer Society Press.

[Wia01b] Wiatr K. Jamro E. *Układy mnożące przez stały współczynnik implementowane w układach programowalnych FPGA*, Kwartalnik Elektronika i Telekomunikacja PAN, Warszawa 2001, Tom 47, Zeszyt 2, pp. 231-251.

[Wil00] Wilton S.J.E., *Heterogeneous Technology Mapping for Area Reduction in FPGA's with Embedded Memory Areays*, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 1, Jan 2000.

[Wir97] 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

[Wir98] Wirthlin M. J., Hutchings B.L., *Improving Functional Density Using Run-Time Circuit Reconfiguration*, IEEE Trans. on VLSI Systems, vol. 6, no. 2, June 1998.

[Woj98] Wojko M., ElGindy H., *Self Configuring Binary Multipliers for LUT addressable FPGAs*, 5th Australian Conference on Parallel and Real-Time Systems. University of Adelaide, Australia, 28-29th September 1998, pp. 201-212

[Woj99] 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.

[Won00] Wong H.S.P., Frank D.J., Solomon P.M., Wann C.H.J. Welser J.J., *Nanoscale CMOS,* Proceedings of the IEEE, vol. 87, no. 4, pp. 537-570, Apr. 1999.

[Xil93] Xilinx Inc. *The programmable Logic Data Book* San Jose, California, 1993

[Xil96] Xilinx Inc. *Using the Dedicated Carry Logic in XC4000E*, Xilinx Application Note XAPP 013 July 4, 1996.

[Xil99a] Xilinx Inc. Core Generator, Foundation 2.1, 1999

[Xil99b] Xilinx Inc. *The Programmable Logic* Data Book, San Jose, California, 1999.

[Xil00] Xilinx Inc. Virtex-E 1.8V, Field Programmable Gate Array, Nov 2000, www.xilinx.com.

[Xin98] Xing S., Yu W.W.H., *FPGA Adders: Performance Evaluation and Optimal Design,* IEEE Design & Test of Computers, pp. 24-29, Jan.-Mar. 1998.

[Xu00] Xu B., Albonesi D.H., *Runtime Reconfiguration Techniques for Efficient General-Purpose Computation,* IEEE Design & Test of Computers, Jan-Mar 2000, pp. 42-52.

[Zam94] Zamojski W., Caban D. (Eds), Proceedings of the 5-th School Computer Vision and Graphics - 1994, Wydawnictwo Prac Naukowych Format, Wrocław 1994.

[Zam95] Zamojski W., Mazurkiewicz J., *Proceedings of the 7-th School VLSI and ASIC Design* - 1995, Wydawnictwo Prac Naukowych Format, Wrocław 1995.