Konrad Möller

Run-time Reconfigurable Constant Multiplication on Field Programmable Gate Arrays

kassel university press

Konrad Möller

Run-time Reconfigurable Constant Multiplication on Field Programmable Gate Arrays

kassel
university
press

This work has been accepted by the Faculty of Electrical Engineering / Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Ingenieurwissenschaften (Dr.-Ing.).

Supervisor: Prof. Dr.-Ing. Peter Zipf, University of Kassel
Co-Supervisor: Prof. Dr.-Ing. Uwe Meyer-Baese, Florida A&M University – Florida State University College of Engineering
Defense day: 11th August 2017

Bibliographic information published by Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at http://dnb.dnb.de.

Zugl.: Kassel, Univ., Diss. 2017
ISBN 978-3-7376-0376-8 (print)
ISBN 978-3-7376-0377-5 (e-book)
DOI: http://dx.medra.org/10.19211/KUP9783737603775
URN: http://nbn-resolving.de/urn:nbn:de:0002-403772

Abstract

This thesis addresses the question how run-time reconfigurable constant multipliers (RCMs) can be efficiently implemented on field programmable gate arrays (FPGAs). RCMs calculate the multiplication of an input number by one out of several constants which can be selected dynamically during run-time. The evaluation of RCMs is done by considering three different reconfiguration principles, namely: reconfiguration using reconfigurable look-up tables (LUTs), reconfiguration using multiplexers and Partial Reconfiguration (PR). While solutions for the latter one are already provided by the FPGA vendor's software tools, this thesis contributes two new methods to generate RCMs using the first two reconfiguration principles. First, a LUT-based constant multiplier is extended to be reconfigurable. Second, optimized constant multipliers without reconfiguration are fused using multiplexers. Moreover, a general post-optimization for multiplexer-based RCMs is proposed. Finally, the design space produced in this way is analyzed using synthesis experiments. The contributed methods provide some important trade-off points in the design space of run-time reconfigurable constant multiplication on FPGAs. This is important as constant multiplication is an essential operation in digital signal processing (DSP) applications. Furthermore, FPGAs become more and more important for DSP applications which were traditionally implemented using application specific integrated circuits (ASICs). As FPGAs have an inherent inefficiency caused by their re-programmability, trade-offs and target optimized FPGA implementations are required to narrow the gap to equivalent ASIC implementations.

Acknowledgements

First of all, I want to thank my wife Sarah and my parents Elfriede and Winfried for their unconditioned support in all life challenges.

I especially want to thank Peter Zipf for supervising this thesis and for supporting my work in the Digital Technology Group of the University of Kassel in all its facets. I am especially thankful for his excellent guidance and his confidence in my skills. Next, I want to kindly thank my colleague Martin Kumm for always being interested in my work and many fruitful common projects. Martin and Peter were always available to discuss research questions. These discussions brought forth new ideas and helped to focus on the right aspects of a topic.

A big thanks to all the students that supported parts of this thesis during their project work, student job and master's thesis. In particular, I would like to emphasize the work of Marco Kleinlein on the VHDL code generator and the PAG fusion implementation and the work of Björn Barschtipan on reconfigurable multiplier block extensions.

Next, I want to thank Mario Garrido from the University of Linköping, Sweden, and Uwe Meyer-Baese from the Florida State University, USA for their confirmation of my work during their stay at the University of Kassel. Special thanks to Mario for our collaboration on multiplexer-based reconfigurable constant multiplication and to Uwe for reviewing this thesis as second assessor.

I also want to thank the mathematicians Diana Fanghänel, Evelyn Lerche and Dörthe Janssen for being interested in my research and for their help with mathematical questions.

I appreciate the financial support and the recognition of the research questions of this thesis by the GSI Helmholtzzentrum für Schwerionenforschung Darmstadt (GSI) and the Deutsche Forschungsgemeinschaft (DFG).

Finally, I want to thank the whole current and former team of the Digital Technology Group for administrative help, a good cooperation and fruitful discussions.

Contents

List of Figures

List of Tables

Abbreviations

ALM	adaptive logic module
ALUT	adaptive look-up table
ASIC	application specific integrated circuit
BLE	basic logic element
CCLK	configuration clock
CDI	configuration data in
CDO	configuration data out
CE	clock enable
CFGLUT	configurable look-up table
CLB	configurable logic block
CMM	constant matrix multiplication
CSD	canonical signed digit
DA	Distributed Arithmetic
DAG	directed acyclic graph
DSP	digital signal processing
FA	full adder
FIFO	first-in-first-out buffer
FIR	finit impuls response
FPGA	field programmable gate array
HDL	hardware description language
HLS	high-level synthesis
IC	integrated circuit

ICAP	Internal Configuration Access Port
ILP	Integer Linear Programming
IP	intellectual property
KCM	Ken Chapman multiplier
LAB	logic array block
LUT	look-up table
MCM	multiple constant multiplication
MSB	most significant bit
OSR	optimal shift reassignment
PAG	pipelined adder graph
PR	Partial Reconfiguration
RAM	random access memory
RCA	ripple-carry adder
RCM	run-time reconfigurable constant multiplier
ReMB	reconfigurable multiplier block
RMCM	reconfigurable multiple constant multiplier
ROM	read-only memory
RPAG	reduced pipelined adder graph
SCM	single constant multiplication
SD	signed digit
SRL	shift register look-up table
VHDL	very high speed integrated circuit hardware description language
XPE	Xilinx Power Estimator

Author's Publications

The following publications were published during the time of PhD preparation between October 2012 and June 2017. Most of them are included in this thesis.

[1] M. Kumm, K. Möller and P. Zipf, "Partial LUT Size Analysis in Distributed Arithmetic FIR Filters on FPGAs," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 2054–2057.

[2] M. Kumm, K. Möller and P. Zipf, "Reconfigurable FIR Filter Using Distributed Arithmetic on FPGAs," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 2058–2061.

[3] M. Kumm, D. Fanghänel, K. Möller, P. Zipf and U. Meyer-Baese, "FIR Filter Optimization for Video Processing on FPGAs," in *EURASIP Journal on Advances in Signal Processing (Springer)*, pp.1-18, 2013.

[4] M. Kumm, K. Möller, and P. Zipf, "Dynamically Reconfigurable FIR Filter Architectures with Fast Reconfiguration," *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2013, pp. 1–8.

[5] K. Möller, M. Kumm, B. Barschtipan, and P. Zipf, "Dynamically Reconfigurable Constant Multiplication on FPGAs." in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014, pp. 159–169.

[6] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, "Pipelined Reconfigurable Multiplication with Constants on FPGAs," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2014, pp. 1–6.

[7] K. Möller, M. Kumm, P. Zipf, K. Groß, D.Lens, and H.Klingbeil, "FPGA Based Tunable Digital Filtering for Closed Loop RF Control in Synchrotrons," in *GSI Helmholtzzentrum für Schwerionenforschung, Scientific Report*, 2014, pp. 331–332.

[8] K. Möller, M. Kumm, C.-F. Müller, and P. Zipf, "Model-based Hardware Design for FPGAs using Folding Transformations based on Subcircuits," in *International Workshop on FPGAs for Software Programmers (FSP)*, 2015, pp. 7–12.

[9] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, "Reconfigurable Constant Multiplication for FPGAs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 36, no. 6, pp. 927-937, 2017.

[10] P. Sittel, M. Kumm, K. Möller, M. Hardieck and P. Zipf, "High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2017.

[11] P. Sittel, M. Kumm, K. Möller, B. Pasca, M. Jervis and P. Zipf, "Automatic Folding for Model-Based Hardware Design using Isomorphic Subgraphs", currently under review to be published at the *International Conference on Field Programmable Technology (FPT)*, 2017.

[12] K. Möller, M. Kumm, M. Garrido, and P. Zipf, "Optimal Shift Reassignment in Reconfigurable Multiplication Circuits," accepted for publication in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017

1 Introduction

In this thesis optimization methods to implement run-time reconfigurable constant multipliers (RCMs) on field programmable gate arrays (FPGAs) are proposed. The performance, hardware effort, reconfiguration time and power consumption of resulting circuits are evaluated. The resulting solutions add some important trade-off points to the design space of RCM on FPGAs and make new applications possible.

1.1 Motivation

Multiplication with constants is one of the most frequent operations in digital signal processing (DSP). At the same time, FPGAs have a growing market in DSP applications which were formerly dominated by application specific integrated circuit (ASIC) implementations. Reasons for this trend are the flexibility provided by the re-programmability of FPGAs and increasing ASIC manufacturing costs. The costs of the re-programmability of FPGAs are that FPGA designs are typically larger, slower and consume more power than an equivalent ASIC realization [1]. Therefore, optimized implementations of DSP algorithms for FPGAs are getting more and more important. This is one of the reasons why embedded multipliers are present in the fabric of FPGAs. Nevertheless, the drawback of those fixed coarse-grained blocks is their inflexibility in word size and their limited quantity. Limited quantity is particularly critical in industrial applications when low-cost FPGAs with only few embedded multipliers have to be chosen and other parts of a design are competing for DSP resources. Thus, alternative logic-based methods for constant multiplication are required which are independent of this embedded special purpose hardware but are, on the other hand, efficient enough to narrow the gap to an ASIC realization. Therefore, optimizing the implementation of constant multiplication as shift-add-based circuit is well studied [2–17]. However, switching between a given limited set of constant multiplications during run-time instead of using larger generic multipliers is important, too. Reconfigurable constant multipliers are used to realize hardware efficient run-time adaptable filters [18–21], e.g., for adaptive control and video coding applications. Specifically in [20] an application with tight reconfiguration time and resource constraints is presented, which motivates the necessity of highly optimized RCMs on FPGAs. There, an FPGA is used as co-processor in the control loop of a particle accelerator.

In addition to that, RCMs can be directly integrated into optimized time-multiplexed realizations of linear DSP transforms like DCT and FFT implementations [22, 23] and can be used in the context of time-multiplexed resource sharing of linear systems in general. Further applications are multi-stage filters for decimation or interpolation, like polyphase finit impuls response (FIR) filters [24]. Therefore, the implementation of run-time reconfigurable constant multipliers using multiplexers is a well studied research field, too [19, 21, 24–28]. However, most of the previous methods to generate RCMs were optimized to be used on ASICs. On FPGAs long routing delays have to be avoided by inserting registers in the data path. That is why previous RCM solutions perform poor when they are directly applied to FPGAs [29]. As FPGA implementations of DSP applications are getting more and more important, the algorithms presented in this work particularly focus on implementations of RCMs on FPGAs.

Three ways of run-time reconfiguration are distinguished in this work: Logic reconfiguration using reconfigurable look-up tables (LUTs), routing reconfiguration using multiplexers and Partial Reconfiguration (PR). The latter reconfiguration method is provided by the FPGA vendors. However, it comes along with long reconfiguration times and a thereby arising large reconfiguration power consumption. This means, it is not particularly suitable for the above mentioned applications. Therefore, the presented RCM generation methods focus on LUT-based reconfiguration and reconfiguration using multiplexers.

1.2 Organization and Contributions of the Thesis

This section gives an overview on the organization of the thesis. Moreover, the main contributions of each chapter are provided.

The background for this thesis is provided in Chapter 2. It gives an introduction to FPGAs as target technology and introduces the different run-time reconfiguration concepts in detail. In addition to that, the background on multiplier-less constant multiplication is provided with a focus on FPGA-specific aspects at the end of the chapter.

Logic reconfiguration using reconfigurable LUTs is covered in Chapter 3. It is shown how reconfigurable constant multiplication based on LUTs can provide hardware optimized RCM solutions, which outperform generic IP core multipliers and non-reconfigurable constant multipliers reconfigured using Partial Reconfiguration. The contributions of this chapter are a new method to generate LUT-based run-time reconfigurable constant multipliers based on Ken Chapman multipliers (KCMs) [30] and two new architectures for LUT-based run-time reconfigurable FIR filters. In addition to a KCM-based FIR filter implementation, an FIR filter architecture based on Distributed Arithmetic (DA) [31, 32] and its automatic generation is presented.

An algorithm to generate reconfigurable constant multipliers using multiplexers is presented in Chapter 4. Previous methods on reconfigurable multiplier-less constant multiplication did not consider costs for registers in the combinatorial path of the circuit, which can be used to split up the combinatorial parts during optimization. However, this is important to realize fast FPGA implementations. Moreover, only heuristic solutions for reconfigurable single constant multiplication (SCM) and reconfigurable multiple constant multiplication (MCM) were presented so far. The contribution of this chapter is a new algorithm to generate multiplexer-based run-time reconfigurable constant multipliers for FPGAs including SCM, MCM and their generalization to constant matrix multiplication (CMM). The algorithm considers the insertion of registers during optimization and can be either used to generate optimal solutions or as heuristic, if run-time limitations are present. The C++ implementation of the proposed algorithm is published as open-source [33]. Moreover, the solutions of the proposed algorithm can be used within a high-level synthesis (HLS) flow called Origami HLS [34]. This enables the integration of low-level optimized reconfigurable components in high-level system development.

A post-optimization for multiplexer-based RCMs is described in Chapter 5. It is applicable to all existing RCM solutions for FPGAs and ASICs. The main contribution is a new algorithm based on Integer Linear Programming (ILP) to find an optimal distribution of shifts in given multiplexer-based RCMs.

The different methods of run-time reconfiguration presented in this thesis add some important points to the design space of run-time reconfigurable constant multiplication. The contribution of Chapter 6 is to contrast the different methods in terms of hardware effort, performance, power and reconfiguration time by evaluating this design space using reconfigurable FIR filter implementations.

Finally, Chapter 7 summarizes this thesis and Chapter 8 provides an outlook to future work.

2 Background

This chapter provides the background for the following chapters, starting with an overview on field programmable gate arrays (FPGAs), which are the target technology for most of the proposed optimizations. Moreover, the different ways to perform a run-time reconfiguration on FPGAs are presented. This is followed by an introduction to multiplier-less constant multiplication, which also covers FPGA specific aspects of this operation.

2.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are array-based integrated circuits, which can be programmed and re-programmed on-site. They are regularly structured as two-dimensional array originally consisting of basic logic elements (BLEs) and an interconnecting network. Programmability is achieved by small programmable memories, which change the logic function of BLEs as well as their interconnection. The price for this programmability is that FPGA designs are larger, slower and consume more dynamic power than an equivalent application specific integrated circuit (ASIC) realization [1]. However, FPGAs are widely used with a growing market in the aerospace and consumer electronics and automotive industry. Reasons for the growing market in these industry segments could be a growing demand for high performance digital signal processing (DSP) applications and the flexibility of programming in field, which speeds up system development and allows fast response on changing requirements. Moreover, good performance and a low price are a great advantage to keep the financial risk for small and medium volume developments low. This is not the case for ASICs having large non-recurring engineering costs.

A simplified overview of the basic FPGA layout is given in Figure 2.1. It shows only a small part of an FPGA which typically consists of hundreds of thousands BLEs. Each BLE consists at least of a function generator realized as look-up table (LUT) and a memory element. The input/output ports (IO) as well as the BLEs are connected to the routing network with connection blocks. Some simplified example connections are shown. Moreover, there are programmable routing switches to switch between different routing segments and routing domains. An FPGA can thusly be programmed to implement any logical circuit, only limited by its required hardware resources and interconnection complexity. Since the first FPGAs were commercially introduced in

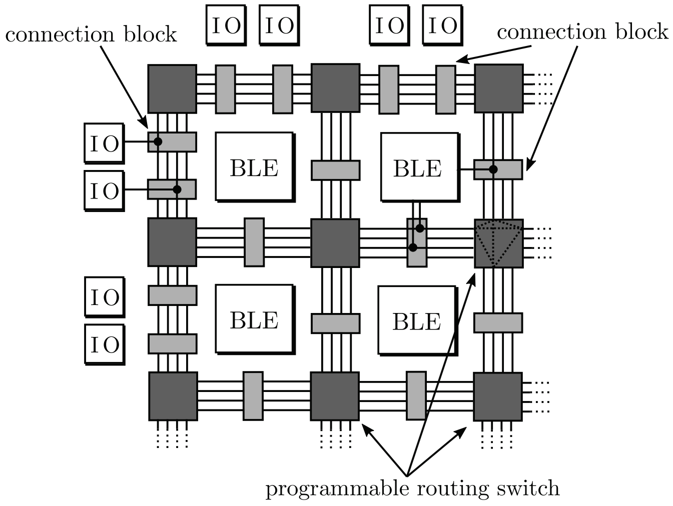


Figure 2.1: Simplified overview of an FPGA layout.

the mid 1980's by Xilinx, FPGAs are subject to ongoing changes and innovations. Important examples are the inclusion of embedded memories, embedded multipliers and many more specialized hardware units into the FPGA layout to support certain applications.

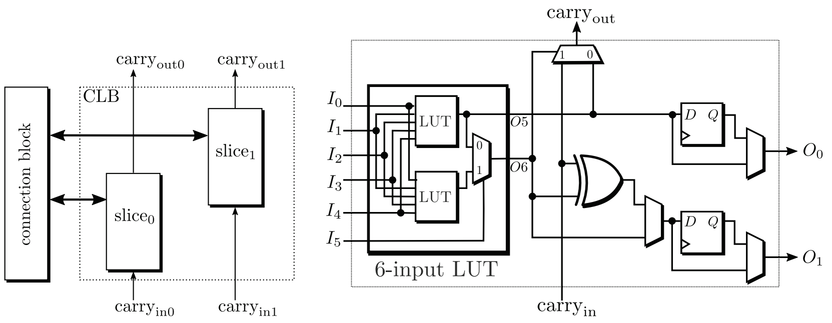
In the following, details on recent FPGAs of the market leaders Xilinx and Intel are presented. The focus is on the features which are important in the following chapters. More details on technology and features of current FPGAs can be found in text books, e. g., [35] and in the FPGA vendor's handbooks and white-papers, e. g., [36], [37]. The selected device families (Xilinx Virtex 6 and Intel Stratix V) are used in the following chapters for the experimental evaluations. Aspects of other device families are given when important.

2.1.1 Architectural Features

In both device families the BLEs are clustered into larger units called configurable logic block (CLB) in Xilinx FPGAs and logic array block (LAB) in Intel FPGAs. Within these units routing is mainly fixed and only a portion of the number of BLEs inputs is connected to the global routing network. This saves routing resources and reduces the complexity of finding a feasible routing solution.

Configurable Logic Block

Each Xilinx CLB consists of two sub-components called slices. As shown in Figure 2.2(a) these slices are connected to the connection block, but are not connected to



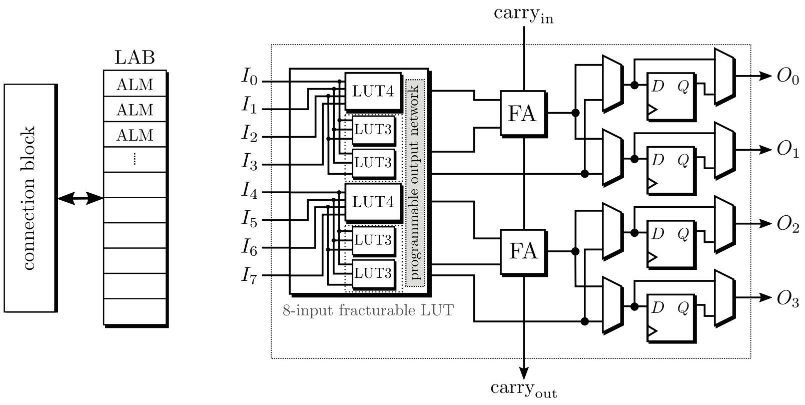
(a) Configurable logic block (b) Simplified block diagram of a Virtex 6 BLE

Figure 2.2: Xilinx Virtex 6 BLE and its clustering to a slice and a CLB.

each other. Moreover, each slice has a special input and output. These are used to propagate the internal carry in and carry out signals to a slice in the next CLB without using the global routing network. A Virtex 6 slice consists of four BLEs. A simplified overview of one such BLE is shown in Figure 2.2(b). It consists of a 6-input LUT, which can be used as two 5-input LUTs, part of a carry chain to build ripple-carry adders (RCAs) and two optional output registers. Hence, any 6-input-1-output function or any 5-input-2-output function with shared inputs can be mapped to a BLE. For example, a full adder (FA) which is required for an RCA can be implemented using the LUT together with the present XOR gate and carry chain multiplexer. There are several multiplexers for signal routing, which are either used within the application's data flow (drawn with a select input) or configured when the FPGA is programmed (drawn without select input). In addition to that, some special interconnections and multiplexers for advanced features are present, which are not shown here. Important are 3 multiplexers, which can be used to link the four BLE's outputs in a slice. An example of their usage is given in Section 2.2.3. A fully detailed CLB and slice block diagram can be found in the vendor's handbook on Virtex 6 CLBs [36].

Logic Array Block

Each Intel LAB consists of ten adaptive logic modules (ALMs) and is connected to the routing network via a connection block (see Figure 2.3(a)). An ALM can be seen as BLE of modern Intel FPGAs. A simplified overview of an ALM is shown in Figure 2.3(b). The shown multiplexers are configured when the FPGA is programmed. The 8-input fracturable LUT can be used to implement a full 6-input LUT or different combinations of LUTs, like, e. g., a 5-input and a 3-input LUT with independent inputs, defined through its configurable output network which is not shown here. All possible configurations can be found in the vendor's device handbook [37]. Like for Xilinx FPGAs two FAs are provided. In contrast to Xilinx FPGAs, the FAs do not require



(a) Stratix V LAB (b) Simplified block diagram of a Stratix V ALM

Figure 2.3: Intel Stratix V ALM and its clustering to an LAB.

any further ALM logic. Similar to Xilinx FPGAs their carry chain is propagated throughout the LAB to the next ALM to enable fast RCA implementations. Four optionally registered outputs are provided to output the adder results as well as the LUT outputs. A detailed LAB and ALM block diagram can be found in the already mentioned handbook. Although the ALMs can be seen as the BLE, the logic utilization on Intel FPGAs is very often given in adaptive look-up tables (ALUTs), which is the estimated number of required half ALMs needed to fit a design.

Memory Blocks

Current FPGAs contain numerous small memory blocks which are distributed all over the chip. They can either be used as small independent memory blocks or combined to build larger blocks of random access memory (RAM). There are various possibilities to realize the memory blocks starting from simple single-port memory up to dual-port memory with two read/write ports with various memory array dimensions. The total memory available for the designer ranges from 7 to 39 Mbits within the Virtex 6 device family and 23 to 64 Mbits in the Stratix V device family. Alternatively, the memory blocks can be used as large shift registers, function generators or first-in-first-out buffers (FIFOs).

Digital Signal Processing Blocks

As FPGAs are very important in digital signal processing applications, specialized hardware blocks for these applications are part of current FPGAs. As multiplication is the most important operation some more details are given on its implementation. In Virtex 6 FPGAs the DSP block called DSP48E1 slice [38] contains a 25 x 18 bit multiplier. Using DSP blocks including its fixed-size multipliers is thus most favorable when a multiplier with this size is required. For smaller multipliers or multipliers with other input word sizes other implementations [39] are required. Equally important is the composition to larger multipliers. For this purpose, DSP blocks can be combined using dedicated routing paths between several DSP blocks. Doing this hardware efficient is part of current research [40], [41].

In Intel FPGAs the DSP blocks are implemented more flexible. Realizations of 9 x 9 bit, 18 x 18 bit, 27 x 27 bit and 36 x 36 bit multipliers are provided by the variable-precision DSP blocks in the Stratix V device family [37].

Besides the multiplier, a DSP block includes functional units which are typically required in DSP applications. Without using the general FPGA logic, DSP blocks provide, e. g., adders in front and after the multiplier, bit-wise logic functions and large registers.

Routing Delay

Regarding the delay of a circuit, the programmability of FPGAs comes at a price. While in non-programmable integrated circuits local connections have a negligibly small contribution to the overall delay, in FPGAs such a connection can include several routing elements. Moreover, programmable parts within the connection blocks and programmable routing switches add additional delay. A common way to overcome this drawback is to add registers to the initial circuit to split up the combinatorial parts. Introducing registers has to be done systematically to keep the functionality of the original circuit. This is achieved by placing registers in the combinatorial path such that the number of introduced registers on each path from an input to an output is equal. This procedure called pipelining increases the latency of the circuit while keeping its functionality. An overhead of required FPGA resources may occur due to a massiv register insertion. On the other side, it is very likely that unused registers in the FPGA's BLEs, which are already used for logic, will be taken. This was shown, e. g., in [42] by a speedup of 111 % of a non-pipelined circuit on a Cyclone II FPGA with a pipelining overhead of only 6 %. The more recent FPGAs presented above should provide even better results, as they have double the number of registers per BLE.

Power Consumption

With an increase in FPGA resources and maximum clock frequency, which can be observed for each new FPGA generation, the power consumption of an FPGA gets more and more important. The power consumed by an FPGA can be separated into the two components static power and dynamic power. Static means, that the power consumption is independent of the switching activity in the device. The main source of static power consumption is transistor leakage current, which tends to increase with decreasing technology size. This is why static power is getting more and more important and consumes a noticeable amount of overall chip power. Dynamic power consumption

P_{dyn} is related to the actual switching activity. It can be basically summarized as product of the capacitance C of the considered component and the supply voltage V squared, multiplied by the frequency f and the switching activity α.

$$P_{\text{dyn}} = CV^2 f\alpha\,. \tag{2.1}$$

Supply voltage and capacitance are technology dependent, hence, the only way to reduce the dynamic power consumption for a given technology is by reducing the frequency or the switching activity within the circuit. One way to reduce the switching activity is again pipelining. It helps to reduce glitches and thereby dynamic power consumption [43]. In the context of run-time reconfiguration switching activity caused by changing logical functions during reconfiguration should be reduced. In signal processing applications the energy required to compute one data sample is an important figure. It can be calculated by relating the dynamic power P_{dyn} to the sample rate given in $[\frac{\text{samples}}{\text{ns}}]$ as

$$E_{\text{s}} = \frac{P_{\text{dyn}}}{\text{sample rate}} \tag{2.2}$$

Moreover, the reconfiguration energy E_{rec} is important to compare different reconfiguration approaches. It is computed as

$$E_{\text{rec}} = P_{\text{rec}} T_{\text{rec}}\,, \tag{2.3}$$

while P_{rec} is the average power consumption during reconfiguration and T_{rec} is the reconfiguration time.

2.1.2 Design Flow

In the last sections some important aspects of hardware resources, their routing, timing and power consumption on FPGAs were presented. Mapping a system to an FPGA offers many trade-offs with respect to these aspects and requires several steps. These steps are shown in the diagram in Figure 2.4 and explained in the following.

The initial step is the design and simulation of the system for a specific application with specific requirements. This can be an algorithmic description, C/C++ code or a model-based design description in, e.g., Matlab/Simulink or LabVIEW. At this point a decision has to be made which parts of the design should be implemented on an FPGA (hardware). This can be the whole system or only parts of the system if the FPGA is used as co-processor in a hardware-software co-design. After this step, the hardware parts have to be transformed into a representation which can be processed by their synthesis tools (e. g., Xilinx ISE, Vivado, Intel Quartus Prime).

The entry point for the FPGA design flow with these tools can be a hardware description written in a hardware description language (HDL) like very high speed inte-

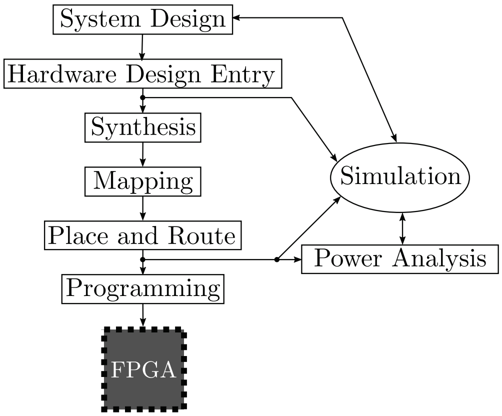


Figure 2.4: Diagram of the steps required to map a system specification to an FPGA.

grated circuit hardware description language (VHDL) or Verilog HDL or a hardware description provided as schematic. All descriptions can include custom blocks called intellectual property (IP) cores, provided by the FPGA vendors or a third party. IP cores are special implementations of a certain function ranging from low complexity operations up to processors. A simulation can be used to verify the functional behavior of the hardware description by comparing it to the initial system design or pre-calculated test data.

The first step performed in the FPGA design flow is the translation of the hardware description into a netlist, called *Synthesis*. A netlist is basically a textual description of the circuit including all components, required IO ports and their interconnection. This step can include target specific optimizations like, for example, removing unused signals or detecting certain functional units. In the next step, the components in the netlist are matched with the physical resources of the target FPGA. This process called *Mapping* is accompanied by some optimizations and design rule checks.

Once the components are mapped to the existing FPGA components, they have to be assigned to specific resources of the FPGA. Moreover, their interconnection has to be realized by the available routing resources. Assignment and interconnection are done during *Place and Route*. In addition to that, a timing analysis is included or implemented as an additional step after *Place and Route*. During the timing analysis the timing information of the final routed netlist, including delays of components, wires, interconnections and routing switches, is collected.

This timing information can be used for a realistic simulation of the circuit represented by the netlist. Moreover, a *Power Analysis* based on timing-accurate switching activity can be performed. For that purpose the netlist including the timing information can be exported and simulated using a test bench. VHDL simulation tools like, e.g., ModelSim [44] are able to output the full simulation data into a file. This file together with the netlist are used as input to the *Power Analysis*.

If all requirements and design constraints are fulfilled, the last step is the *Programming* of the FPGA. For this purpose, the netlist after *Place and Route* is stored as a bit file. Each bit in this file indicates if a specific bit in the FPGA configuration (routing and logic) is set. *Programming* thus means in this context, setting all required configuration bits by sequentially streaming the bit file content into the FPGA.

2.2 Run-Time Reconfiguration on Field Programmable Gate Arrays

Besides the full reconfiguration, referred to as *Programming* in the last section, reconfiguration of parts of the FPGA during run-time has become an important feature of recent FPGAs. There are basically three ways to perform such a run-time reconfiguration on FPGAs:

1. Partial reconfiguration including routing and logic with partial bit files.
2. Logic reconfiguration using run-time reconfigurable LUTs.
3. Routing reconfiguration using multiplexer-based resource sharing.

The investigation of their influence on the required hardware resources, resulting performance, power consumption and reconfiguration time in the context of reconfigurable constant multiplication is a main part of later chapters (logic reconfiguration in Chapter 3, routing reconfiguration in Chapter 4 and Chapter 5). The following three sections provide the required background information and implementation details for the aforementioned methods.

2.2.1 Partial Reconfiguration

A great advantage of FPGAs is that they can be programmed on-site and that they can be re-programmed, instead of a re-fabrication as required for non-programmable integrated circuits. This design flexibility is even higher with Partial Reconfiguration (PR) [45], [46]. PR is available for the latest FPGAs, namely the Intel Arria V, Cyclone V, Stratix V, Arria 10 device families and Xilinx Virtex 4-6, 7 series and UltraScale device families. Mutually exclusive designs can be placed in the same region of the FPGA. For this part of the FPGA, logic functions and routing can be exchanged dynamically, while the remaining parts of the device continue to operate unaffected. To do so, bit files for these reconfigurable regions are created in the design phase by dividing the FPGA into reconfigurable and static logic.

The smallest size of a reconfigurable region in the Xilinx tool flow corresponds to a certain number of CLBs, called configuration frame. A good overview over the

Table 2.1: Frame size for the different Xilinx FPGA families.

FPGA family	CLBs per frame	DSP48 per frame	Block RAM per frame
Virtex 4	16 high by 1 wide	–	–
Virtex 5	20 high by 1 wide	–	–
Virtex 6	40 high by 1 wide	–	–
7 series	50 high by 1 wide	10 high by 1 wide	10 high by 1 wide
UltraScale	60 high by 1 wide	24 high by 1 wide	12 high by 1 wide

composition of a Xilinx bit stream can be found in [47]. The size of a configuration frame for the different Xillinx FPGA families can be found in Table 2.1. The smallest PR design for a Xilinx Virtex 6 FPGA will, for example, block at least (if routable) 40 CLBs, which equals 80 slices. This is important as a reconfiguration port is used to exchange the partial bit files. This causes a certain reconfiguration time. For the Xilinx FPGAs the partial bit file can be loaded, e.g., via the Internal Configuration Access Port (ICAP), which is a 32 bit wide interface running at 100 MHz. This results in a theoretical maximum reconfiguration speed of 400 MB/s. Experimental results by Liu et al. [48] show that a maximum reconfiguration speed of 371.4 MB/s is possible. With a size of 93312 bit per frame, the minimal ICAP reconfiguration time for Xilinx Virtex 6 FPGAs is about 29 µs. Further details on Xilinx PR can be found in the vendor handbooks, e.g., [45].

In the Intel PR flow each LAB, RAM block, DSP block and routing multiplexer can theoretically be reconfigured individually. However, like for Xilinx FPGA the reconfiguration time depends on the size and orientation of the reconfigurable parts of the design. The partial bit file can be loaded, e.g., via the PR Region Controller IP block which provides an up to 32 bit wide configuration data interface. With an interface speed of up to 250 MHz reported in the vendor handbooks [46] considerably small reconfiguration time overheads should be possible with Intel FPGAs if only a small portion of the design has to be reconfigurable.

2.2.2 Logic Reconfiguration using Dynamically Reconfigurable LUTs

In modern Xilinx FPGAs, namely Virtex 5/6, Spartan 6 and the whole 7 series [49] a special dynamically reconfigurable LUT, called configurable look-up table (CFGLUT), is provided. It is available by using the HDL primitive *CFGLUT5* in the Xilinx tools ISE and Vivado, respectively. Reconfigurable means that the LUT contents realizing a certain logic function can be changed during run-time. Fig. 2.5(a) shows the interface of such a CFGLUT, which has five LUT inputs and can be used as a 5-input-1-output logic

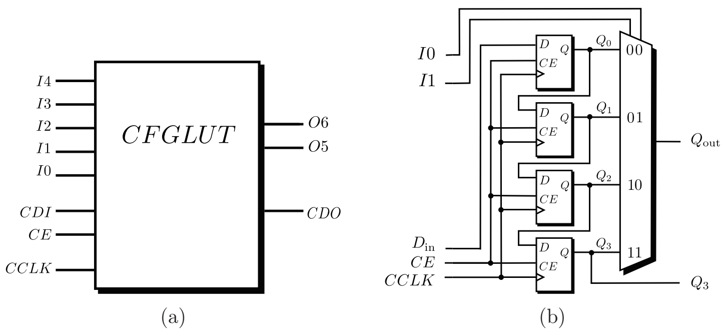


Figure 2.5: Specialized Xilinx FPGA components to enable LUT reconfiguration: (a) interface of a Xilinx CFGLUT, (b) reduced block diagram to show the functionality of a Xilinx shift register LUT.

function or as 4-input-2-output logic function when input $I4$ is tied to a logical one. One CFGLUT utilizes the slice resource of a standard 6-input LUT. It provides a special reconfiguration interface with the signals configuration data in (CDI), configuration data out (CDO), clock enable (CE) and configuration clock (CCLK). A new output function can be loaded by shifting a new 32 bit LUT configuration sequentially into CDI, while CE is tied to a logical one, following the clock CCLK. At the same time, the previous LUT configuration is shifted out of CDO. This property can be used for a serial reconfiguration of several CFGLUTs. Reconfiguration of CFGLUTs in parallel takes 32 CCLK clock cycles, while the time for a serial reconfiguration has to be multiplied by the number of serially connected CFGLUTs. This means, reconfiguration time is in the range of some hundred ns, dependent on CCLK (typically between 200 and 500 MHz) and the selected reconfiguration mode (parallel or serial).

In fact, the implementation of the so called shift register look-up table (SRL) with clock enable, which was already present in older FPGAs, e. g., Virtex II and Spartan 3, is reused. A simplified block diagram of the SRL circuit with only two LUT inputs is shown in Fig. 2.5(b). $\mathrm{D_{in}}$ is the input of a delay chain, which is enabled by CE. A multiplexer is used to select a delayed value of $\mathrm{D_{in}}$ using the inputs $I0$ and $I1$. Replacing $\mathrm{D_{in}}$ by CDI and Q_3 by CDO in Fig. 2.5(b) would lead to a CFGLUT with 4 configuration bits. Thus, SRLs provide the same logic reconfiguration possibility as CFGLUTs. Therefore, a certain backwards compatibility for older Xilinx FPGAs is given for architectures based on CFGLUTs.

For Intel FPGAs the reconfigurable LUT is not available. Shift registers are present, but realized as a chain of ALM output registers. Thus, they can not be used like a CFGLUT and would require a multiple of hardware resources. Block RAM which is present on most of the modern FPGA families could be used instead [50]. This comes with drawbacks in the resulting routing and a lower performance. Moreover, the availability of a sufficient quantity of available RAM could be critical, if other parts of the design are competing for this resource.

2.2.3 Routing Reconfiguration using Logic Multiplexers on FPGAs

The programmable routing of an FPGA is fixed during run-time. However, multiplexers realized with BLEs can be used to change signal routing during run-time. This is especially interesting in the context of switching between the different circuit alternatives for resource sharing like, e. g., in multiplexer-based run-time reconfigurable constant multipliers (RCMs), which are the topic in Chapter 4. Besides their mapping into the soft logic (slices/ALMs) as shown in the following, the mapping of multiplexers onto otherwise unused DSP blocks has been investigated [51], too.

A Virtex 6 slice [36] consists of four 6-input LUTs, which can be used as any 6-input logic function (see Section 2.1.1). Hence, each LUT can be used as an up to 4:1 1-bit multiplexer. Moreover, the Virtex 6 slice includes two 2:1 multiplexers (*MUXF7*) to switch between two of the LUT results, which extends the usage to two up to 8:1 1-bit multiplexers. Finally, there is another 2:1 multiplexer (*MUXF8*) to switch between the outputs of the two *MUXF7s*. This means, four LUTs (= one slice) are required to build an up to 16:1 1-bit multiplexer as shown in Figure 2.6. Two of those 16:1 multiplexers can be combined to a 32:1 1-bit multiplexer utilizing only one additional LUT and so on. Using *Primitives* [36] in the VHDL description makes it possible to use the slices exactly in that way. This results in an optimized multiplexer implementation first proposed by Chapman [52]. The gain of this implementation can be seen in Figure 2.7 with results originally published in [29]. It shows the LUT consumption of the mapping achieved by Xilinx ISE 13.4 (gray) and the improved solution by using *Primitives* (black when better, otherwise equal to ISE mapping) for 2:1 to 16:1 multiplexers. The operating frequency is not shown, due to the fact that only one slices is required, which leads to frequency estimations that are unrealistic for a final design, as there should be more limiting parts elsewhere.

As shown before, an Intel ALUT in a Stratix V ALM can be used to map a variety of different LUT configurations up to a 7-input function in some cases. In the case under consideration each ALUT (half ALM) can implement an up to 4:1 1-bit multiplexer. The 8-input fracturable LUT does not provide a specialized multiplexers implementation within one ALM. Synthesis results showed that large multiplexers are built as a cascade of 4:1 multiplexers. Each cascaded multiplexer except the first one, can process three additional multiplexers. This results in an estimated ALUT consumption of

$$\#\text{ALUTs} = \left\lceil \frac{\#\text{multiplexer inputs} - 1}{3} \right\rceil \tag{2.4}$$

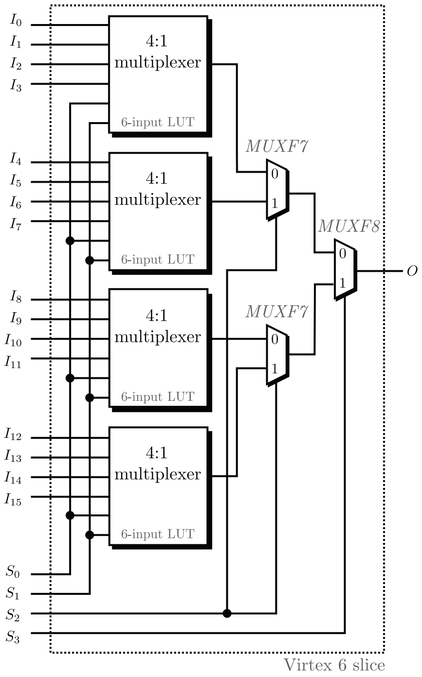


Figure 2.6: Mapping of a 1 bit 16:1 multiplexer into a Virtex 6 slice.

To ensure this assumption is correct, multiplexers with different input word sizes ranging from a 2:1 to a 32:1 were mapped on a Stratix V FPGA using Quartus Prime 15.1. The resulting average values for required ALUTs per bit for up to 32 multiplexer inputs can be found in Figure 2.8. These numbers confirm equation 2.4.

2.2.4 Evaluation of Run-Time Reconfigurable Designs

Important properties for the evaluation of a design implemented on an FPGA are classically its resource usage, performance, and power consumption. The resource usage and power consumption reflect costs of the considered design and can therefore be used to compare different implementations of the same functionality. The required FPGA gets more expensive as the required resources increase. An increased power consumption will limit the battery life time in a mobile application or will in general raise the operating costs. Moreover, the design of the overall system can get more complicated, when the FPGA's peak power consumption becomes critical. The performance, often given as maximum possible operating speed, is a number to decide whether the given implementation is suitable for a certain application or not. However, the required resources are not always independent from a given performance constraint. Pipelining can be used, as described before, to reduce the routing delay, which on the other hand

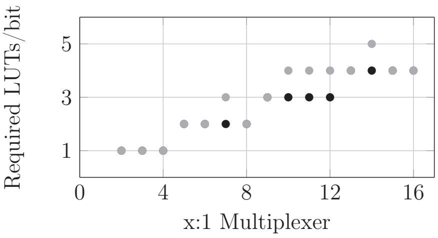


Figure 2.7: Required LUTs per bit for an x:1 multiplexer on a Virtex 6 FPGA. ISE solution (gray) and improvement by *Primitive* usage, using the multiplexer implementation described by Chapman.

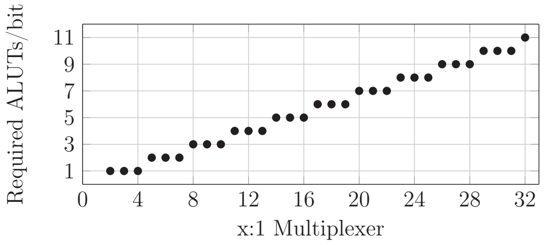


Figure 2.8: Required ALUTs per bit for an x:1 multiplexer on a Stratix V FPGA using Quartus.

can increase the required resources. Therefore, if no specific performance constraint is given, the compared implementations should have a similar performance or, at least, the trade-off between required resources and performance should be shown.

In the context of run-time reconfiguration the reconfiguration time, required configuration memory and reconfiguration power of a design have to be evaluated. Again, a separation into costs and application dependency is possible. The configuration memory and reconfiguration power can increase the overall system's costs by a larger hardware requirement and design complexity, respectively. The required reconfiguration time is application dependent and may exclude some run-time reconfigurable design variant from a certain application.

Resources and Performance

To get realistic numbers for the above-mentioned design properties an FPGA synthesis has to be performed. The resulting resource consumption as well as the maximum possible operating speed can be directly gained from the netlist after place and route. While the provided numbers for the required resources represent the real resource consumption for the given netlist, these numbers are strongly dependent on synthesis options and algorithms. They can hence differ for a slight change in the design description or in the synthesis random seed over different FPGA synthesis runs leading to different netlists. Nevertheless, comparing resources after place and route is the de facto standard when different implementations targeting FPGAs are compared. Moreover, during experimental evaluations it could be observed, that the reported resource consumption is meaningful and in the same range for equally complex designs with slightly different design descriptions [29, 53]. The given numbers after place and route will, therefore, be used to compare different design implementations. In non-pipelined designs the number of required LUTs is often a good representation of required resources. However, for designs containing many registers –like pipelined designs– the utilization of BLEs plays a major role. Therefore, the number of required BLEs (slices/ALMs) should be considered. The maximum possible operating speed provided by the vendor tools is a lower bound estimate. This is sufficient to determine if the analyzed design meets given requirements.

Power Consumption

The power consumption for FPGAs can be estimated using the vendor tools or measured using high precision amplifiers and an oscilloscope [47], [54]. Estimation is done by summing up the power of used resources based on their switching activity (cf. Section 2.1.1). Moreover, the resulting data depends on an FPGA specific capacitance model provided by the FPGA vendors. It was shown by Becker et al. [55] by a comparison to a power measurement that the data gained by the vendor tool Xilinx Power Estimator (XPE) [56] is reliable. Therefore, XPE is used for the analysis of run-time reconfigurable circuits in Chapter 6. For this purpose a netlist after *Place and Route* is simulated using ModelSim [44] using random input data. Then, the simulated switching behavior as well as the netlist are used as input for the power analysis in XPE.

However, XPE does not provide a method to estimate the power consumption of the partial reconfiguration process described in Section 2.2.1. While there are investigations how to model reconfiguration power, e.g. by Bonamy et al. [47], a power estimation tool for PR power is still not available.

Reconfiguration Time and Reconfiguration Memory

The reconfiguration time and reconfiguration memory depend on the used reconfiguration approach. For the multiplexer-based approaches the configuration can be changed from one clock cycle to the other and no reconfiguration memory is required. For the LUT-based approach (cf. Section 2.2.2) the reconfiguration time depends on the reconfiguration mode (serial or parallel) and the required configuration memory. The configuration memory can be calculated in advance based on the number and size of reloadable output functions. Further details on the determination of reconfiguration time and configuration memory are provided in Chapter 3. In the context of PR the configuration time depends on the reconfiguration memory and the used configuration interface. The size of the configuration memory is identical to the size of the partial bit file. However, this bit file is strongly dependent on the number of required configuration frames (cf. Section 2.2.1). This number can not be calculated or estimated in advance and requires an FPGA synthesis. Nevertheless, following the arguments given for the resource consumption, the numbers after place and route can be used for a comparison of reconfiguration time and reconfiguration memory to other approaches.

2.3 Constant Multiplication on Integrated Circuits

After this FPGA-specific introduction to run-time reconfiguration, this section provides the background on multiplier-less constant multiplication. Multiplication with constants is an essential arithmetic operation and used in nearly any DSP algorithm. The implementation of this operation on integrated circuits (ICs) is thus a well studied research topic. Instead of using generic multipliers, constant multiplication is implemented multiplier-less using additions, subtractions and bit shifts. This is advantageous as bit shifts can be realized as wires on ICs and special properties in the constant's number representation can be individually exploited.

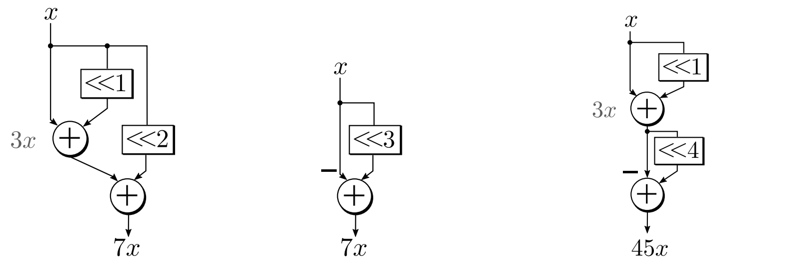
2.3.1 Single Constant Multiplication

Consider two unsigned binary numbers $x = \{x_{B_x-1}x_{B_x-2}\ldots x_0\}$ and $c = \{c_{B_c-1}c_{B_c-2}\ldots c_0\}$. The multiplication of these two numbers is

$$xc = x\sum_{b=0}^{B_c-1} 2^b c_b = \sum_{b=0}^{B_c-1} 2^b x c_b. \tag{2.5}$$

As a c_b of zero has no contribution to the result, multiplication by a constant can be reduced to the sum of all products xc_b for which c_b is one, weighted by 2^b. Note that a multiplication by 2^b corresponds to a left shift by b bit of a binary number. The number of shifted inputs to be added is the number of digits in c which are not zero. This number of so called *non-zeros* can be reduced when a different number representation is used. The representation of a number in the signed digit (SD) representation [2] includes the values 1,0 and -1 for a digit (-1 will be noted as $\overline{1}$ in the following). While the unsigned representation of the decimal number 7 is 111_{bin} (three non-zeros), an SD representation of 7_{dec} is $100\overline{1}_{\text{SD}}$ (two non-zeros). This means, the multiplier-less multiplication by 7 needs two instead of three additions/subtractions when an SD representation is used. The resulting multiplication circuits are shown in Figure 2.9(a) and 2.9(b). A subtraction is shown as addition with a negative input. In the following addition and subtraction will be summarized in the term addition, as the resulting hardware effort is nearly the same for both operations.

The SD representation of a number is not unique, so there are many SD representations of a number. Moreover, the number representation can contain patterns, leading



(a) $7x$ (binary) (b) $7x$ (signed digit) (c) $45x$ (sub-expression sharing)

Figure 2.9: Multiplier-less constant multiplication with different realization methods.

to fewer required adders in the final constant multiplier. The number 45, for example, has the number representations (only a subset of all possible representations) 101101_{bin} and $10\bar{1}0\bar{1}01_{\mathrm{SD}}$. A direct implementation requires three additions, as the number of non-zeros is four. However, if common patterns are used, the number of additions can be reduced to two: for 101101_{bin} the pattern $101_{\mathrm{bin}} = 5$ can be used to compute $45x = (2^3+1)\,5x = (2^3+1)\,(2^2+1)\,x$ and with $10\bar{1}_{\mathrm{SD}} = -\bar{1}01_{\mathrm{SD}} = 3$, $10\bar{1}0\bar{1}01_{\mathrm{SD}}$ can be expressed as $45x = (2^4-1)\,3x = (2^4-1)\,(2^1+1)\,x$ (cf. Figure 2.9(c)). In the latter two examples so called sub-expression sharing [3] is used and the resulting circuits require one addition fewer than the direct implementations. Moreover, in the example case the sub-expression for $3x = (2^1+1)x$ was preferred as this is leading to a smaller word size in the first required adder. In general, sub-expression sharing can lead to better results in terms of required additions, but finding the optimal solution is not guaranteed. This results from the fact that potentially good solutions are omitted by sub-expression sharing, because of so called hidden non-zeros. A deeper analysis of this topic is provided by Faust et al. [4].

The problem of finding an optimal realization of a constant multiplication was shown to be NP-complete by Cappello and Steiglitz [57]. This means, optimal solutions can only be found for a limited problem size, which is a limited constant bit widths in the constant multiplication case. Nevertheless, optimal SCM solutions in terms of required additions for all constants of up to 12 bit using up to four additions were found by Dempster et al. [5]. These results were extended by using up to five additions to constants of up to 19 bit by Gustafsson et al. [6] and further extended for constants of up to 32 bit by Thong et al. [7]. These solutions cover most of the relevant applications. As sub-expression sharing is not able to provide optimal solutions, the aforementioned solutions were generated by an exhaustive search using so called adder graphs with different numbers of adders. An adder graph G is a directed acyclic graph (DAG) in which each vertex, except the input, represents an adder computing a certain multiple of the input, called fundamental. The edges can contain weights, which represent a certain bit shift. Positive numbers are left bit shifts and negative numbers are right bit

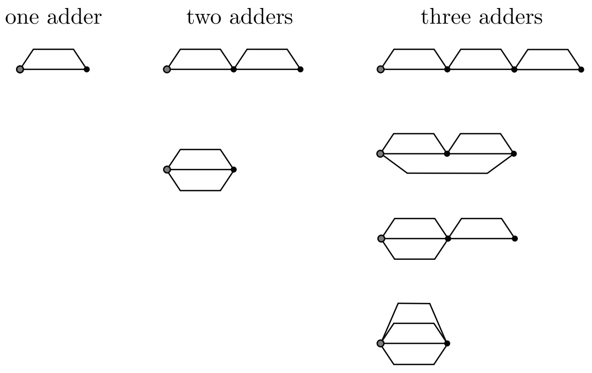


Figure 2.10: Adder topologies for up to three adders from Gustafsson et al. [5].

shifts. All node fundamentals can be formally represented as $\mathcal{A}$-operation [8], which is defined as

$$\mathcal{A}_q(a_1, a_2) = |2^{l_1}a_1 + (-1)^{\phi}2^{l_2}a_2|2^{-r} \tag{2.6}$$

with $q = (l_1, l_2, r, \phi)$, where a_1 and a_2 are the input fundamentals, l_1, l_2 are the bit shifts at the inputs and r is the bit shift at the output. The sign bit $\phi \in \{0, 1\}$ denotes whether an addition or subtraction is performed. Starting with the input fundamental 1, all output fundamental values using one adder are computed by evaluation the $\mathcal{A}$-operation with input 1. Then, all resulting fundamentals and the input are used to determine all output fundamentals using two adders. This is repeated until the desired output fundamental is reached. It was shown by Dempster et al. [5] that each graph with even fundamentals can be transformed into an equivalent graph with only odd fundamentals, as all even constants can be generated by left-shifting odd constants. Moreover, negative fundamentals are not included in the search as their negation is assumed to be done by changing adders to subtractors and vice versa in the succeeding circuit. The odd and positive fundamental property reduces the search complexity. However, there are several topologies to connect the adder outputs and inputs when more than one adder is required to compute the output fundamental. An exhaustive search thus includes the evaluation of all possible topologies. All possible topologies for up to three adders provided by Gustafsson et al. [6] are shown as vertex reduced graph in Figure 2.10. The input is drawn as gray circle, adders as black circles and the edges as simple lines (from left to right). Note that adders with more than two inputs will be realized using multiple two-input adders. Topologies for a larger number of adders and further topology considerations can be found in [5–7].

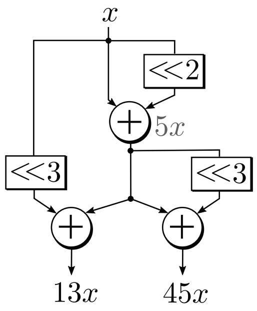


Figure 2.11: Example of an MCM circuit for the multiplication $45x$ and $13x$ with reused intermediate result.

2.3.2 Multiple Constant Multiplication

In the last paragraph, the multiplication with a single constant was considered. In many applications like, e. g., digital filters the multiplication of an input by several constants, called multiple constant multiplication (MCM), is an important operation. It can be formalized as multiplication of an input x by N constants resulting in a vector

$$(y_0, y_1, \ldots, y_{N-1})^T = (c_0, c_1, \ldots, c_{N-1})^T x \tag{2.7}$$

In multiplier-less MCM circuits using addition and bit shifts, intermediate results can be reused among several output constants' adder graphs. This reduces the required hardware resources and computation effort compared to several separate single constant multiplication (SCM) implementations. In Figure 2.11 an example of the multiplication circuit for $45x$ and $13x$ is shown. Taking the number representations $45 = 101101_{\text{bin}}$ and $13 = 1101_{\text{bin}}$, the intermediate result 5 which has the common pattern 101_{bin} can be reused in both products.

Sharing intermediate results among several output constants is a generalization of finding SCM solutions. Thus, finding MCM solutions is consequently NP-complete, too. Nevertheless, some optimal methods exist, which can be used for small and intermediate problems sizes [9–12]. Moreover, there are very good heuristics to find the best possible solutions [3, 8, 13–16]. A good overview over the different MCM approaches is provided in [11] and [53]. In general, there are two basic strategies how to generate MCM solutions. The first strategy is using common sub-expressions in the number representations of the different output constants. As stated before, this reduces the search space due to hidden non-zeros. Hence, finding an optimal solution cannot be guaranteed. However, there are many MCM algorithms based on that idea, starting from Hartley [3] and Potkonjak et al. [14] in 1996. The second strategy is again using graphs based on the $\mathcal{A}$-operation (2.6) or comparable formulations of the graph node's operation. Finding an optimal MCM solution means here, finding an adder graph G with a minimum number of adders, such that all required output constants can be implemented. Starting from the required output constants, called target set, the minimum number of intermediate constants has to be found, such that the target set can be implemented. Methods to find optimal solutions were proposed and shown to be valid for small problems, e. g., by Aksoy et al. using breadth-first [10] and depth-first [11] search and Gustafsson [9] using hyper graphs. Moreover, an optimal approach for pipelined MCM was proposed by Kumm et al. [12]. Important heuristics to find MCM solutions are the reduced adder graph (RAG-n) algorithm proposed by Dempster and Macleod [13], the $\mathrm{H}_{\mathrm{cub}}$ algorithm proposed by Voronenko and Püschel [8] and the difference adder graph (DiffAG) algorithm proposed by Gustafsson [15]. Furthermore, Kumm et al. proposed a heuristic to generate pipelined MCM solutions called reduced pipelined adder graph (RPAG) algorithm [16]. The latter one is the most important approach for the method presented in Chapter 4, as it is considering pipelining during optimization and is thus well-suited for FPGAs. Therefore, the next section includes a brief introduction into the basic concepts of RPAG as an example MCM heuristic.

2.4 Implementation of Constant Multiplication on FPGAs

This section presents two different methods to efficiently implement constant multiplication on FPGAs. First, the already mentioned RPAG algorithm to generate pipelined SCM, MCM and constant matrix multiplication (CMM) circuits is presented. This algorithm is presented and analyzed in detail in [16] and [53] and its source code is available online [33]. However, this section gives a brief introduction into the most important concepts of the RPAG algorithm which are highlighted by using bold face. Moreover, LUT-based constant multiplication is introduced as alternative to addition and bit-shift-based multiplier-less constant multiplication. Due to its basic concept of using LUTs as partial product generators, it is well-suited for FPGAs.

2.4.1 Reduced Pipelined Adder Graph Algorithm

Like introduced before, the results of the RPAG algorithm are adder graphs representing pipelined constant multipliers using additions and bit shifts only. Previous approaches for pipelined MCM applied pipelining to existing MCM solutions. However, in the RPAG algorithm pipelining is considered already during the MCM adder graph generation. This includes that the optimization starts from the required output constants, which is not the case for most of the other existing SCM and MCM methods (see Section 2.3.1). Starting from the outputs is used to force a **minimum adder depth** of the resulting circuit [16]. The adder depth is the maximum of the number of adders on each path from the input to the output(s). Forcing the minimum adder depth, which can be calculated in advance [4], for pipelined implementations seems to be reasonable as it tends to reduce the number of required pipeline balancing registers. Moreover, adder costs as well as **costs for pipeline registers** can be considered during optimization. Using the set of output constants, the RPAG algorithm is **backward-exploring** reachable intermediate constants by evaluating the $\mathcal{A}$-operation (2.6) in a step-wise greedy search until the input is reached. The goal of the heuristic is to select the intermediate constants which can be **reused most frequently** for other constants of the current exploration step and have the lowest implementation costs. This is done by a four step process using different sets:

I. The target constants are normalized to odd numbers and the overall minimum adder depth is calculated. This minimum adder depth equals the number of pipeline stages S in the final MCM solution. The normalized target constants are inserted into the set $X_s = X_S$ in which the information of realized constants in stage s is stored.

II. For the current stage s a working set W is filled with the elements of X_s. At the same time a set of predecessors P is cleared.

III. The predecessors for the constants in W are determined and evaluated based on five topologies. When determining the predecessors, the working set element can be computed by one predecessor or by two predecessors. The first case is preferred. Finally, the preceding constant or pair of constants with the highest reuse frequency and lowest implementation costs is added to P. All constants in W which can be computed by the elements in P are then removed from W. This step is repeated until W is empty.

IV. Now, the elements of P are moved to the set of realized constants of the previous stage X_{s-1} and the algorithm proceeds with the steps II and III with $s \leftarrow s - 1$. The algorithm stops when all stages are determined.

Using this concept can directly take advantage of the registers present in an FPGA's BLE. It was shown by Kumm [53], that the pipeline aware MCM optimization used in the RPAG algorithm outperforms other MCM approaches like $\mathrm{H}_{\mathrm{cub}}$ [8] when their results are optimally pipelined like, e. g., done in [17].

Constant Matrix Multiplication

Moreover, the RPAG algorithm was the first algorithm which was able to generate pipeline optimized solutions for CMM [58]. This operation is defined as the multiplication of an input vector $\mathbf{x} = (x_0, x_1, \dots, x_{M-1})^T$ with a constant matrix

$$\mathbf{C} = \begin{pmatrix} c_{1,1} & \dots & c_{1,M} \\ \vdots & \ddots & \vdots \\ c_{N,1} & \dots & c_{N,M} \end{pmatrix}. \tag{2.8}$$

The output is a vector $\mathbf{y} = (y_0, y_1, \dots, y_{M-1})^T = \mathbf{C}\,\mathbf{x}$, where each element is a weighted sum of the inputs. Again, intermediate results can be shared among different outputs. The main difference to the optimization for SCM and MCM is, that an adder is used to add intermediate results of different inputs. Therefore, the $\mathcal{A}$-operation has to be redefined for vectors as well as the minimum adder depth, which now depends on the number of non-zeros in the whole vector. Using these adaptions, RPAG offers a direct optimization heuristic for the CMM operation. An example of this operation can be found in Section 4.2.2. Further details on the implementation of the CMM operation on FPGAs and ASICs can be found in related work, e. g. [53, 58–61]

2.4.2 Look-Up Table Based Constant Multiplication

So far, the focus of the introduction of multiplier-less constant multiplication has been on implementations using additions and bit shifts. Alternatively, constant multiplication can be performed by dividing the multiplication into partial products. These partial products are then realized with LUTs or block RAMs. The basic concept known as KCM was described by Chapman [30], detailed in [62] and further extended by pipelining [12]. This method is especially interesting for FPGAs, due to their LUTs in the BLEs and the presence of block RAM on nearly every recent FPGA. Again, the multiplication of a number x by a constant c is considered. As the sign bit for signed multiplication is important it is included in the derivation from the beginning. The two's complement representation of a signed number x with a width of B_x bit is

$$x = -2^{B_x-1} x_{B_x-1} + \sum_{b=0}^{B_x-2} 2^b x_b. \tag{2.9}$$

The multiplication by a constant c with a width of B_c can be written as

$$cx = c\left(\sum_{b=0}^{B_x-2} 2^b x_b - 2^{B_x-1} x_{B_x-1}\right) \tag{2.10}$$

and further divided into K products using partial sum terms of length L. For this step is assumed that $B_x = KL$. The resulting partial sum term is

$$\begin{aligned} cx =& c\sum_{b=0}^{L-1} 2^b x_b + c\sum_{b=L}^{2L-1} 2^b x_b + \dots \\ & \dots + c\left(\sum_{b=(K-1)L}^{KL-2} 2^b x_b - 2^{KL-1} x_{KL-1}\right) . \end{aligned} \tag{2.11}$$

The lower bound of summation for each partial product can be normalized to $b = 0$ by a bit shift with multiples of L, which results in

$$\begin{aligned} \underbrace{cx}_{B_c \times B_x} =& \underbrace{c\sum_{b=0}^{L-1} 2^b x_b}_{B_c \times L \text{ mult.}} + 2^L \underbrace{c\sum_{b=0}^{L-1} 2^b x_{b+L}}_{B_c \times L \text{ mult.}} + \dots \\ & \dots + 2^{(K-1)L} \underbrace{c\left(\sum_{b=0}^{L-2} 2^b x_{b+(K-1)L} - 2^{L-1} x_{KL-1}\right)}_{B_c \times L \text{ mult.}} . \end{aligned} \tag{2.12}$$

This step is required to realize the partial product in an L-input LUT. Finally, the constant multiplication is performed by realizing each of the K partial products in a LUT with 2^L pre-calculated products. The outputs of these LUTs are shifted by the appropriate weight and added together as shown in Figure 2.12. One LUT for each of the output bits of the partial products has to be provided (simplified in Figure 2.12). The resulting output word size B_{LUT} of each partial product is the sum of the constant's word size B_c and L

$$B_{\text{LUT}} = B_c + L \ , \tag{2.13}$$

as each partial product LUT performs a $B_c \times L$ bit multiplication. A left bit shift is indicated in Figure 2.12 by an arrow from top to bottom. The final summation of shifted LUT outputs is shown as pipelined adder tree, but could be realized in any way, e. g., as cascaded array of carry-propagate adders like done in [62]. For a signed multiplication a special LUT for the most significant partial product has to be used. In this LUT the most significant bit has a significance of $-2^L c$. Moreover, a sign-extension in the summarizing adder tree has to be implemented. For a direct mapping to an FPGA the length L of the partial products has to be adapted to the target FPGA's LUT input count. An example of the architecture of a 8×4 bit LUT-based signed multiplier with an $L = 4$ is shown in Figure 2.13. The introduced pipeline registers are already present in the BLEs used as partial product generator or as adder. Only a small number of pipeline balancing registers is required, like, e. g., the four registers of

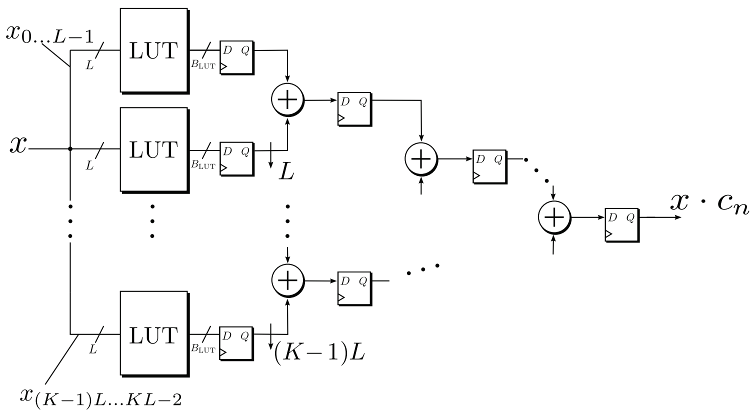


Figure 2.12: LUT-based constant multiplication using a pipelined adder tree.

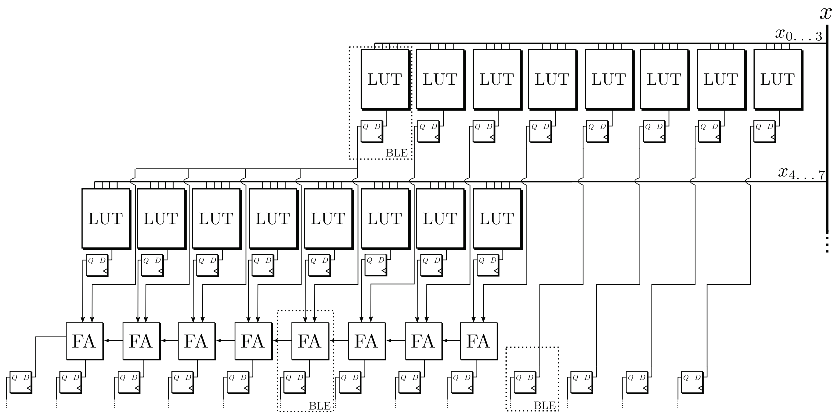


Figure 2.13: Example implementation of an 8×4 bit LUT-based signed multiplier.

the least significant bits in the last stage of Figure 2.13. That means, pipelining costs almost nothing in terms of hardware resources.

3 Reconfigurable Constant Multiplication using LUTs

In this chapter a reconfigurable constant multiplication method using look-up tables (LUTs) is presented. It is based on constant multiplication using LUTs known as KCM described by Chapman [30] (cf. Section 2.4.2). Reconfiguration is achieved by changing the LUT contents of partial products during run-time. This is achieved by using the logic reconfiguration in Xilinx FPGAs presented in Section 2.2.2 and was originally published in [63]. After a short introduction of related work, the architecture of the reconfigurable LUT multiplier is presented and compared to a generic multiplier implementation and constant multipliers reconfigured using Partial Reconfiguration (PR) and the Internal Configuration Access Port (ICAP). Finally, an application of the presented reconfiguration concepts is shown using run-time reconfigurable finit impuls response (FIR) filter architectures.

3.1 Related Work

The application of reconfigurable LUTs for run-time reconfiguration of KCM has not been proposed before. However, in a previous approach on reconfigurable KCM the use of random access memory (RAM) for the partial products, to be able to switch between several constants during run-time, was proposed by Jamro et al. [50, 64]. The implementation called DKCM by Jamro, includes an overhead for read and write access of the required RAM blocks, when these are used for reconfiguration. This is an area overhead when dual port RAM is used and an area and circuit delay overhead when single port RAM and address multiplexers are used. These drawbacks are not present with the use of configurable look-up tables (CFGLUTs) proposed here, as changing the LUT content is independent of reading the LUT content (cf. Section 2.2.2). An approach for reconfigurable LUT-based constant multiplication using shift register look-up tables (SRLs) was developed by Hormigo et al. [65] concurrently to the approach described here. Although older FPGAs and another low-level block type were used, their contributions confirm the benefit of the reconfiguration principle presented in this chapter.

Contribution

The following section presents a new LUT-based method using low level FPGA features for run-time logic reconfiguration of constant multiplication. In addition to that, two new reconfigurable FIR filter architectures based on CFGLUTs are introduced. Moreover, important trade-off points in the design space for run-time reconfiguration on FPGAs are provided.

3.2 Dynamically Reconfigurable Multiplication using LUTs

The LUTs of the constant multiplier presented in Section 2.4.2, Figure 2.12 can be replaced by reconfigurable LUTs presented in Section 2.2.2. To be able to do this replacement, some preliminary decisions have to be made.

3.2.1 Architectural Preliminaries

Output Word Size

In contrast to the constant multiplier without reconfiguration, like presented before, the structure and word sizes of the reconfigurable multiplier are forced to be equal for any realized constant. Its size is predetermined by the maximum constant c_{max}, which should be supported by the multiplier. Therefore, the output word size B_{LUT} equals

$$B_{\mathrm{LUT}} = B_{c_{\mathrm{max}}} + L \ , \tag{3.1}$$

while $B_{c_{\mathrm{max}}}$ is the maximum constant's word size and L is the partial product generator's input size.

LUT Input Size

Besides the output word size B_{LUT}, the LUT input size L of the partial product generators has to be fixed. Referring to the two different ways to use a CFGLUT (see Section 2.2.2), the question is, whether a 5-input LUT or a 4-input LUT should be chosen to realize the reconfigurable multiplier. Assuming that the input word size B_x is divisible by L, the number of required LUTs for K partial products is

$$\eta_{\mathrm{LUT}} = K B_{\mathrm{LUT}} = \frac{B_x}{L}(B_c + L) \ . \tag{3.2}$$

For a 5-input LUT this results in

$$\eta_{\mathrm{CFG}_5} = \frac{1}{5} B_x B_c + B_x \ . \tag{3.3}$$

Table 3.1: LUT contents for pre-calculation of constant c_n $(L = 4)$.

LUT address	MSB LUT	other LUTs
0000	$0 \cdot c_n$	$0 \cdot c_n$
0001	$1 \cdot c_n$	$1 \cdot c_n$
...	...	...
0111	$7 \cdot c_n$	$7 \cdot c_n$
1000	$-8 \cdot c_n$	$8 \cdot c_n$
1001	$-7 \cdot c_n$	$9 \cdot c_n$
...	...	...
1111	$-1 \cdot c_n$	$15 \cdot c_n$

In case of a 4-input LUT with shared inputs, each CFGLUT can be used as two basic LUTs. Assuming that the number of required LUTs is divisible by two leads to a number of required CFGLUTs for $L = 4$ of

$$\eta_{\mathrm{CFG}_4} = \frac{1}{2}\left(\frac{1}{4}B_x B_c + B_x\right) = \frac{1}{8}B_x B_c + \frac{1}{2}B_x \; . \tag{3.4}$$

Hence, using 4-input LUTs is always better. Note that for $L = 4$ the 32 bit configuration of the CFGLUT has to be assembled from two corresponding 16 bit configurations, realizing two independent output functions and partial product bits, respectively.

3.2.2 LUT Reconfiguration

There are two ways to provide the LUT contents for reconfiguration. First, the appropriate LUT contents can be pre-calculated and stored in the design phase (offline). Second, the appropriate LUT contents can be calculated online. Moreover, the reconfiguration can be done serially or in parallel. These aspects are considered in the following.

Pre-calculated Configurations

The values in the pre-calculated configuration equal multiples of the required output constant c_n as shown in Table 3.1. All partial product LUTs except the most significant bit (MSB) LUT use the same configuration data, because they all represent the multiplication of one 4 bit input with the same constant. Only the MSB LUT has to be reconfigured with special data because of the sign bit. The generated LUT configurations are stored in block RAM. The required constant can be loaded bitwise

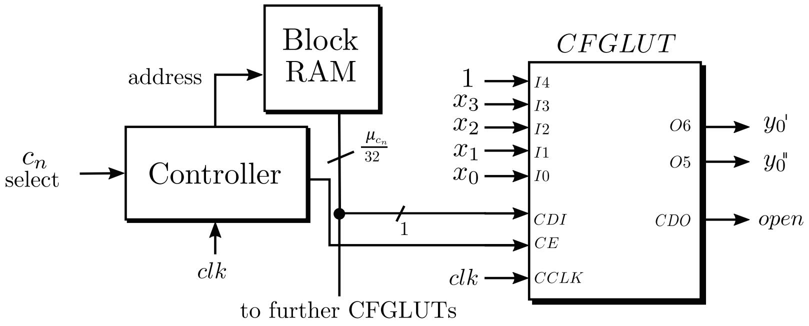


Figure 3.1: Architecture for the configuration update using pre-calculated configurations.

from a dedicated address. The required memory size μ for one constant c_n is calculated according to (3.1) as

$$\mu_{c_n} = 32 \cdot 2 \cdot \frac{B_{\mathrm{LUT}}}{2} = 32\,(B_{c_{\max}} + 4)\ . \tag{3.5}$$

Note that the required memory is independent of the number of partial products K and, thus, is independent of the input word size B_x of the multiplier. The LUT contents of all partial product LUTs are merged to a matrix with $B_c + 4$ columns ($\widehat{=}\,2\,B_{\mathrm{LUT}}$) and 32 rows (configuration bits) and stored. A LUT configuration for a new multiplication constant can be loaded via the configuration data in (CDI) signal of the CFGLUT interface (cf. 2.2.2). Reconfiguration can be done by enabling the reconfiguration interface and selecting the address pointing to the top element of the desired LUT configuration in RAM. In each of the 32 clock cycles after the interface has been enabled the current LUT content is written into the configuration of the CFGLUTs and the address pointer is moved to the next row of the reconfiguration matrix. After 32 clock cycles the output of the reconfigurable LUT multiplier is valid again. Fig. 3.1 shows the associated reconfiguration architecture for one CFGLUT with $L = 4$ inputs and two output bits of an exemplary partial product y. The logical '1' at input I_4 indicates that the CFGLUT is configured to have two output functions. The block RAM contains the pre-calculated configurations. The controller decodes the address of the selected constant and controls the reconfiguration process. During the reconfiguration process, the output of the multiplier is not valid. This problem can be fixed by duplicating each CFGLUT. In doing so, one LUT can be reconfigured with the new configuration while the other LUT is processing data. A multiplexer can be used to switch between the two CFGLUTs to achieve a glitch free data processing. This comes, however, at the cost of doubling the required CFGLUTs and additional multiplexers.

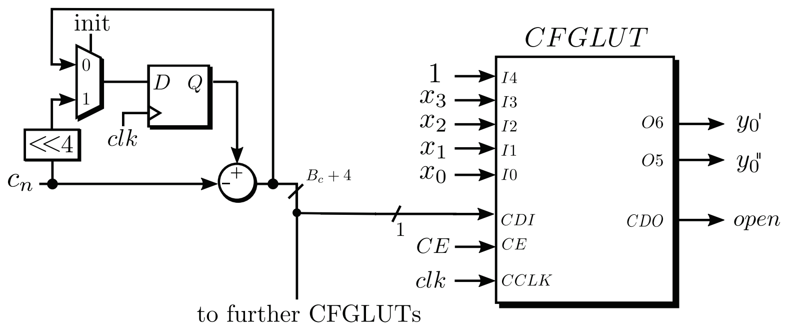


Figure 3.2: Reconfiguration circuit of one CFGLUT for the online update mechanism.

Online Configuration Update

The number of available pre-calculated configurations is limited by the available memory. If this is a problem, the configuration for a certain constant can be calculated online using a subtractor, a multiplexer and a register as shown in Figure 3.2. The shown circuit has to provide the values given in Table 3.1 starting with the last row, as the MSB is shifted through the whole LUT. Therefore, the circuit is initialized with the constant shifted by 4. In the following 16 reconfiguration cycles the constant c_n is subtracted from the previous configuration value. This process generates the sequence $\{15c_n, 14c_n, \ldots, 1c_n, 0c_n\}$ which is passed to the CDI ports of the CFGLUTs. Like before, each CFGLUT requires a 1 bit configuration input. For each CFGLUT a bit with a certain significance is tapped from the $B_c + 4$ bit configuration output. The whole online update procedure has to be repeated once to fill the upper half of the CFGLUT for a second partial product generation, because each CFGLUT has 32 bit of configuration memory. The upper half (LSBs) for output O_5 and the lower half (MSBs) for output O_6. Again, a controller (not shown) assures valid data processing by controlling clock enable (CE) and the initialization. For the MSB LUT the circuit looks similar. However, the initialization is $0c_n$. This produces the required sequence $\{-1c_n, -2c_n, \ldots, -8c_n, 7c_n, \ldots, 1c_n, 0c_n\}$, if the adder's output word size is limited to B_{LUT} bit.

While for this reconfiguration variant no memory is required, a resource overhead has to be introduced for the configuration update circuit. On the other hand, arbitrary constants not exceeding the maximum word size $B_{c_{\max}}$ can be loaded. This is not the case for the pre-calculation variant.

Serial and Parallel Reconfiguration

The reconfiguration interface of a CFGLUT can be used to perform a serial or a parallel reconfiguration (see Section 2.2.2). The parallel reconfiguration is very fast. Only 32 clock cycles are required for a full reconfiguration. On the other hand, a large routing effort may be necessary as each CFGLUT has to be connected to a global CDI signal. This is not the case for the serial reconfiguration in which all CFGLUTs are connected using the provided configuration data out (CDO) signal. However, in this case the required parallel reconfiguration time has to be multiplied by the number of chained CFGLUTs. In the end, the reconfiguration time has to be traded off against the routing effort based on a given application's constraints.

3.2.3 Implementation

A VHDL code generator was written in Matlab to generate the code for the reconfigurable LUT multiplier architecture as well as the reconfiguration data and the code for the reconfiguration interface. Using Matlab is advantageous as it supports many matrix and table manipulation functions by default. These are required to generate and manage the reconfiguration data, which consists of tables (cf. Table 3.1). On the other hand, no special computational performance is required for the code generation as the structure of the KCM-based RCM is very regular (cf. Figure 2.12). The code generation itself was done using string arrays, which can be concatenated and managed very conveniently in Matlab. Finally, the VHDL code was printed into a file using a standard *printf* function supported by Matlab. The inputs of the code generation are the file name, the required constants, the input word size B_x and the configuration update mechanism. The output is a VHDL file.

3.2.4 Experimental Evaluation

This section provides synthesis results to evaluate the reconfigurable KCM-based LUT multiplier design within multiplication circuits on FPGAs concerning the resource usage, speed, reconfiguration time and required memory. Several multipliers with different input word size B_x and constant word size B_c were simulated using ModelSim and synthesized with Xilinx ISE 13.4 for a Virtex 6 FPGA. As reference, generic multipliers (two non-constant inputs) were created using the Xilinx CORE Generator [66] with the default pipeline depth settings (resulting depth was 3) and the specification to use LUTs. These can be reconfigured by switching one of the two inputs between several constants. To have a fair comparison, the most compact way for many constants is to use a register with a control architecture similar to the one used for the dynamically reconfigurable multiplier. Besides this, signed constant multipliers using distributed RAM were generated with the same tool. They can be reconfigured via Xilinx Partial Reconfiguration (PR)(cf. Section 2.2.1) using the ICAP and were taken as a reference, too. As the resource consumption of a constant multiplier heavily depends on the constant's value, constant multiplier instances with ten different constants were created for each $B_x \times B_c$ combination.

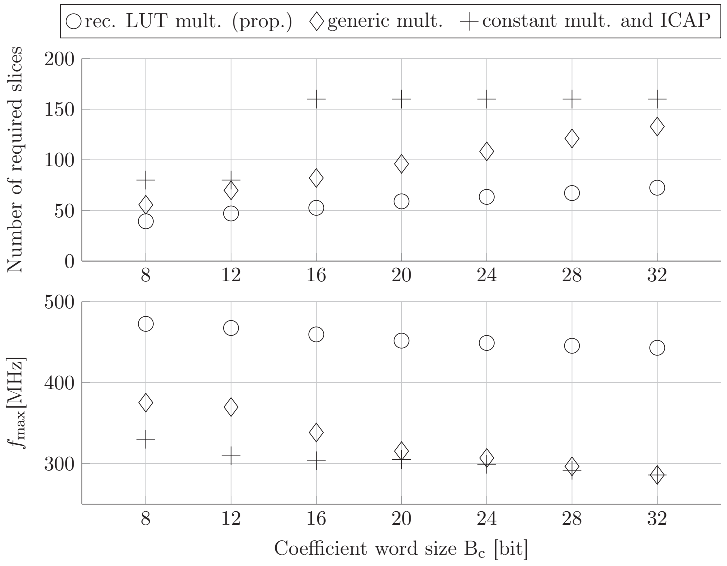


Figure 3.3: Comparison of slice utilization and frequency of the proposed reconfigurable KCM-based design with a generic multiplier and a reconfigurable constant multiplier using ICAP. The shown values are average values for variations of the input word size B_x over the coefficient word size B_c

Slice Utilization and Operating Speed

A comparison of the slice utilization and the operating speed of the proposed reconfigurable KCM-based LUT multiplier design with a generic multiplier and a reconfigurable constant multiplier using ICAP is shown in Figure 3.3. It illustrates the synthesis results after place and route by average values for variations of the input word size B_x over the coefficient word size Bc. The concrete data of the synthesis results can be found in Appendix B, Table B.4 for the sake of the clarity. When PR is applied to the constant multipliers, the resulting slice utilization is based upon the largest design of a given $B_x \times B_c$ combination. This has to be mapped to an FPGA frame (cf. Section 2.2.1). Therefore, the number of slices which has to be reserved on the FPGA for a PR design is calculated as

$$\eta_{\text{slpr}} = \left\lceil \frac{\eta_{\text{slraw}}}{\kappa_{\text{sl}}} \right\rceil \kappa_{\text{sl}} \tag{3.6}$$

while η_{slraw} is the number of required slices without PR and κ_{sl} is the capacity of a frame, which is $\kappa_{\text{sl}} = 80\frac{slices}{frame}$ for the used Virtex 6 FPGA. The maximal working frequency in this case depends on the slowest design. Therefore, only the maximal values of slices and minimal values of $f_{\max}$ of the generated constant multipliers are presented.

The proposed reconfigurable KCM-based LUT multipliers provide the solutions with the lowest slice consumption. Beyond that, the reconfigurable KCM-based LUT multipliers are on average 53 % faster than a reconfigurable constant multipliers using ICAP. In comparison with the generic multiplier the KCM-based LUT multipliers only need 55 % of the slice resources and are 49 % faster on average, which is a great benefit. The related memory utilization and reconfiguration time are analyzed in the following sections.

Memory Utilization

As shown in Section 3.2, the memory utilization of one coefficient for the reconfigurable KCM-based LUT multiplier using pre-calculated configurations only depends on the coefficient word size $B_{c_{\max}}$. For a given number of different coefficients n_c the memory utilization is:

$$\mu_{\text{rlutm}} = n_c \mu_{c_n} = n_c \cdot 32(B_{c_{\max}} + 4) \tag{3.7}$$

The smallest Virtex 5 FPGA [67] offers $1,152$ kbit of block RAM which corresponds to $1,000$ different 32 bit coefficients. With the largest Virtex 7 FPGA [68] about $44,000$ different 32 bit coefficients are possible. For the generic multiplier the required RAM memory is

$$\mu_{\text{gmult}} = n_c B_{c_{\max}} \ . \tag{3.8}$$

In the case of the constant coefficient multiplier using ICAP the number of different coefficients leads to a larger memory utilization for the design reconfiguration. Besides the bit file of the static design, each configuration of a replaceable constant coefficient multiplier adds a partial bit file to the full design configuration. The size of this partial bit file depends on the number of required slices of the largest constant multiplier. These have to be mapped into the reconfiguration frames, of which each additional frame with a capacity of κ_{sl} slices adds B_{frame} bit to the partial bit file (cf. Section 2.2.1). So in the best case, the required reconfiguration memory is

$$\mu_{\text{cmult}} = (n_c - 1) \left\lceil \frac{\eta_{\text{slraw}}}{\kappa_{\text{sl}}} \right\rceil B_{\text{frame}} \ . \tag{3.9}$$

The used Virtex 6 FPGA has a capacity of $\kappa_{\text{sl}} = 80 \frac{slices}{frame}$ and each frame adds $B_{\text{frame}} = 93,312$ bit to the partial bit file. Examples of the required reconfiguration memory and reconfiguration times of the considered multipliers are shown in Table 3.2.

Table 3.2: Reconfiguration memory μ_x(for one configuration) and reconfiguration times t_{rec} for the evaluated designs.

	rec. LUT mult. (prop.)		generic mult.		constant mult. + ICAP	
$B_x \times B_c$	μ_{rlutm}	t_{rec}	μ_{gmult}	t_{rec}	μ_{cmult}	t_{rec}
8 × 8	384 bit	56.80 ns	8 bit	2.77 ns	93,312 bit	29.16 μs
16 × 16	640 bit	60.02 ns	16 bit	4.20 ns	93,312 bit	29.16 μs
24 × 24	896 bit	82.26 ns	24 bit	3.72 ns	186,632 bit	58.32 μs
32 × 32	1,152 bit	78.24 ns	32 bit	5.71 ns	186,632 bit	58.32 μs

Reconfiguration Time

The time that is needed to reconfigure the multiplier is an important requirement in time-critical applications. The fastest reconfiguration can be provided by the generic multiplier architecture, whose coefficient can be changed within one clock cycle. For the presented reconfigurable KCM-based LUT multiplier design 32 reconfiguration clock cycles are required until the output of the multiplier is valid again (90.65 ns for the slowest design). This is very fast compared to the PR concept offered by Xilinx via the ICAP. There, the reconfiguration time depends on the size of the partial bit file replacing the currently running constant multiplier bit file. With a maximum reconfiguration clock rate of 100 MHz and 32 bit data width [45] the reconfiguration time for a given partial bit file size μ_{cmult} is

$$t_{\text{rec}} = \frac{\mu_{\text{cmult}}}{400\frac{MB}{s}} \; . \tag{3.10}$$

The reconfiguration times for four representative designs where evaluated by creating partial reconfigurable designs using the Xilinx tool PlanAhead. Following the given design flow resulted in the partial bit file sizes listed in Table 3.2. The resulting reconfiguration times are a hundred times higher than those of the generic multiplier and the proposed reconfigurable KCM-based LUT multiplier. To illustrate that the proposed reconfigurable LUT multiplier can be ranged between the generic multiplier and the reconfigurable constant multipliers using ICAP in terms of the required reconfiguration time and that it provides the best solutions in terms of the required number of slices, all reconfiguration times of the instances from Table B.4 are plotted over the corresponding number of required slices in Figure 3.4.

3.2.5 Relevance of the Results

By the utilization of the pipeline optimized constant coefficient multiplication and the CFGLUT feature the presented run-time reconfigurable multiplier can be beneficially used instead of a generic multiplier as long as enough reconfiguration memory (block

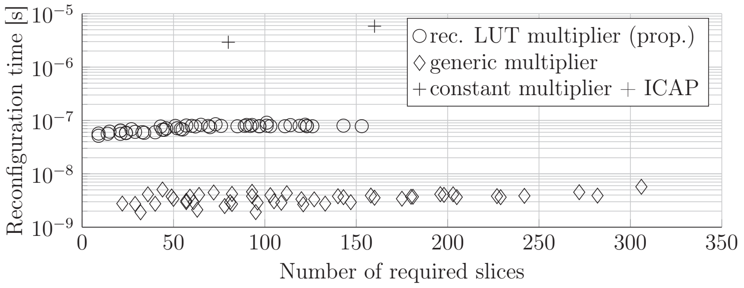


Figure 3.4: Reconfiguration time in seconds over required slices for the different reconfigurable multiplier implementations.

RAM) is available. Only 55 % of the slice resources are required on average and the design is much faster. Moreover, the proposed multiplier is the best choice when short reconfiguration times are required instead of using constant coefficient multiplier IP cores and the PR via the ICAP. There the reconfiguration times are very long with a much higher slice utilization at an operation speed of only 65 %.

3.3 Application of LUT Reconfiguration in Reconfigurable FIR Filters

An important application of the presented constant multiplier is a digital FIR filter whose characteristics can be changed during run-time. Reconfigurable FIR filters using reconfigurable LUTs are further investigated in this section.

3.3.1 Direct Approach

The fundamental operation in an FIR filter with filter length N is the inner product of two vectors

$$y_k = \mathbf{c}\mathbf{x} = \sum_{n=0}^{N-1} c_n \, x_{k-n} \;, \tag{3.11}$$

while $\mathbf{c}$ consists of weighting constants, called coefficients and $\mathbf{x}$ consists of the time-shifted input values of the filter with reference to time step k.

A block diagram of such an FIR filter in direct form can be found in Figure 3.5. A constant multiplication by c_n is shown as triangle. With regard to the last section, the direct approach to implement a run-time reconfiguration FIR filter is to realize each constant multiplier as reconfigurable KCM-based multiplier using CFGLUTs. An implementation of the direct approach using the online configuration update was used in [20].

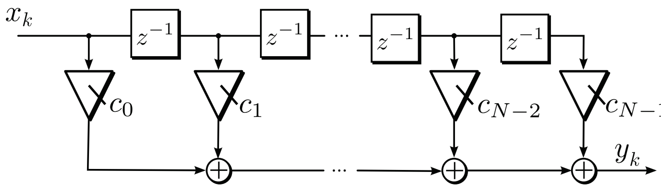


Figure 3.5: Block diagram of an FIR filter in direct form.

3.3.2 Distributed Arithmetic Approach

An alternative to the direct approach is using Distributed Arithmetic (DA). The concept of DA [31,32] was taken and adopted to reconfiguration using CFGLUTs. The resulting reconfigurable FIR filter architecture based on DA was originally published in [69]. Each input x_n in (3.11) can be represented as binary two's complement number. A bit b of this B_x bit number is denoted as $x_{n,b}$. The inner product can be first rewritten and then reordered to

$$\begin{aligned} y &= \sum_{n=0}^{N-1} c_n \left(\sum_{b=0}^{B_x-2} 2^b x_{k-n,b} - 2^{B_x-1} x_{k-n,B_x-1} \right) \\ &= \sum_{b=0}^{B_x-2} 2^b \underbrace{\sum_{n=0}^{N-1} c_n x_{k-n,b}}_{=f(\tilde{x}_b^N)} - 2^{B_x-1} \underbrace{\sum_{n=0}^{N-1} c_n x_{k-n,B_x-1}}_{=f(\tilde{x}_{B_x-1}^N)} \end{aligned} \tag{3.12}$$

where $\tilde{x}_b^N = (x_{0,b}, \dots, x_{N-1,b})^T$ is a bit vector of length N containing the b'th bit of each input x_n. The underlined function

$$f(\tilde{x}_b^N) = \sum_{n=0}^{N-1} c_n x_{k-n,b} \tag{3.13}$$

can only have 2^N values, which can be pre-computed and stored in a LUT with N inputs. The inner product is computed by the sum of shifted LUT outputs according to (3.12). The use of CFGLUTs is only possible with 4/5-input LUTs. Therefore, the N-input LUT has to be divided into several 4/5-input LUTs. This is done by splitting the sum in (3.13) into several smaller sums, like described in [70]. Setting the number of LUT inputs to L, leads to a separation of the sum into $K = \frac{N}{L}$ sub terms, resulting in

$$f(\tilde{x}_b^N) = \sum_{l=0}^{K-1} \underbrace{\sum_{n=lL}^{(l+1)L-1} c_n x_{k-n,b}}_{f_l(\tilde{x}_b^L)} \, . \tag{3.14}$$

It is assumed that $L < N$ and that N is divisible by L. If the latter is not the case the last $f_l(\tilde{x}_b^L)$ term is realized as partial LUT with $L' = N \bmod L$ inputs.

Again, it has to be evaluated whether $L = 4$ or $L = 5$ has to be chosen for the CFGLUT realization. For this estimation it is assumed that N is divisible by L and that the output word size of each function $f(\tilde{x}_b^N)$ in 3.14 is

$$B_f^L = \lceil \log_2(L) \rceil + B_c \; . \tag{3.15}$$

For $L = 4$, a CFGLUT has two outputs, leading to

$$\eta_{\text{CFG}_4} = \frac{N}{L}\frac{B_f^4}{2} = \frac{NB_f^4}{8} \tag{3.16}$$

CFGLUTs. For $L = 5$, B_f^5 CFGLUTs are required for each 5-input LUT, leading to

$$\eta_{\text{CFG}_5} = \frac{N}{L}B_f^5 = \frac{NB_f^5}{5} \tag{3.17}$$

CFGLUTs in total. With $L = 4$ and $L = 5$ in 3.15 follows $B_f^5 = B_f^4 + 1$ and

$$\frac{NB_f^5}{5} = \frac{N(B_f^4 + 1)}{5} > \frac{NB_f^4}{8} \tag{3.18}$$

This means, a LUT input size of $L = 4$ always leads to fewer CFGLUTs for the same function generator. In the case that N is not divisible by L, additional CFGLUTs are required. However, for large N and large B_f^L these terms can be neglected.

For linear phase FIR filters, which are the most common filter design, the weighting constants are symmetric. This symmetry can be exploited to save further CFGLUTs. Symmetry means that

$$c_n = \pm c_{N-n-1} \tag{3.19}$$

for $n = 0 \ldots N-1$. Thus, 3.11 can be rewritten for even N to

$$y = \sum_{n=0}^{N/2-1} c_n \left(x_{k-n} \pm x_{k-(N-n-1)}\right) \; , \tag{3.20}$$

and for odd N to

$$y = \sum_{n=0}^{(N-1)/2-1} c_n \underbrace{\left(x_{k-n} \pm x_{k-(N-n-1)}\right)}_{=z_n} + c_{(N-1)/2} \underbrace{x_{k-((N-1)/2)}}_{=z_{M-1}} \tag{3.21}$$

In terms of computational complexity this means, that only $M = \lceil \frac{N}{2} \rceil$ sum terms are required, which approximately halves the input size of $f(\tilde{x}_b^N)$ (cf. 3.14). At the same time, the number of required CFGLUTs is halved, by the costs of M additional adders.

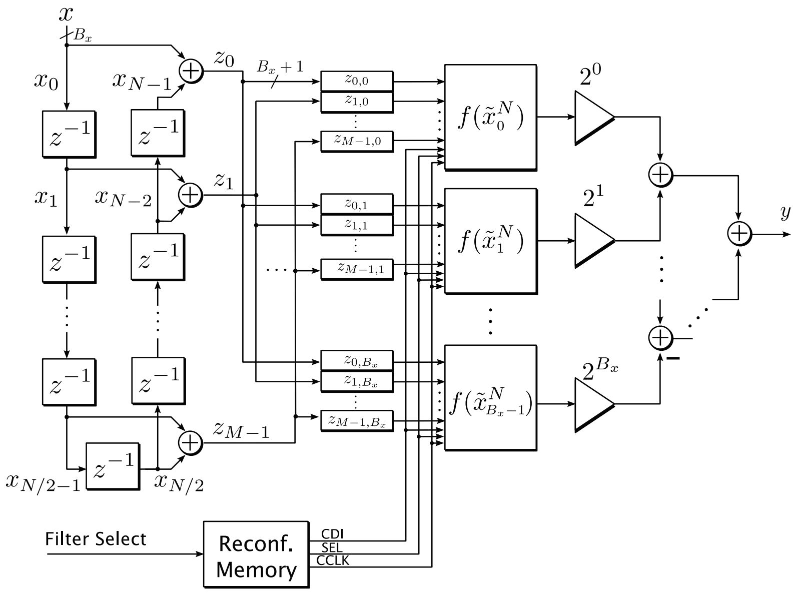


Figure 3.6: Block diagram of an N-tap even symmetric reconfigurable DA FIR filter.

The resulting block diagram for a reconfigurable DA FIR filter is shown in Figure 3.6. For a pipelined realization all adders are followed by registers. Moreover, some shift operations in the output adder tree can be moved towards the output, to reduce the adder's word size. Both optimizations are not shown here for reasons of clarity.

Each function generator $f(\tilde{x}_b^N)$ is realized using CFGLUTs with $L = 4$ and an adder tree as shown in Figure 3.7. This includes that each CFGLUT contains two 4-input LUTs and thus generates two output bits. All outputs are added with the same weight, following equation 3.14. Again, the pipelining of the adder tree using registers is not shown. Reconfiguration is performed using the configuration interface of the CFGLUT. In the example shown here, all configuration interfaces are connected serially, to perform a serial configuration. A parallel reconfiguration is also possible by providing a separate CDI for each CFGLUT.

The shown FIR filter can be loaded with any pre-calculated configuration with up to N symmetric filter taps up to the word size B_c. For smaller N or B_c some bits have to be set to zero. Moreover, all configurations have to be forced to the same symmetry at filter design time. An update with a simple counter as shown for the reconfigurable KCM-based LUT multiplier is not possible for the DA approach.

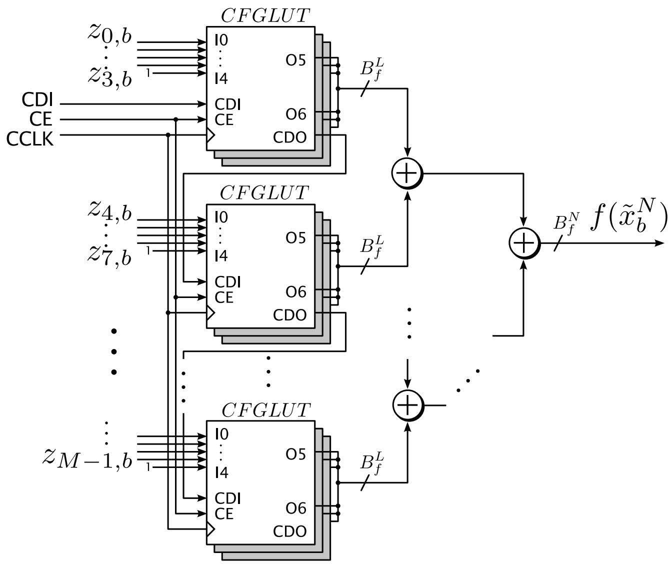


Figure 3.7: Realization of the reconfigurable function $f(\tilde{x}_b^N)$ using CFGLUTs.

3.3.3 Comparison of Both Approaches

Both approaches are first compared using estimation functions for the required FPGA resources and configuration memory. These are then verified by an experimental evaluation. The results can be used as decision support for selecting the right architecture for a given filter. The relevant parameters for resource consumption of a given filter specification are the number of taps N, the filter coefficient word size B_c and the input word size B_x.

Required FPGA Resources

The resource estimation can be done separately for CFGLUTs and adders. Symmetry can be used to save resources for both approaches. For this case, the number of required reconfigurable KCM-based LUT multipliers is $M = \frac{1}{2}N$ and the reconfigurable DA FIR filter consists of $B_x + 1$ M-input LUTs. The reconfigurable KCM-based FIR filter consists of M $B_x + 1$-input LUT multipliers, realized as $\left\lceil \frac{B_x+1}{L} \right\rceil$ L-input LUTs. The output word size of these LUTs is $B_c + L = B_c + 4$ (see equation 3.1). As two 4-input LUTs can be mapped into one CFGLUT the total number of CFGLUTs is

$$\eta_{\text{CFGLUT,KCM}} = M \left\lceil \frac{B_x + 1}{4} \right\rceil \left\lceil \frac{B_c}{2} + 2 \right\rceil . \tag{3.22}$$

In the reconfigurable DA FIR filter the reconfigurable function $f(\tilde{x}_b^N)$ consists of $B_x + 1$ M-input LUTs (see equation 3.13). These are sub-divided into $\left\lceil \frac{M}{L} \right\rceil$ L-input LUTs, each with an output word size of $\lceil \log_2(L) \rceil + B_c$ (see equation 3.15). Again, two 4-input LUTs can be mapped into one CFGLUT and the total number of CFGLUTs is

$$\eta_{\text{CFGLUT,DA}} = (B_x + 1) \left\lceil \frac{M}{4} \right\rceil \left\lceil \frac{B_c}{2} + 1 \right\rceil \ . \tag{3.23}$$

Hence, if M and $B_x + 1$ are both divisible by 4, the number of required CFGLUTs is nearly identical. Therefore, the number of required adders is the important figure for the design choice.

In the reconfigurable KCM-based FIR filter, M adders are necessary for the symmetry exploitation and $M - 1$ structural adders are required to sum up the multiplier results. The M reconfigurable multipliers consist of $\left\lceil \frac{B_x}{L} \right\rceil = \left\lceil \frac{B_x}{4} \right\rceil$ adders each. This results in an adder requirement of

$$\eta_{\text{ADD,KCM}} = M + M - 1 + M \left\lceil \frac{B_x}{4} \right\rceil \tag{3.24}$$

adders. In the reconfigurable DA FIR filter M adders are necessary for the symmetry exploitation and B_x structural adders are required to sum up the DA function generator results (see Figure 3.6). Each function generator contains $\left\lceil \frac{M}{L} \right\rceil = \left\lceil \frac{M}{4} \right\rceil$ adders (see Figure 3.7), which adds up to

$$\eta_{\text{ADD,DA}} = M + B_x + (B_x + 1) \left\lceil \frac{M}{4} \right\rceil \tag{3.25}$$

required adders for the DA FIR filter. To give an estimate for the architecture with the least adders, we assume that M and B_x are both divisible by 4. Then, the KCM-based LUT multiplier architecture needs fewer adders when

$$\begin{aligned} \eta_{\text{ADD,KCM}} &< \eta_{\text{ADD,DA}} \\ M + M - 1 + M \left\lceil \frac{B_x}{4} \right\rceil &< M + B_x + (B_x + 1) \left\lceil \frac{M}{4} \right\rceil \\ \frac{3}{4} M - 1 &< B_x \end{aligned} \tag{3.26}$$

is fulfilled. In summary, the direct KCM-based approach should be preferred to the DA-based approach for filter instances with a short filter length when the input word size B_x is larger than about $\frac{3}{4}M$.

Required Configuration Memory

The configuration memory, which means the required storage in bit for one FIR filter configuration here, can be calculated as the product of required different LUTs and the number of resulting bits for each LUT. For the reconfigurable KCM-based FIR filter only two different LUTs have to be provided. One for the MSB LUT and one for all other LUTs. With a storage requirement of 32 bit per CFGLUT this adds up to

$$\mu_{\text{KCM}} = 32M \left\lceil \frac{B_c}{2} + 2 \right\rceil + 32M \left\lceil \frac{B_c}{2} + 2 \right\rceil = 64M \left\lceil \frac{B_c}{2} \right\rceil \tag{3.27}$$

bit. For the the reconfigurable DA FIR filter all function generators in Figure 3.6 have the same content. With a storage requirement of 32 bit per CFGLUT the storage requirement per configuration is

$$\mu_{\text{DA}} = 32 \left\lceil \frac{M}{4} \right\rceil \left\lceil \frac{B_c}{2} + 1 \right\rceil \tag{3.28}$$

bit. Hence, the required configuration memory for the reconfigurable KCM-based LUT multiplier FIR filter is approximately eight times higher compared to the reconfigurable DA FIR filter.

Experimental Evaluation

The estimations of the last sections provide a rule which architecture has to be chosen for a given filter. However, not all influences of the synthesis process on the aforementioned parameters can be known in advance. Moreover, a possible resource overhead due to pipelining, unbalanced adder trees or input word sizes and filter lengths which are not divisible by 4 has not been considered. Therefore, the reconfigurable KCM-based FIR filter and the reconfigurable DA FIR filter were analyzed in a synthesis experiment using different filter lengths N and input word sizes B_x. A benchmark set of nine filters with filter lengths from $N = 6$ up to $N = 151$ was used. This benchmark set was already used for comparison in previous work on FIR filters [17,69,71,72]. The filter coefficients are available online as MIRZAEI10_N within the *FIRsuite* [73]. An input word size set of $B_x = \{8, 16, 24, 32\}$ bit was used. The code generator described in Section 3.2.3 was extended to support the generation of DA LUTs and the DA FIR architecture (cf. Figure 3.6). This includes the pre-adders for symmetry usage and structural adders of final sum, which were also included in the code generator for the KCM-based LUT multiplier FIR filter.

The resulting filters were synthesized for a Virtex 6 FPGA (xc6vlx75t-ff784-2) using Xilinx ISE v13.4. The same clock signal was used for the filter and the configuration clock (CCLK). The reconfigurable KCM-based LUT multiplier FIR filter is compared to the reconfigurable DA FIR filter concerning configuration memory (μ), number of required slices, maximum clock frequency (f_{max}) and the reconfiguration time T_{rec} calculated as $32/f_{\text{max}}$ (parallel reconfiguration) in Table 3.3. Moreover, the values of the adder cost estimation (cf. equation 3.26) are provided in the first two columns, as well as the slice improvement of the KCM-based LUT multiplier FIR over the DA FIR in % in the last column. The latter provides the most interesting results. Positive

Table 3.3: Synthesis results for the reconfigurable KCM-based LUT multiplier FIR filter compared to the reconfigurable DA FIR filter ($f_{\max}$ = maximum operating speed in MHz, T_{rec} = reconfiguration time in ns, μ = required configuration memory in bit).

			Rec. DA FIR				Rec. KCM FIR				Slice impr.
B_x	$\lceil \frac{3}{4}M \rceil - 1$	N	μ	slices	$f_{\max}$	T_{rec}	μ	slices	$f_{\max}$	T_{rec}	$1 - \frac{\text{LUT}}{\text{DA}}$
8	4	6	320	145	570.5	56.1	2112	93	647.3	49.4	36 %
8	7	10	640	199	492.4	65.0	3520	143	575.0	55.6	28 %
8	9	13	640	191	601.3	53.2	4928	188	608.6	52.6	2 %
8	14	20	960	365	496.3	64.5	7040	297	534.8	59.8	19 %
8	20	28	1280	374	525.2	6 9	9856	406	551.9	58.0	-9 %
8	30	41	1920	627	522.7	61.2	14784	595	573.4	55.8	5 %
8	45	61	2560	835	543.2	58.9	21824	837	499.5	64.1	0 %
8	89	119	4800	1487	499.5	64.1	42240	1668	504.0	63.5	-12 %
8	113	151	6080	1813	395.4	8 9	53504	2156	415.3	77.1	-19 %
16	4	6	320	253	622.7	51.4	2112	179	576.7	55.5	29 %
16	7	10	640	372	552.8	57.9	3520	262	572.4	55.9	30 %
16	9	13	640	353	545.3	58.7	4928	370	522.2	61.3	-5 %
16	14	20	960	635	467.7	68.4	7040	544	518.4	61.7	4 %
16	20	28	1280	707	525.8	6 9	9856	782	540.3	59.2	-11 %
16	30	41	1920	1071	521.9	61.3	14784	1108	487.8	65.6	-3 %
16	45	61	2560	1341	413.7	77.3	21824	1575	463.8	69.0	-17 %
16	89	119	4800	2594	348.3	91.9	42240	3257	480.3	66.6	-26 %
16	113	151	6080	4256	410.5	78.0	53504	4149	428.8	74.6	3 %
24	4	6	320	371	550.7	58.1	2112	303	536.2	59.7	18 %
24	7	10	640	522	556.2	57.5	3520	431	498.3	64.2	17 %
24	9	13	640	535	470.8	68.0	4928	587	545.6	58.7	10 %
24	14	20	960	939	573.7	55.8	7040	884	531.6	6 2	6 %
24	20	28	1280	1029	416.5	76.8	9856	1172	482.9	66.3	-14 %
24	30	41	1920	1558	383.6	83.4	14784	1691	467.1	68.5	-9 %
24	45	61	2560	1958	372.0	86.0	21824	2332	435.0	73.6	-19 %
24	89	119	4800	4047	355.4	9 0	42240	5047	393.1	81.4	-25 %
24	113	151	6080	5802	385.2	83.1	53504	6515	380.7	84.1	-12 %
32	4	6	320	490	524.9	61.0	2112	320	512.3	62.5	35 %
32	7	10	640	734	480.3	66.6	3520	531	485.4	65.9	28 %
32	9	13	640	704	507.1	63.1	4928	771	495.8	64.5	10 %
32	14	20	960	1130	517.9	61.8	7040	1026	473.9	67.5	9 %
32	20	28	1280	1419	430.7	74.3	9856	1482	442.9	72.3	-4 %
32	30	41	1920	1965	376.5	85.0	14784	2091	446.0	71.7	-6 %
32	45	61	2560	2570	358.6	89.2	21824	3601	401.6	79.7	-40 %
32	89	119	4800	4889	302.9	105.6	42240	6523	377.2	84.8	-33 %
32	113	151	6080	7170	391.1	81.8	53504	8016	386.7	82.8	-12 %

values mean that the KCM-based LUT multiplier architecture is the better choice, while negative values mean the opposite. For the smallest and largest filter lengths the estimation provided in the last section is fulfilled, meaning that the KCM-based architecture is the better choice for smaller filter instances when inequation 3.26 is true. There are some exceptions from the estimation, which can have multiple reasons, like already mentioned at the beginning of this section. However, up to 36 % fewer slices are required by using the KCM-based LUT multiplier architecture instead of the DA architecture for small filters, like estimated. For the opposite choice for larger filters, up to 29 % fewer slices are required. Like expected from (3.27) and (3.28) the configuration memory of the reconfigurable KCM-based LUT multiplier FIR filter is eight times higher than for the reconfigurable DA FIR filter. Though, if this larger memory consumption is a problem the online configuration update presented in Section 3.2.2 could be used for the KCM-based LUT multiplier based approach with a small resource overhead. The average reconfiguration times and maximum clock frequencies for both architectures are nearly the same ($\overline{T}_{\text{recDA}} = 69.94\,\text{ns}$, $\overline{T}_{\text{recKCM}} = 65.94\,\text{ns}$, $\overline{f}_{\text{maxDA}} = 472.4\,\text{MHz}$ and $\overline{f}_{\text{maxKCM}} = 494.2\,\text{MHz}$).

3.4 Conclusion

In this section a run-time reconfigurable constant multiplication circuit based on reconfigurable LUTs was presented. It could be shown that the proposed realization can be beneficially used instead of a generic multiplier in terms of required hardware resources and provides a solution with short reconfiguration times compared to using constant coefficient multiplier IP cores and Partial Reconfiguration via the ICAP. An experimental evaluation showed that the CFGLUT reconfiguration with only 32 clock cycles of reconfiguration time provides important trade-off points in the design space for run-time reconfigurable constant multipliers on FPGAs. Moreover, two run-time reconfigurable FIR filter architectures using LUT reconfiguration were introduced and analyzed in this chapter. A direct integration of the presented FIR filter architecture into the adaptive control loop of a particle accelerator could be shown in [20]. For this application no other FIR filter implementation than the reconfigurable KCM-based multiplier presented in this chapter was able to fulfill the strict reconfiguration time and FPGA resource constraints. Finally, decision support for selecting one of the presented architectures for an FIR filter with given filter length and word sizes was provided and supported by an experimental evaluation. A comparison of the presented reconfigurable FIR filter approaches to reconfigurable FIR filters using other reconfiguration methods is part of Chapter 6.

4 Multiplexer-based Reconfigurable Constant Multiplication

This chapter introduces an algorithm to generate multiplexer-based reconfigurable constant multipliers. Constant multiplication is done using additions, subtractions and bit shifts only, like introduced in Section 2.3. The difference here is, that the output constant c can be switched between a limited predefined set of N constants during run-time using multiplexers in the arithmetic data path. As a result the multiplication $c_i x$ is performed, while x is a fixed-point input and i is the index of the selectable output constant c_i. The data path of c_i is called configuration o_i and N the number of configurations in the following. An example of such a run-time reconfigurable constant multiplier (RCM) is shown in Figure 4.1. It shows an RCM with one input x and one output which can be switched between the configurations $12305x$ and $20746x$. Bit shifts are given as edge weight. Resulting scaling factors of the input are given as column vector at the side of each adder. The vector index starting from 0 corresponds to the selected multiplexer input. In the highlighted part, e.g., $3x = x + 2^1 x$ and $5x = x + 2^2 x$ are calculated, depending on the selected multiplexer input. The generation, optimization and evaluation of RCMs for single constant multiplication (SCM), multiple constant multiplication (MCM) and constant matrix multiplication (CMM) is part of this chapter and was originally published in [29, 74, 75]. The source code of the presented algorithm is available online [33].

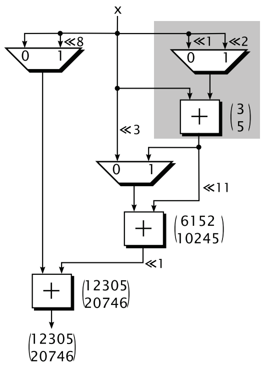
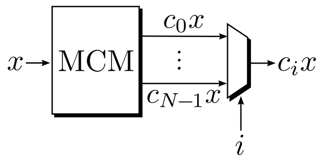
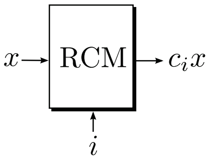


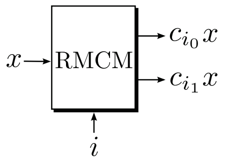
Figure 4.1: Example of an RCM computing $12305x$ or $20746x$.



(a) the straight forward approach using MCM and a multiplexer



(b) multiplexers embedded in the RCM circuit



(c) RMCM block diagram with two outputs

Figure 4.2: Block diagrams for reconfigurable constant multiplication variants.

After a detailed introduction of related work, an optimal method for reconfigurable constant multiplication on FPGAs is presented. This is followed by variations of the solving strategy leading to a heuristic to generate larger RCMs and an experimental evaluation.

4.1 Related Work

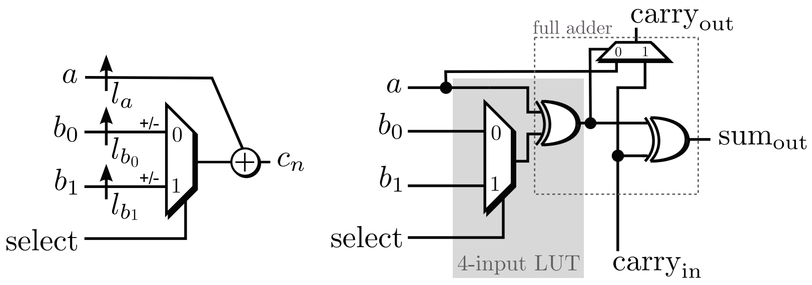
A straight forward approach to generate RCMs is using an MCM circuit followed by a multiplexer to select one of multiple results as shown in Figure 4.2 (a). This realization includes that the internal results of all configurations are calculated, while only one result is required. This is inefficient in terms of hardware resources and power, as some adders could be saved, as well as their power consumption to compute an unnecessary result. Moreover, in pipelined realizations additional registers are required to store possibly unused results. Therefore, it is more efficient to embed the multiplexers directly in the RCM circuit, like indicated in Figure 4.2 (b). That way, hardware resources can be reused and thus saved. This is not limited to single output RCMs and can also be done for multiple outputs like indicated in Figure 4.2 (c), showing a reconfigurable multiple constant multiplier (RMCM) with two outputs. The generation of RCMs with embedded multiplexers has been a vivid research field for the last ten years. As finding run-time reconfigurable solutions for SCM and MCM circuits is a generalization of the basic SCM/MCM problem and thus NP-complete, too, heuristic solutions were presented so far [19, 21, 24–28].

Prior work on multiplier-less RCM is separated into methods targeting FPGAs using basic computation kernels and methods targeting application specific integrated circuits (ASICs) based on the fusion of SCM/MCM solutions. This chapter introduces an optimized method for single and multiple output RCM fusion with a special focus on pipelined realizations. Moreover, in Chapter 5 an optimal shift reassignment method is introduced which is capable to further optimize multiplexer-based RCMs solutions targeting FPGAs and ASICs in terms of required multiplexers. Therefore, the related work on multiplexer-based RCMs provided in the following represents the related work for this chapter and for Chapter 5. It is divided into related work on basic computation kernel methods and fusion methods.

4.1.1 Basic Computation Kernel Algorithms

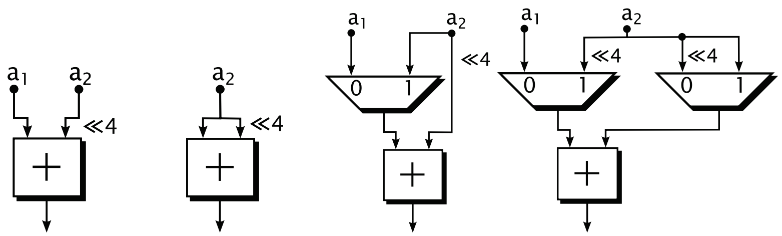
The two methods based on computation kernels presented in the following were proposed concurrently. One by Demirsoy et al. called reconfigurable multiplier block (ReMB) [25] and one by Turner and Woods called limited range multiplier [26]. The basic idea in both algorithms is to define computation kernels which perfectly fit into an FPGA's logic. An example of such a kernel can be found in Figure 4.3. It shows the block diagram of a kernel which can compute $c_n = a2^{l_a} \pm b_0 2^{l_{b0}}$ or $c_n = a2^{l_a} \pm b_1 2^{l_{b1}}$, dependent on the multiplexer's select input. In Figure 4.3 (a), a,b_0 and b_1 are inputs and l_a, l_{b0} and l_{b1} are corresponding left shifts. Moreover, the mapping to a 4-input LUT and carry chain of an FPGA's basic logic element (BLE) is shown in Figure 4.3 (b). a, b_0, b_1 and *select* are 1 bit signals with the appropriate shift already provided through routing.

By cascading the computation kernels larger RCM circuits can be generated. The goal is to find the solution with the least number of required basic computation kernels. In [25] and [26] the search is started from the inputs using exhaustive search over cascaded computation kernels. In a later version of the ReMB algorithm proposed by Demirsoy [19], the search is started from the given target output constants using tables to store, rate and select possible intermediate results, of already fixed computation kernels. Moreover, inserting pipeline registers to shorten the critical delay path is considered. An extended version of the ReMB algorithm utilizing the 6-input LUTs in recent FPGAs, has been presented in [76]. Besides a technology update, the problem of convergence is fixed, which is present for some configurations in the original algorithm [19]. A detailed performance analysis of the algorithm using the algorithm from [76] showed, that the complexity of actually constructing an RCM from computation kernels is far too memory and run-time intensive. Memory and run-time increase exponentially with the number of configurations and exponentially with the output constant word size. Thus, computation kernel-based algorithms are only applicable for single output RCMs with few configurations and small constant word sizes. To give an example, not more than four different configurations for constants with up to 10 bit and not more than seven different configurations for constants with up to 6 bit are practical. This limitation is one motivation for the fusion-based RCM method for FPGAs presented in this chapter. An overview of the related work on fusion-based RCM generation is given in the next section.



(a) block diagram (b) low level mapping

Figure 4.3: Basic computation kernel for a 4-input LUT FPGA.



(a) frag(G_0) (b) frag(G_1) (c) solution 1 (d) solution 2

Figure 4.4: Example fusion of two fragments frag(G_0) and frag(G_1).

4.1.2 Fusion of Constant Multipliers

There are different RCM fusion methods targeting ASICs, all focusing on multiplexer-based reconfiguration. Tummeltshammer et al. propose to fuse several optimized SCM graphs G_i by a recursive algorithm called DAG fusion [27]. As DAG fusion is the most important related work for this chapter and for Chapter 5 and as it is a good representative for fusion algorithms, it will be described in detail in the following.

DAG fusion generates non-pipelined multiplier-less RCMs based on optimal SCM solutions taken from [6]. Optimal SCM means in this context, that the number of required adders in the input graphs is minimal. Like described in Section 2.3, the base of this kind of multiplier-less multiplication is their composition of an addition of shifted inputs formalized as $\mathcal{A}$-operation (cf. 2.6). Initially, DAG fusion fuses two of these SCM input adder graphs G_0 and G_1 by inserting a minimum number of multiplexers. This results in an RCM graph G_*. Next, the remaining input adder graphs, if any, are fused consecutively into G_* until all N input graphs are included. Two example fusions for fragments of two adder graphs G_0 and G_1 can be found in Figure 4.4. The two fragments frag(G_0) in Figure 4.4 (a) and frag(G_1) in Figure 4.4 (b) can be fused to solution 1 in Figure 4.4 (c) and solution 2 in Figure 4.4 (d). Clearly, solution 1 is the better choice in this example (only one multiplexer is required) and would be preferred for fusion. The order of consideration for the sequential fusion has an effect on the results [27], hence, to find the best overall solution, all permutations of consideration of input graphs have to be evaluated. However, this is not leading to an optimal solution as the fusion is done sequentially. When SCM graphs generated with [6] are used for the fusion, this includes their odd fundamental property (cf. Section 2.3), which means they contain exclusively odd intermediate constants. For SCM circuits, this is a desired property to reduce the search complexity. The DAG fusion results are thus limited to this property, too. Considering even intermediate constants as well, will be shown to improve the number of required multiplexers in Chapter 5.

Besides DAG fusion there are several other RCM algorithms based on multiplexer insertion. Chen et al. [28] focus on reducing hardware costs for adders and multiplexers by using identical patterns in the canonical signed digit (CSD) representation of the target constants when creating single output RCMs. It is strongly related to the sub-expression sharing methods described in Section 2.3, but multiplexers are inserted to switch between sharable sub-expressions. A fusion approach similar to DAG fusion was used by Faust et al. [24]. In addition to the previous approach, special care was taken on keeping a minimal logic depth. Moreover, the presented algorithm can generate RCMs with more than one switchable output (RMCM), which is not possible with the methods by Tummeltshammer et al. [27] and Chen et al. [28]. Such RMCMs can also be generated using ORPHEUS proposed by Aksoy et al. [21]. Optimized MCM solutions provided by $\mathrm{H}_{\mathrm{cub}}$ [8] are fused with a heuristic by inserting multiplexers for the reconfiguration, like in all other approaches. Alternative realizations of RMCMs are evaluated during the run-time and the overall best solution is finally selected.

Contribution

The presented fusion algorithms for single output RCMs and RMCMs do not provide optimal results as fusion is done sequentially or by using the CSD representation. The method proposed in Section 4.2, is able to provide a minimal solution for the fusion step in terms of required multiplexers for a given MCM solution by considering all configurations at once during the fusion. In addition to that, pipelining or other FPGA specific issues were not considered during optimization, so far. However, introducing pipelining is essential to overcome long routing delays in FPGAs. Moreover, consideration of FPGA specific resources can lead to better mapping results. These aspects are considered in the new fusion algorithm focusing on pipelined RCMs for SCM, MCM, and CMM presented in Section 4.2. No method to generate pipelined reconfigurable CMM for FPGAs could be found in the analyzed related work. Note that the generation of RCMs is a two-step process of first generating SCM, MCM, or CMM solutions and second fusing them. The proposed method is able to provide the best solution only for the fusion step due to a full search over all possible adder permutations in the given SCM, MCM, or CMM solutions. Therefore, this does not guarantee a globally optimal solution for the overall problem.

Finally, the presented approaches mainly focused on the minimization of adder costs within the RCM construction and multiplexer costs for reconfiguration, but no special focus was on the shift value selection. It will be shown in Chapter 5 by a new algorithm for optimal shift reassignment, that a special focus on the shift values themselves leads to a reduction of the number of required multiplexers.

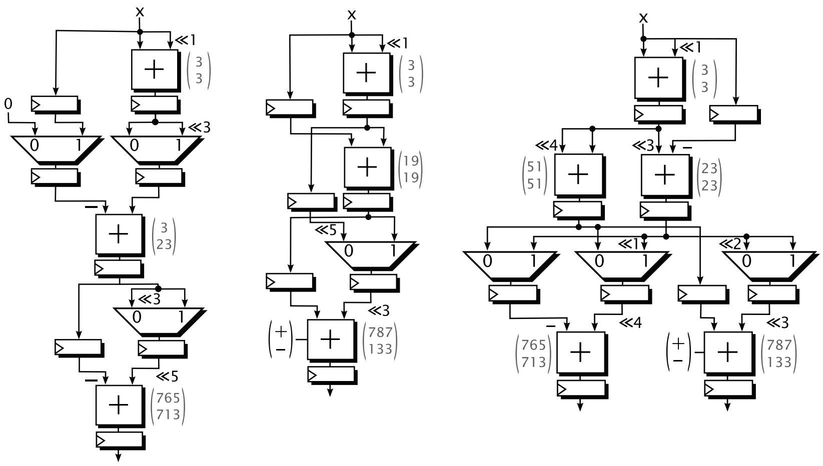
4.2 Reconfigurable Constant Multiplication for FPGAs

This section introduces a new algorithm for pipelined reconfigurable constant multiplication. It is based on the fusion of optimized MCM and CMM solutions generated with the reduced pipelined adder graph (RPAG) algorithm [16]. RPAG generates optimized realizations for SCM, MCM, and CMM targeting FPGAs. Pipelining is considered during optimization and was shown to outperform MCM methods like $\mathrm{H}_{\mathrm{cub}}$ [8] when their results are optimally pipelined [17]. For that reason, the presented fusion algorithm is shown to be beneficial for FPGAs, when compared to subsequently pipelined results of the DAG fusion algorithm [27]. Moreover, the introduced algorithm generates circuits for reconfigurable SCM, MCM, and CMM. In the latter ones, the utilization of already required hardware resources by the sharing of intermediate results is raised, compared to generating multiple single output RCMs. Figure 4.5 illustrates this by showing two pipelined single output RCMs in Figure 4.5 (a)/(b) and the corresponding pipelined RMCM in Figure 4.5 (c). Pipeline registers are inserted after each operation including multiplexers. A column vector in gray at the side of each adder corresponds to the final or intermediate output factor for a specific multiplexer configuration. In addition to the notation used earlier, a switchable adder/subtractor is depicted as adder with a sign vector input. It can be seen that for the pipelined RMCM solution fewer multiplexers, adders and registers are required. Moreover, the maximum pipeline depth (latency) is smaller for the RMCM solution. In the following section the optimal fusion of given RPAG solutions to single output RCMs, RMCMs, and reconfigurable CMM circuits is presented.

4.2.1 Pipelined Adder Graph Fusion

Optimization Entry Point

The input to the pipelined adder graph (PAG) fusion is the required output configuration set $O = (o_0, o_1, o_{N-1})^T$, while N is the number of configurations i and each o_i can be a row vector in the RMCM case. The output configuration set is application dependent, which includes the order of outputs in the RMCM case. In the example in Figure 4.5 (c), $o_0 = (765, 787)$ and $o_1 = (713, 133)$, which means that the circuit multiplies the input x by 765 and 787 for configuration 0 or by 713 and 133 for configuration



(a) RCM solution for 765 or 713

(b) RCM solution for 787 or 133

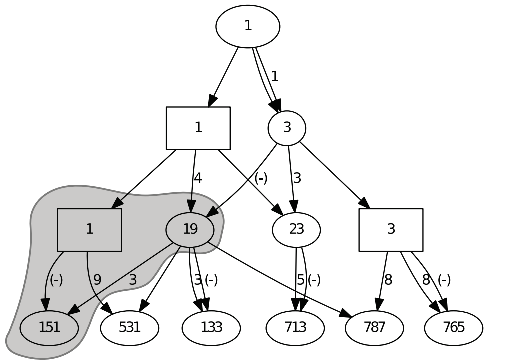
(c) RMCM solution for both outputs

Figure 4.5: Realization of multiple constant sets with two RCMs or one RMCM.

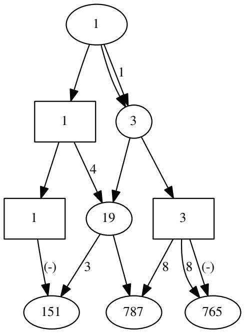
1, while 765 and 713 are assigned to the first output and 787 and 133 are assigned to the second output.

In the first step of the PAG fusion, the RPAG algorithm is used to generate a pipeline optimized MCM solution for the union O^* of all constants in O. This step is followed by an extraction of the resulting MCM adder graph G_* into the corresponding MCM adder graphs G_i for each o_i. Taking the MCM solution of the union of all constants O^* instead of an MCM solution for all o_i separately is important to achieve a large sharing of intermediate results provided by the RPAG algorithm. For the running example used in this section, O is extended to $o_0 = (765, 787, 151)$ and $o_1 = (713, 133, 531)$. Figure 4.6 shows an abstract adder graph representation of the MCM solution of the union of elements $O^* = (765, 787, 151, 713, 133, 531)$ in Figure 4.6 (a). Pipelined adders are shown as circles, and balancing registers as boxes. A number in a circle/box corresponds to the multiple of the input, which is calculated or stored in the particular node. The edges can contain shifts, while positiv values denote a left shift. In addition to that, a sign symbol marks subtraction inputs. For example, $151x$ is calculated as $19x \cdot 2^3 - 1x$ (marked in Figure 4.6 (a)). Another important property is the distance from the input to a specific node, as it determines the number of required pipeline stages in a later implementation. This property is called stage of a node in the following. For example, the adder producing "3" is in stage 1, while the adder producing "23" is in stage 2 and the adder producing "713" is in stage 3.

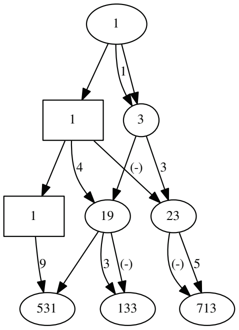
The extracted graphs of $o_0 = (765, 787, 151)$ and $o_1 = (713, 133, 531)$, which are shown in Figure 4.6 (b) and (c), represent the data flow graphs for a specific multiplexer



(a) MCM realization G_* of union of elements in O, $O^* = (765, 787, 151, 713, 133, 531)$



(b) Extracted graph G_0 of configuration o_0



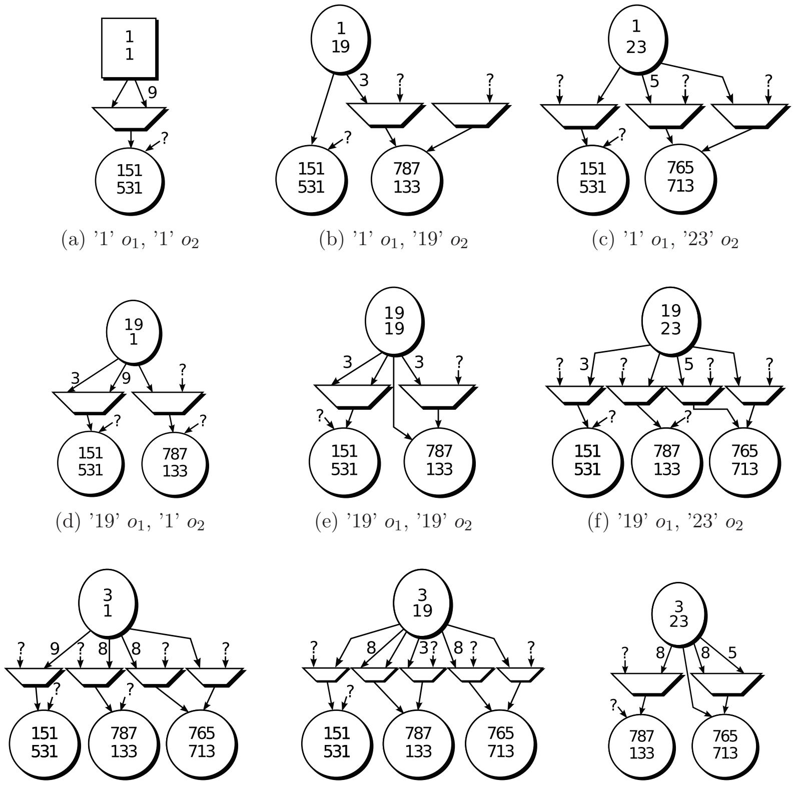
(c) Extracted graph G_1 of configuration o_1

Figure 4.6: Adder graphs to illustrate the optimization entry point of the PAG fusion.

configuration (0 or 1 in this case). These graphs are the input of the fusion algorithm presented in the following.

Fusing Principle

Fusion is done stage-wise beginning with the output stage. The problem is to find the optimal mapping of the intermediate values of different configurations to adders in the respective preceding stage. When this mapping is found, the algorithm proceeds with the next preceding stage until the input is reached. The goal is to find the mapping, leading to the smallest number of necessary multiplexers. This is done by evaluating all combinations of intermediate values and their costs in terms of multiplexers in each stage. All combinations of nodes in stage 2 of o_0 and o_1 of the running example can be found in Figure 4.7. It can be seen, that the number of required multiplexers differs



(a) '1' o_1, '1' o_2 (b) '1' o_1, '19' o_2 (c) '1' o_1, '23' o_2

(d) '19' o_1, '1' o_2 (e) '19' o_1, '19' o_2 (f) '19' o_1, '23' o_2

(g) '3' o_1, '1' o_2 (h) '3' o_1, '19' o_2 (i) '3' o_1, '23' o_2

Figure 4.7: All combinations of nodes in stage 2 for the configurations o_1 and o_2.

between the different combinations. Multiplexers are required at the inputs of the currently investigated stage if

I. the inputs have a different shift value,

II. or the inputs have different sources.

An example of I can be found in Figure 4.7 (a). All multiplexers marked with a "?" as locally unknown inputs in Figure 4.7 are examples of II. The inputs are undefined in this combination step, as six triplets out of the nine partly mutually exclusive combinations can be selected as valid solutions. An example of such a valid triplet in which each intermediate value is only selected once is the selection of (a),(e) and (i). It defines the preceding stage as shown in Figure 4.8. Note that the total sum of multiplexer inputs with no "?" in Figure 4.7 (a),(e) and (i) is equal to the sum of required multiplexer inputs in Figure 4.8. This means, the locally undefined inputs are considered in another

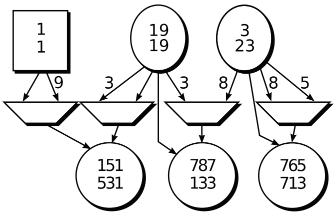


Figure 4.8: Example of a valid selection for stage 2 in the running example.

Table 4.1: Cost matrix for all possible combinations in stage 2.

	1	19	23
1	1	1	1.5
19	1.5	1.5	2
3	2	3	1.5

combination already. This is the main advantage of the fusion principle described here, as the costs for a combination can be evaluated separately, without loss of optimality.

Cost Evaluation

The cost evaluation is following the assumption that multiplexers with more than two inputs will be realized as a cascade of 2:1 multiplexers and $N - 1$ 2:1 multiplexers are required to switch between N configurations. Therefore, each multiplexer input adds costs of

$$\text{cost}_{\text{MUX}} = \frac{N-1}{N} \; , \tag{4.1}$$

while N is the number of configurations. When a register is fused with an adder one multiplexer input becomes 0 (see Figure 4.5 (a) leftmost multiplexer). These inputs can be realized by resetting the succeeding register and are not considered as multiplexer inputs. The costs for the combinations in terms of multiplexer inputs are stored in a multidimensional cost matrix for each stage. An example of such a cost matrix for the combinations in Figure 4.7 can be found in Table 4.1. For example, the second entry in the third row (3) corresponds to combination Figure 4.7 (h) in which six multiplexer inputs, each with a cost_{MUX} of $\frac{1}{2}$, are used.

A valid selection of combinations can be found by first selecting a combination for the first preceding adder. This will directly reduce the possible selection of the corresponding combinations for the second adder and so on. Hence, each valid selection for a preceding stage consists of a set of combinations, each with a unique row and column index. Thus, finding the cheapest selection for a specific stage is equal to minimizing the sum of costs when selecting a valid solution. An example is the selection of (a),(e)

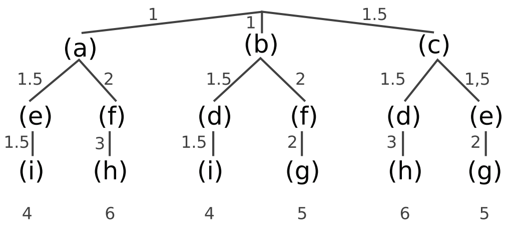


Figure 4.9: Decision tree for the fusion of stage 2 with the combinations of Figure 4.7

and (i) in Figure 4.7 shown in Figure 4.8 with a total $\text{cost}_{\text{MUX}} = 4$. This is equal to the sum of costs $(1 + 1.5 + 1.5)$ of this selection in the cost matrix (main diagonal in Table 4.1). This means, the local selection based on weighting and summing up the required multiplexer inputs is suitable to reduce the total number of required multiplexers. A selection for a specific stage affects the costs in the preceding stages. Therefore, the cheapest selection for a stage is not necessarily the globally best choice. This means, a full search over all possibilities is required to find the minimal solution.

As an abstraction, the search space is illustrated as a decision tree in Figure 4.9. Each decision is a node corresponding to a combination in Figure 4.7 in this example. The edges contain the costs for the specific decision. Each branch is a valid selection for the specific stage with the total costs noted at the end of the branch. The selections (a),(e),(i) and (b),(d),(i) are the cheapest ones and require only four multiplexers for the realization of stage 2.

Full PAG Fusion

The full search space can be constructed by recursively adding the search tree of all preceding stages until the input is reached. An analysis of the search space complexity, which can be found in Appendix A, shows that an exhaustive search can be very time and memory consuming for larger problems. In summary, the number of possible decisions grows factorial with the number of adders K_s in each stage and exponential with the number of configurations N. Nevertheless, the memory consumption for the presented algorithm is moderat, as only the currently optimized branch has to be stored instead of the full search space. Besides that, a branch-and-bound method can be applied, which prunes irrelevant branches.

The simplified pseudo-code of the fusion process including branch-and-bound is shown in Figure 4.10. For each stage s a mapping O for the succeeding stage is provided. In the initial function call this is the desired output mapping defined in O. Moreover, the costs for the current branch have to be provided, which are 0 at the beginning of the fusion. Besides this, a width w can be provided, which is ∞ in the optimal fusion algorithm and will be discussed further in Section 4.2.2. The width is part of a generalization of the algorithm, which enables its usage as heuristic and in an

```
 1 Fuse (O,s,w,cur_costs)
 2 if s > 0
 3   C = evaluate_fusion_costs(O);
 4   C = sort_     ascending(C);
 5   if cur_costs+min(C)>=cur_best_costs
 6     return; -- subtree cut
 7   else
 8     for i = 0...w
 9       if cur_costs+costs(C(i))>=cur_best_costs
10         return; -- normal b&b cut
11       else
12         O = mapping(C(i));
13         cur_costs += costs(C(i));
14         Fuse (O,s-1,w,cur_costs);
15       end if;
16     end for;
17   end if;
18 else
19   if cur_costs < cur_best_costs
20     cur_best_costs = cur_costs;
21     best_solution = cur_solution;
22   end if
23   return;
24 end if;
```

Figure 4.10: Listing of the main recursion of the fusion algorithm. Inputs are the output mapping O of stage $s+1$, current stage s, the search width w and the costs of the current branch *cur_ costs*.

optimal way. In the first step (line 3), all combinations of adders and their multiplexer costs are evaluated and stored in the cost matrix C. The cost matrix C is then sorted (line 4). In this way, the algorithm follows the best cost solutions first, which increases the chance to prune many irrelevant branches of the search space. Equal cost solutions are sorted in the order of their appearance. Branch-and-bound is used to stop searching a branch, whenever the total branch costs exceed the current global minimum costs. There are two branch-and-bound methods implemented within the fusion. The conventional branch-and-bound method (line 9–10) and an advanced branch-and-bound method (line 5–6). The advanced method uses the property that the sum of minimum values of each row in the cost matrix provides a lower bound of costs for the considered stage. The whole subtree can be pruned if this lower bound added to the current costs already exceeds the global minimum costs. Though large portions of the search space can be pruned, the result is still optimal. If no pruning is applied the selection made in one *Fuse* step determines the mapping O for the evaluation of the preceding stage $s-1$ (line 12–14). The fusion of one valid solution is finished when the input (stage 0) is reached. If the just evaluated branch is a better solution, the global best solution

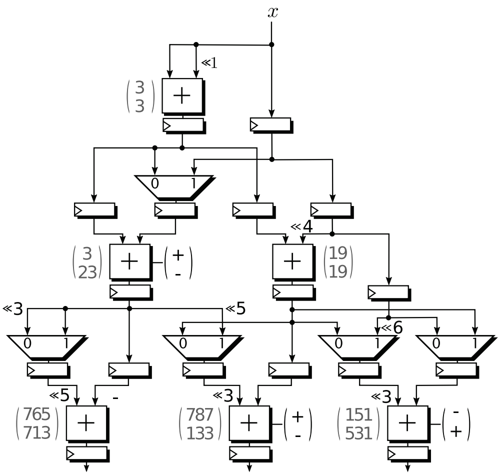


Figure 4.11: RMCM solution for $o_0 = (765, 787, 151)$ and $o_1 = (713, 133, 531)$.

and costs are adjusted correspondingly (line 19–21). It is assumed that the global best solution and its costs are globally known.

As the *Fuse* procedure is called recursively, the optimization stops after returning to the initial function call. In the optimal fusion algorithm ($w = \infty$) this includes an exploration of the full relevant search space. Relevant means in this context, that the pruned search space is excluded, but does not contain better solutions anyway. The resulting optimal solution for the running example can be found in Figure 4.11. Another example including the fusion of three configurations can be found in [29]. The pipelined RCM solution shown in Figure 4.11 already contains the application of a post-optimization to reduce multiplexer and register word sizes in some cases, which is described together with other advanced features of the PAG fusion algorithm in the following.

4.2.2 Advanced Features of the Algorithm

Input Shift Normalization

Some multiplexers switch between different shifts, while all shifts are larger than zero (see Figure 4.8, right multiplexer). This can lead to an unnecessary large word size in the multiplexer and the register after the multiplexer. Normally, the synthesis tools should detect this and reduce the word size accordingly. To ensure the reduction of the unnecessary large multiplexer word size, a post-optimization is performed after fusion which normalizes these shifts. As an example, the normalization of the mentioned shifts can be seen at the leftmost multiplexer in Figure 4.11. A shift of 5 is performed at the

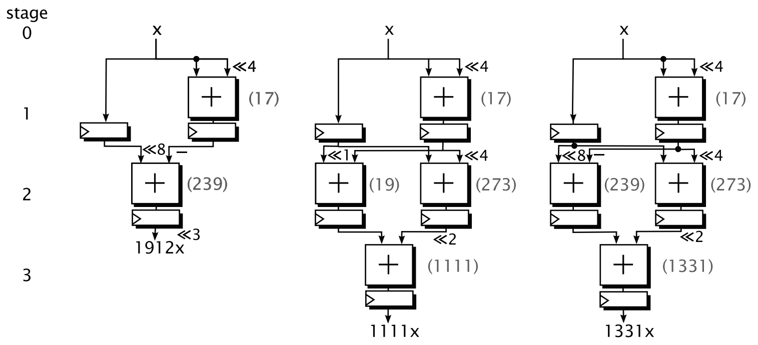


Figure 4.12: RPAG solutions for the constants 1912, 1111, 1331.

output of the multiplexer. The multiplexer's input shifts are reduced accordingly from 8 and 5 to 3 and 0.

Special Treatment for Different Input Shapes

The fused PAGs in the running example have the same shape, which means that they have the same number of nodes in each stage and the same number of stages. This is not always the case when fusing PAGs. An example for different input shapes can be found in the three RPAG solutions shown in Figure 4.12. The circuit representing the multiplication by 1912 (left PAG in Figure 4.12) consists of only 2 stages and requires only one adder in stage 2. The other PAGs in Figure 4.12 consist of 3 stages and require 2 adders in stage 2. The problem of the different pipeline depths can be solved by inserting a register at the output of the leftmost PAG in Figure 4.12. The problem of the missing adder is solved by inserting virtual nodes for each missing adder. During optimization these nodes have the value "−" (don't care) and produce no multiplexer inputs or adders and thus no costs. The fused circuit can be found in Figure 4.13. It can be switched between the constants 1912, 1111, 1331. Note that the first configuration of the right adder in stage 3 is "−" and that the succeeding register has to be set to 0 whenever configuration o_0 is active to ensure a valid output.

Heuristic for Larger Problems

While the search space pruning and local decisions are an effective way to increase the solvable problem size of the optimal PAG fusion, finding an optimal solution for larger RMCM problems in a feasible run-time can not be guaranteed. For these cases a heuristic which provides close-to-optimal solutions is required. Finding an appropriate strategy to explore only a portion of the present search space without loosing good solutions is difficult. Therefore, the PAG fusion search space was analyzed, by sorting local solutions and evaluating their relevance for the optimal and close-to-optimal solutions. The analysis showed, that selecting cheap branches in the local decision phase,

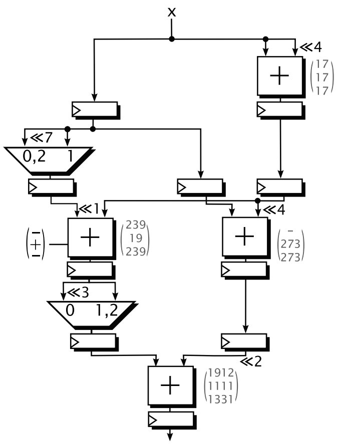


Figure 4.13: Reconfigurable SCM implementation with $O = ((1912), (1111), (1331))^T$.

increases the chance find the optimal or close-to-optimal solutions. That is why in the heuristic version of the proposed PAG fusion, the number of searched braches can be limited to a certain number of locally best solutions in each search tree stage. This is done by the search width w, which determines how many solutions are evaluated in each stage (line 8–16 in the listing in Figure 4.10). This search strategy is related to the *beam search* strategy by Reddy [77]. The presented heuristic is an easy way to control the algorithm's run-time by setting an appropriate w. Moreover, it offers the possibility to increase the solvable problem size for reconfigurable SCM, MCM and CMM.

Exploiting Ternary Adders

Adders with three inputs, also referred to as ternary adders, requiring the same resources as adders with two inputs are supported in recent Intel and Xilinx FPGAs, namely Arria I, II, V, 10, Stratix II-V and Virtex 5-7 [78, 79]. While the Intel tools support the operation by default, its implementation for Xilinx FPGAs was taken from OpenCores [80]. The integration of ternary adders into the RPAG algorithm [81] was shown to significantly reduce the number of required operations and thereby the required hardware. This was the motivation to extend the PAG fusion to adder graphs using ternary adders. The extension of available operations from $a + b$ and $a - b$ in the two-input adder case to $a + b + c$, $a - b + c$, $a + b - c$ or $a - b - c$ ternary adder case, comes with adaptions to the cost evaluation. Moreover, the data structure has to be extended to support nodes with three inputs. An additional problem is, that nodes with two and nodes with three inputs have to be fused. This case can be handled by an addition with 0 at the otherwise unused input, but introduces additional flexibility when selecting the node's input mapping and evaluating a mapping's costs. Another

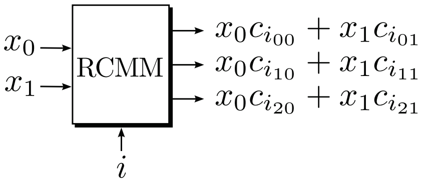

x_0
x_1
RCMM
i
$x_0c_{i_{00}} + x_1c_{i_{01}}$
$x_0c_{i_{10}} + x_1c_{i_{11}}$
$x_0c_{i_{20}} + x_1c_{i_{21}}$

Figure 4.14: Block diagram for a reconfigurable CMM with two inputs and three outputs

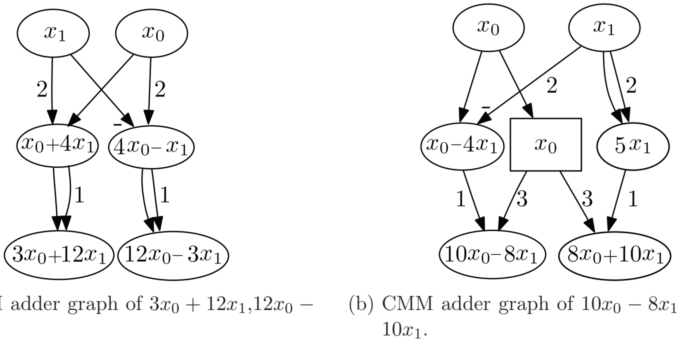
extension is provided within the mapping of negative inputs. In case of a two-input adder, it is always possible to map all negative inputs to the same input, which can be realized by a subtractor or the standard switchable adder/subtractor provided by the FPGA synthesis tools. Mapping all negative inputs to the same input is not always possible for the ternary adder $(a - b - c)$. However, a swapping of inputs to reduce the number of different input signs was adapted for the ternary adder support. While these adaptions are supposed to increase the complexity of the local fusion process, the overall run-time should be reduced. This can be explained by the reduction of the number of adders K_s per stage, due to an operation reduction, like shown by Kumm et al. [81].

Reconfigurable Constant Matrix Multiplication

Besides the ternary adders, the fusion of several CMM circuits generated with the RPAG algorithm is supported by the proposed algorithm. A block diagram of this operation can be found in Figure 4.14. Multiple inputs are multiplied by a reconfigurable constant matrix

$$\mathbf{C_i} = \begin{pmatrix} c_{i_{1,1}} & \cdots & c_{i_{1,M}} \\ \vdots & \ddots & \vdots \\ c_{i_{N,1}} & \cdots & c_{i_{N,M}} \end{pmatrix} . \tag{4.2}$$

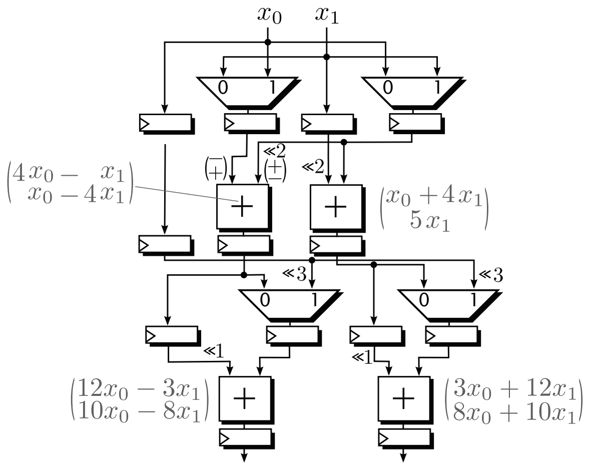
For the fusion of CMM adder graphs the data structure of the fusion was extended to support vectors as adder inputs to run RPAG properly and to be able to validate the resulting reconfigurable CMM circuits. This is a straightforward adaption of the representation of adders in RPAG for CMM. Two example adder graphs G_0 and G_1 generated with RPAG are shown in Figure 4.15. In contrast to the adder graphs shown before, the multiple of a the specific input is shown within the nodes. The input nodes can be seen as vector with only one element unequal to zero $(x_0 + 0x_1)$, $(0x_0 + x_1)$. This results from the fact that adder graph inputs with the same vector have to be mapped to the same single input in the hardware realization of the fused circuit.



(a) CMM adder graph of $3x_0 + 12x_1$,$12x_0 - 3x_1$

(b) CMM adder graph of $10x_0 - 8x_1$, $8x_0 + 10x_1$.

Figure 4.15: Two example CMM adder graphs G_0 and G_1 generated with RPAG.



x_0
x_1
0
1
0
1
≪2
≪2
$\binom{4x_0 - x_1}{x_0 - 4x_1}$
$\binom{x_0 + 4x_1}{5x_1}$
≪3
≪3
≪1
≪1
$\binom{12x_0 - 3x_1}{10x_0 - 8x_1}$
$\binom{3x_0 + 12x_1}{8x_0 + 10x_1}$

Figure 4.16: Reconfigurable CMM adder graph of $o_0 = (3x_0 + 12x_1, 12x_0 - 3x_1)$ and $o_1 = (10x_0 - 8x_1, 8x_0 + 10x_1)$.

Another special case is, that negative intermediate results can appear. Considering this was not necessary for SCM and MCM as a negative constant is assumed to be negated by changing adders to subtractors and vice versa in the succeeding circuit. For reconfigurable CMM this is not possible due to adding intermediate results of different inputs. However, for the fusion of given adder graphs, like presented in the last section the actual value within the node is irrelevant. The multiplexers introduced by the fusion are only dependent on the mapping of nodes and the resulting data flow. Therefore, the fusion itself can be adopted without major changes for reconfigurable CMM. The resulting reconfigurable CMM adder graph can be found in Figure 4.16. Again, the vector index of intermediate results corresponds to the selected configuration. Caused by negative intermediate results like mentioned before, the left adder in the second stage is a switchable adder/subtractors in which the input that is subtracted can be

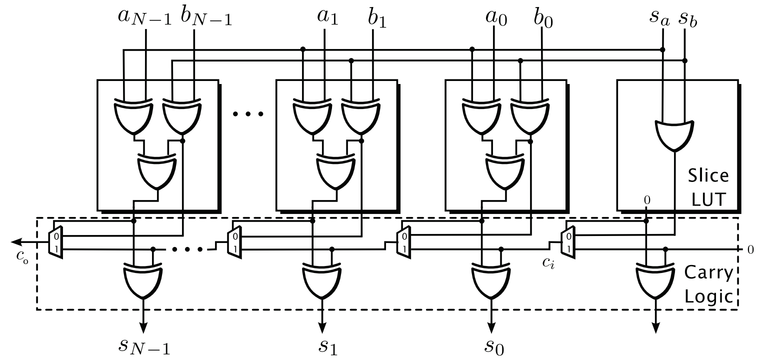


Figure 4.17: Realization of switchable adder/subtractor on Xilinx Virtex 5-7 slices.

either the first or the second input. This kind of adder can be implemented with low additional hardware effort (1 LUT per adder) compared to a conventional adder within Xilinx Virtex 5-7 slices as shown in Figure 4.17. The additional LUT is required to provide the correct carry input. The following LUTs provide an XOR of the inverted or non-inverted inputs, which is added up to a full adder using the slice's carry logic. The inputs a and b are inverted when required by an additional XOR at each input with a corresponding subtraction input s_a or s_b set to 1. The subtraction of both a and b is not supported by the given implementation. Using this switchable adder/subtractor reconfigurable CMM can be implemented very efficiently on Xilinx FPGAs.

Moreover, for non-pipelined realizations the reconfigurable CMM proposed here seems to provide good results. This can be shown by the example constants in Figure 4.16, which were taken from [23]. A hand-optimized non-pipelined realization shown there ([23], Fig. 5) needed the same number of adders, the same logic depth, but two 2:1 multiplexers more. Two further hand-optimized non-pipelined realization examples ([23], Fig. 11 (a) and (b)) could be improved (same number of adders, same logic depth, but one (a) and two (b) fewer 2:1 multiplexers) by the proposed automatic fusion of CMMs realizations presented here.

4.2.3 Implementation

The PAG fusion was implemented in C++ as a command-line tool. Its code is available online as open source within the PAGSuite project [33] and can be compiled using standard C++ compilers (gcc [82] and clang [83] were already tested on Linux, MacOS and Windows, respectively). C++ was chosen to implement and evaluate the PAG fusion algorithm as it provides a better performance compared to a Matlab implementation, as it is highly portable, supports a multi-platform development and as it provides a very convenient way to automate the evaluation of large benchmarks using a command-line interface and generic bash scripts. The command-line tool can be used to provide the required output mapping O in the PAGSuite standard syntax for PAGs described in Appendix C, the width for the fusion heuristic, a possible time-out after which the best solution found so far is returned, the name of the output file containing the fused PAG and the required level of debugging information. The output file contains a description of the fused graph in the PAGSuite standard syntax. This can be used to generate VHDL code using an implemented code generator based on the FloPoCo library [84], which is available online in the FloPoCo branch of the University of Kassel [85]. Moreover, the Origami high-level synthesis (HLS) flow [34] was extended to transform the fused PAG description of reconfigurable SCM, MCM and CMM graphs into a Matlab/Simulink model. The Origami HLS flow provides a Matlab/Simulink to hardware tool chain developed to ease the process of creating, testing and evaluating HLS algorithms. It is used to develop and optimize algorithms to efficiently transform model-based system descriptions into optimized FPGA implementations. The extension mentioned before makes the optimized PAG fusion available to this model-based system development. An example of such a Matlab/Simulink model of the RCM shown in Figure 4.13 can be found in Appendix D.

4.2.4 Experimental Evaluation

In this section the proposed method is evaluated by synthesis experiments. First, the proposed PAG fusion heuristic is analyzed with regard to the quality of results. This is done by comparing the heuristic against the optimal PAG fusion in terms of required slices and algorithm run-time. Then, the results of this analysis are used for a comparison with a state-of-the-art method for RCM generation (DAG fusion by Tummeltshammer [27]) with and without pipelining for the single output case, to evaluate the benefit of the proposed algorithm. This is done to answer the question, if an optimization which considers already pipelined adder graphs for fusion and all configurations in a single run, can provide better results for FPGAs. Moreover, the results will be used to determine the number of configurations for which multiplexer-based RCMs are valuable on FPGAs. Finally, the generation of pipelined RMCMs is evaluated, which is required, e. g., for run-time reconfigurable FIR filters. The application of the proposed method in such filters is further discussed in Chapter 6.

The same FloPoCo-based VHDL code generator was used for all experiments to create synthesizable VHDL code. The VHDL code was synthesized using identical settings to a Virtex 6 FPGA (xc6vlx75t-2ff484-2) using Xilinx ISE v13.4 and to an Intel Stratix V FPGA (5SGSMD3E3H29C4) using Quartus Prime 15.1.

Evaluation of the Heuristic

The evaluation of the heuristic was performed using an SCM benchmark with 2 up to 14 configurations, which can be found online [33]. For each number of configurations it consists of 100 constant sets, whose random constants are uniformly distributed be-

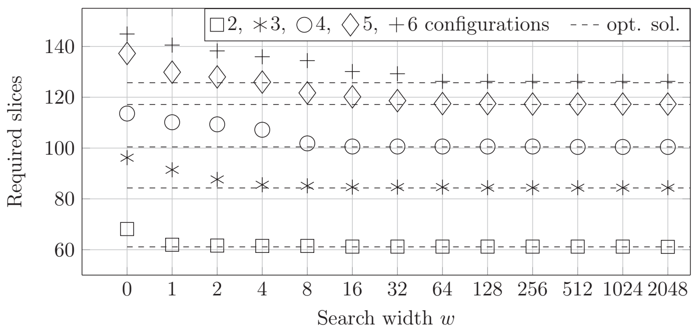


Figure 4.18: Comparison of the average number of required slices of the heuristic with different search widths w and optimal solution (dashed lines).

Table 4.2: Comparison of the average run-time of the heuristic ($w = 64$) to the optimal method and average area overhead of the heuristic solution.

number of configurations	run-time in seconds		speedup by the heuristic	area degradation of the heuristic
	heuristic with $w = 64$	optimal		
2	<0.001	<0.001	1.00 x	0.41 %
3	0.00188	0.00190	1.01 x	0.69 %
4	0.9293	0.9355	1.01 x	0.37 %
5	2.51	120.52	48.0 x	0.46 %
6	14.16	1,842.24	130.10 x	0.96 %

tween 1 and $2^{16} - 1$. Optimal solutions using the proposed PAG fusion were generated as baseline. Moreover, the heuristic was used with different search widths for all benchmark sets. The comparison of the average number of required slices of the optimal PAG fusion and the PAG fusion heuristic for constant sets with 2 to 6 configurations using 2-input adders can be read out of Figure 4.18. Each symbol represents a specific number of configurations, while a dashed line corresponds to the optimal solution the heuristic solution converges to for larger search widths w. Each data point is an average value of 100 constant sets. The main observation is, that close-to-optimal solutions are found with a rather small search width $w = 64$ for the RSCM benchmark sets with up to 6 configurations. The resulting maximum clock frequency is distributed equally between 443 and 469 MHz for all solutions. The average run-time for the heuristic with a width $w = 64$ is compared to the run-time of the optimal algorithm in Table 4.2. Moreover, the speedup by the heuristic and the area degradation of the heuristic are listed.

It can be concluded, that the speedup for the heuristic increases with an increasing problem complexity, given as larger number of configurations. Moreover, it can be observed that the area degradation of the heuristic is smaller than 1 % at the same time. This confirms that selecting cheap branches in the local decision phase is a good strategy to find close-to-optimal or even optimal solutions very quickly. For larger numbers of configurations (> 6) the run-time of the optimal fusion algorithm exceeded a provided time limit of three hours. In the next section it will be shown that these numbers of configurations are not relevant for generating single output RCMs. However, to generate results for RMCM problems with much larger complexity the optimal method could be too time-consuming. The results presented in this section encourage the use of the heuristic for these cases.

Comparison to Other Implementations

This section contains a comparison of the proposed fusion algorithm to the DAG fusion algorithm by Tummeltshammer et al. [27] and a soft-core multiplier IP Core generated with Xilinx CoreGen [49]. DAG fusion relies on the fusion of adder graphs, but performs a sequential fusion and no pipelining is considered during optimization. Single output RCMs for the benchmark described in the previous section were generated using the proposed fusion heuristic. Furthermore, pipelined and non-pipelined RCMs were generated using the DAG fusion source code, which is available within the SPIRAL project [86]. As DAG fusion can not generate pipelined solutions, the algorithm was extended to be able to insert registers after each adder, subtractor, adder/subtractor, multiplexer as well as registers for pipeline balancing. For the proposed fusion algorithm, the heuristic was used with $w = 64$, to get close-to-optimal or even optimal fusion solutions. DAG fusion can be executed in a so called *restricted mode*, which was used when the run-time exceeded 3 hours. This was typically the case for the benchmark sets with more than 9 configurations.

The synthesis results for the required slices and the maximum clock frequency f_{max} after place and route can be found in Figure 4.19. Each data point is an average value of 100 constant sets. The concrete data of the synthesis results is listed in Appendix B, Table B.1 for the sake of the clarity. As baseline for an alternative realization on an FPGA, a 16×16 bit CoreGen soft-core multiplier (LUT-based implementation) with distributed RAM to store the coefficients was used. Its pipeline depth was set to the pipeline depth of the proposed fusion algorithm's solutions (6 pipeline stages).

When comparing the pipelined implementations, the proposed algorithm is clearly the better choice in terms of sliced as it has a lower slice utilization than pipelined DAG fusion in all cases. The reduction compared to DAG fusion, is 9 % on average (2-input adders) and 26 % on average (ternary adders). The 2-input adder circuits provide nearly the same possible maximum clock frequency as the pipelined DAG fusion circuits and the CoreGen reference. This can be explained by a similar critical path, which can be found in a single adder or in the multiplexers with varying size, due to pipelining. The performance degradation of the ternary adders is about 39 % on average and was also reported by Kumm et al. [81]. The non-pipelined DAG fusion results are sometimes better in terms of slices than the proposed pipelined 2-input and

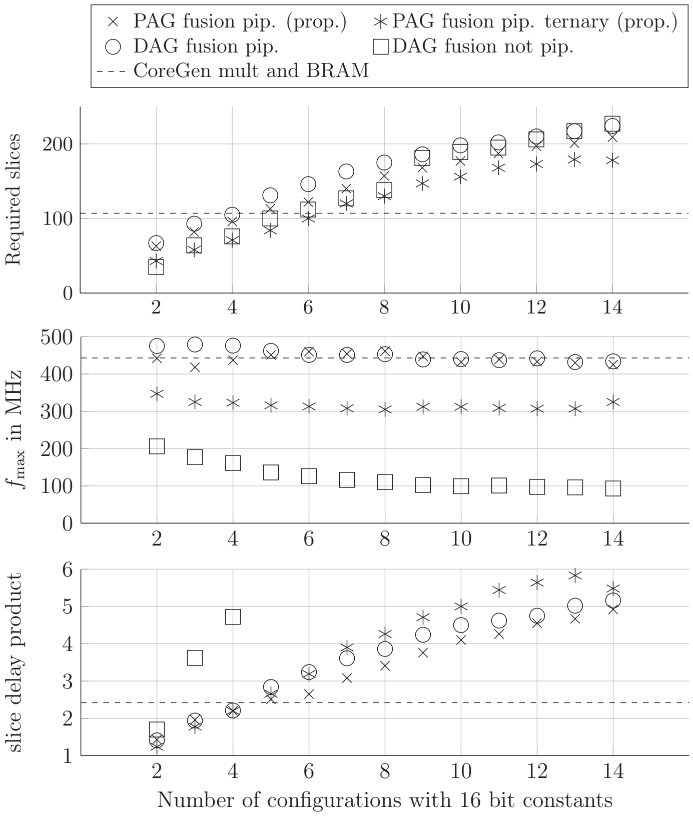


Figure 4.19: Comparison of the required slices (top), the maximum clock frequency (middle), and the slice delay product (bottom) for the proposed method, DAG fusion and a generic multiplier.

ternary adder results. On the other hand, they are up to 5 times slower, which clearly shows the necessity of pipelining on FPGAs. It can be concluded by the comparison to DAG fusion that an optimization which considers pipelining and all configurations in a single run (proposed) leads to better results for FPGAs, than a sequential fusion and post-optimization pipelining (DAG fusion).

In comparison to the generic soft-core multiplier implementation by CoreGen it can be seen that the proposed method is valuable for up to four configurations (2-input adders) and up to six configurations (ternary adders). For more configurations the soft-core multiplier provides the cheapest solutions. For ASICs, DAG fusion was shown to be valuable for single output RCMs with up to 19 configurations compared to a generic multiplier (cf. Table II in [27]). This seems to be the maximum gap between an opti-

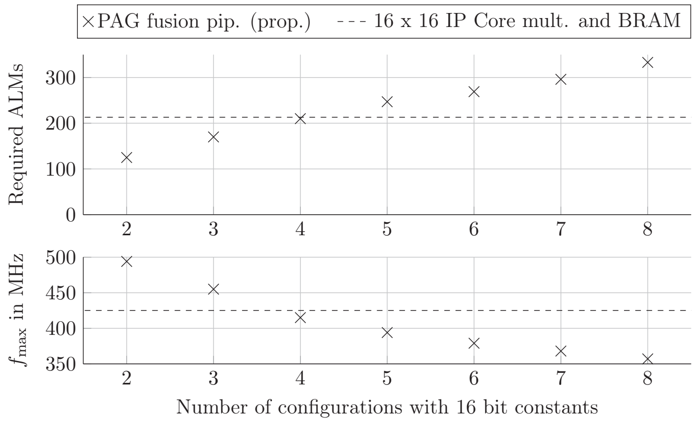


Figure 4.20: Comparison of the required ALMs (top) and the maximum clock frequency (bottom) for the proposed method and a generic multiplier implementation.

mized shift-adder-based RCM implementation and a generic multiplier implementation, which has to be of course smaller for FPGAs.

A similar experimental evaluation for Intel FPGAs also confirms that the results of the proposed PAG fusion are valuable for up to four configurations for 2-input adders. This is shown in Figure 4.20 (cf. Table B.2 in Appendix B). Baseline is a generic multiplier IP core generated with Quartus Prime 15.1. The point of intersection of the proposed solution and a generic multiplier is again at up to four configurations for 2-input adders. Another interesting observation of this experiment is that the generic multiplier IP cores from Xilinx and Intel have a similar hardware requirement (half Virtex 6 slice $\approx$ Stratix V ALM). Moreover, the proposed RCMs have the same relativ FPGA resource consumption, which confirms that the proposed method is applicable for Intel FPGAs, too. Both observations together are the reason for the similar point of intersection for both vendor's FPGAs.

In the introduction of ternary adder support for the proposed fusion it was assumed that the overall algorithm run-time should be reduced for the ternary adder graph fusion. Therefore, the average run-times of the ternary adder graph fusion and 2-input adder graph fusion were analyzed for the whole benchmark. An average algorithm run-time decrease of 5 % could be observed for the ternary adder graph fusion. This confirms the aforementioned assumption for the used benchmarks.

For the shown range of valuable solutions (2–4/2–6 configurations) the proposed algorithm can generate optimal fusion solutions in a moderate time ($<$ half an hour) with significantly lower resource consumption and similar performance. However, the heuristic is unconditionally required to enable multiplexer-based RMCM generation.

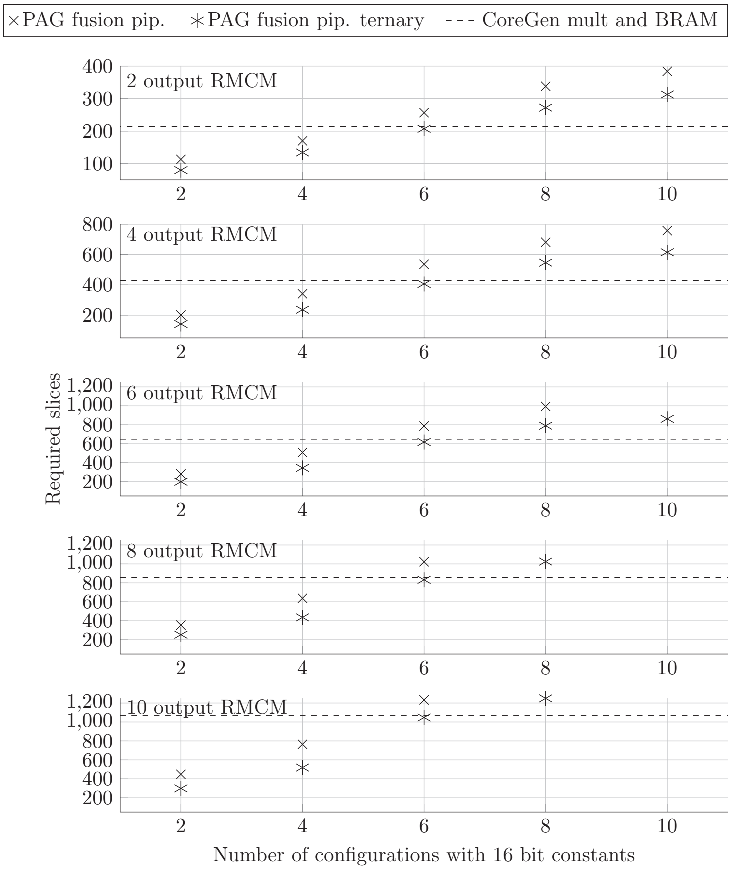


Figure 4.21: Comparison of RMCM using 2-input and ternary adders to a CoreGen soft-core multiplier implementation.

Reconfigurable Multiple Constant Multiplication

In this section RMCMs generated with the proposed fusion heuristic are compared to generic soft-core multiplier IP cores generated with Xilinx CoreGen. Multiple CoreGen multipliers and coefficient RAMs are used to get multiple outputs. Five different MCM scenarios (2, 4, 6, 8 and 10 outputs), each with 2, 4, 6, 8 and 10 configurations were analyzed using a benchmark consisting of 50 constant sets per case. The random constants are uniformly distributed between 1 and $2^{16} - 1$. The search width was again $w = 64$. The synthesis results after place and route of the 2-input and ternary adder implementations compared to generic multipliers can be found in Figure 4.21 (cf. Table B.3 in Appendix B). Each data point is an average of 50 constant sets.

It can be observed, that by the additional reuse of intermediate results between the circuits for the different outputs, the RMCMs generated with the proposed fusion algortihm have a larger range of valuable solutions than in the single output case. The generic CoreGen soft-core multiplier implementation is better for 6 or more configurations (2-input adders) and 8 or more configurations (ternary adders). However, for the application domains of hardware efficient run-time adaptable filters [18, 19, 21] as well as multi-stage filters for decimation or interpolation like polyphase FIR filters [24], 2 to 6 MCM configurations are common. This is the range in which large savings are possible, when preferring the solutions generated by the proposed fusion algorithm. For the best data points, only 25 % of the slice resources are required, meaning that up to 750 slices can be saved compared to using a generic multiplier in the 10 output RMCM case.

4.3 Conclusion

In this section an algorithm to generate pipelined reconfigurable constant multipliers was presented. It is based on the fusion of optimized pipelined adder graphs. Reconfiguration is achieved by switching between different adder graph parts using logic multiplexers. A new optimal algorithm using branch-and-bound was presented to generate single output RCMs and extended by a heuristic for RMCMs and reconfigurable CMM. The heuristic is necessary as the search space is getting too large with increasing numbers of operations in the fused circuits. Using the heuristic with its controllable search width, was shown to be able to find close-to-optimal or even optimal fusion solutions within a feasible run-time. The experimental evaluation showed superiority over previous work originally optimized for ASICs by a slice reduction of 9 % on average for 2-input adders and 26% on average for ternary adders. This points out that considering pipelining and all configurations at once during optimization is a great advantage of the proposed algorithm for FPGAs. Finally, an evaluation of RMCMs showed that up to 75 % of slice resources can be saved, when RCMs generated by the proposed fusion algorithm are used instead of generic multiplier IP cores. An application of pipelined RMCMs in a reconfigurable FIR filter and a comparison to other reconfigurable FIR filter implementations can be found in Chapter 6.

5 Optimizing Shifts in Multiplexer-based Reconfigurable Constant Multipliers

The focus of the last chapter was on the fusion of several non-reconfigurable multiplier-less constant multipliers to run-time reconfigurable constant multipliers (RCMs). During the fusion only the reduction of multiplexers by finding the most favorable mapping of adder graph nodes was considered. This mainly addresses the necessity of a multiplexer due to different input sources (cf. Section 4.2.1). However, the inputs in different configurations can have the same source, but different shift values. This means, additional multiplexers could be saved if the majority of input shifts (in the best case all) for the different configurations were equal. Optimizing the input shifts in that way is the main topic of this chapter. After a consideration of related work, the shift reassignment is motivated by an example. Then, the optimal shift reassignment (OSR) method is presented and finally evaluated by experiments.

5.1 Background and Related Work

The post-optimization presented here can be applied to all previous methods to generate multiplexer-based RCMs [19, 21, 24–29]. An introduction to their construction is provided in Chapter 4. Saving additional multiplexers by an alignment of already equal shifts was considered in [27] and [29]. But there, the shift values were not modified to enforce an alignment, as this is a change in the generation of the constants themselves. The generation of adder graphs realizing a certain constant is done as separate step before the fusion. After this step, the shift values within the adder graph and the topology of the adder graph are fixed. Moreover, typically all constants except the outputs are odd numbers (cf. Section 2.3). Optimization can be done this way, as each even number can be generated by left shifting an odd number. The odd fundamental property is beneficial in the context of multiplier-less constant multiplication without reconfiguration, as it simplifies the optimization by reducing possible intermediate results.

Contribution

However, in this chapter it will be shown by the presented OSR method using Integer Linear Programming (ILP), that a shift reassignment can be used to further reduce the number of fusion multiplexers. This modification of shift values to include odd and even intermediate constants during the adder graph fusion, was not considered in previous work on reconfigurable constant multiplication [19, 21, 24–29] and was originally published in [75].

5.2 Optimal Shift Reassignment in Multiplexer-based RCMs

The reassignment of shifts can be illustrated by a sequence of shift value relocations. Although, the optimal method will be done in a different way, the example in the following helps to understand the main ideas. Two example RCMs for $O = (12305, 20746)^T$ are shown in Figure 5.1. The solution in Figure 5.1 (a) is an original RCM solution based on odd fundamental graphs. An optimized realization using the proposed OSR approach is shown in Figure 5.1 (b). Both circuits calculate the same output, but differ considerably in the number of required multiplexers. A reduction of 50 % of required multiplexers can be seen. Moreover, the optimized realization has a smaller logic depth. There are mainly two reason why exactly this RCM solution was taken as example: The shift distribution in the original solution is ideal to save many multiplexers with only two optimization steps, while the consequences of the shift relocation can be seen clearly. In the following example, the relocation steps to transform the solution in Figure 5.1 (a) to the solution in Figure 5.1 (b) are described.

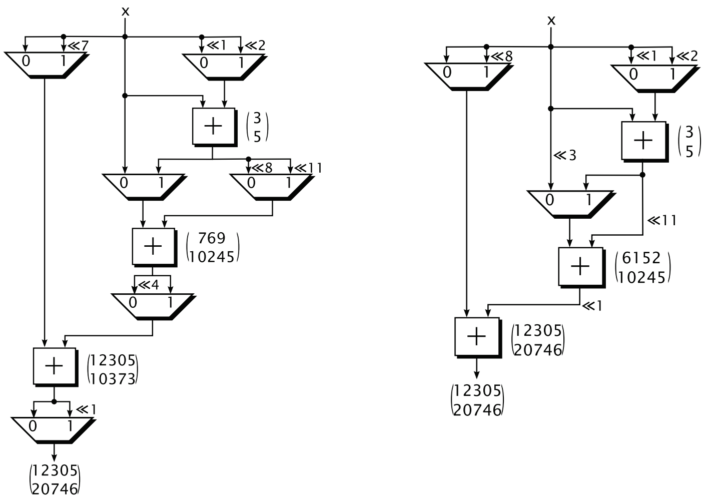
Example Steps for Shift Reassignment

I. The left-shift of 1 at input '1' of the output multiplexer is reduced by 1, which is then shifted to the inputs of the previous adder's multiplexers within configuration o_1.

   Consequences:

   - The output multiplexer disappears (equal input shifts and same input source).
   - The shift at the left input of the last adder increases by 1 from 7 to 8 for o_1.
   - The shift at the right input of the last adder increases by 1 from 0 to 1 for o_1.
   - The intermediate result of the last adder is multiplied by 2^1 for configuration o_1 and changes from 10373 to 20746.

   The RCM after this step is shown in Figure 5.2. All changes are highlighted.



(a) Original (b) Optimized

Figure 5.1: Example of the optimization of an RCM based on odd fundamental graphs by DAG fusion (a) with the proposed OSR approach (b).

II. The left-shift of 4 at the output of the 2nd adder is reduced by 3, which is then shifted to the inputs of the adder's multiplexers within configuration o_0.

Consequences:

- The shift at the left input of the 2nd adder increases by 3 from 0 to 3 for c_0.
- The shift at the right input of the 2nd adder increases by 3 from 8 to 11 for c_0.
- The multiplexer after the 2nd adder disappears (equal shifts).
- The multiplexer at the right input of the 2nd adder disappears (equal shifts).

The resulting optimized circuit is shown in Figure 5.1 (b). In this example the optimization was done stepwise by relocating shift values. The order of this steps has an influence on finding good or the optimal solution. The shift reassignment will be formulated in a different and more general way in the following section. The target is to find the optimal distribution of shifts in the RCM by a complete reassignment using ILP, rather than optimizing the sequence of shift relocations.

5.2.1 Optimal Shift Reassignment Method

Path Properties in Multiplier-less Constant Multiplication Circuits

Changing a shift value within the data path of an addition and bit-shift-based constant multiplier has implications for other shift values in the data path. This results from

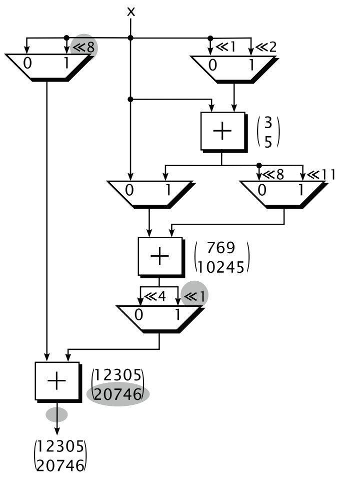


Figure 5.2: Intermediate result after step I of the optimization of the reconfigurable multiplier shown in Figure 5.1 (a).

the property that in adder graphs the sums of shifts on each path from the input to the output determine the resulting output constant. More formally, let S_{ivk} be the shift in configuration i at input k of an adder with index v. In the following $k = 0$ is used for the left and $k = 1$ for the right input of an adder. The output shift is a special case in which k can be ignored and v is equal to the output y. Now, let $\mathcal{P}_{ip}$ be the set of all shifts on path p of a constant multiplication by c_i. The property mentioned above is that the sum

$$\sigma_{ip} = \sum_{S \in \mathcal{P}_{ip}} S_{ivk} \tag{5.1}$$

of these shifts is a constant for each path p for specific constant c_i.

Figure 5.3 shows two realizations of the adder graph G_0^* and G_0^{**} of configuration o_0 realizing constant c_0 in Figure 5.1. There are 4 paths from the input x to the output $y = 12305x$. With the notation given above, the shift of input 1 (right input) of adder 3 S_{031} is 4 in G_0^* and 1 in G_0^{**}. However, the sums of shifts on each path are the constants $\sigma_{00} = 0$, $\sigma_{01} = 4$, $\sigma_{02} = 12$ and $\sigma_{03} = 13$ in both realizations of $c_0 = 12305$. The relation of path sums to the output constant is

$$c_i = \sum_{p=0}^{P_i} \phi_{ip} 2^{\sigma_{ip}}, \tag{5.2}$$

where P_i is the total number of paths for constant c_i. The term $\phi_{ip} \in \{-1, 1\}$ reflects the sign on path p, which is the product of all signs on that path. For the example in

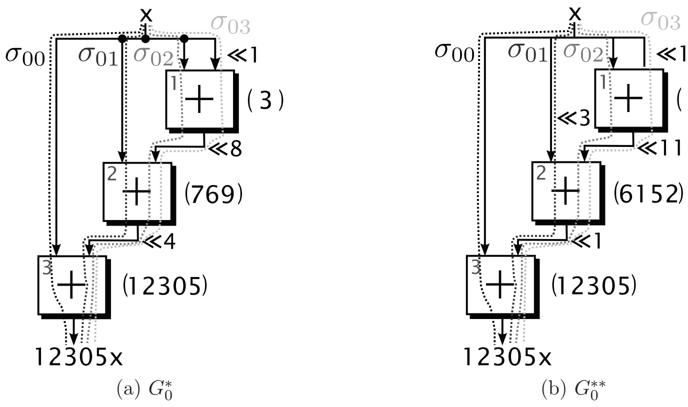


Figure 5.3: Two different adder graph representations of the constant $c_0 = 12305$.

Figure 5.3, $c_0 = 2^0 + 2^4 + 2^{12} + 2^{13} = 12305$. As a consequence, all distributions of shifts having the required path sums for a given adder graph topology are valid solutions for c_i. This property is the reason, why a shift distribution is possible.

ILP Formulation for the Optimal Distribution of Shifts

The objective of the ILP formulation presented in this section is to find the distribution of shifts for a given RCM, leading to a minimum number of multiplexers. A k:1 multiplexer is considered as $k - 1$ 2:1 multiplexers, like motivated before. The shift values S_{ivk} defined in the last section can be directly used as integer variables in the ILP formulation. As the input graphs G_i include the input x and the output y, all shifts on edges from the input to the first adder and all shifts on edges from the last adder to the output are considered, too. In addition to that, binary variables s_{ivkb} are defined, which are 1 when S_{ivk} is equal to a bit shift of b. They are required to link a given shift distribution to the resulting multiplexer costs, which are realized by binary variables M_{uvkb}. These variables are 1 whenever a bit shift of b is set for the edge from adder u to the k's input of adder v. Note that all M_{uvkb} variables are independent from the configuration index i. Hence, if the same bit shift b can be used at a specific input in several graphs G_i, fewer M_{uvkb} variables are 1. This means, the number of required multiplexer inputs is equal to the sum of all M_{uvkb}. Therefore, the objective can be formulated as minimizing this sum. The ILP formulation is summarized in the following.

$$\min \sum_{u\to v\in G_*} \sum_{k=0}^{1} \sum_{b=0}^{B_{\max}} M_{uvkb}$$

subject to

$$\text{C1: } \sum_{S\in\mathcal{P}_{ip}} S_{ivk} = \sigma_{ip} \qquad \text{for all } v \text{ in } G_*,\ p = 0\ldots P_i,\ i = 0\ldots N-1,\ k\in\{0,1\}$$

$$\text{C2: } \sum_{b=0}^{B_{\max}} s_{ivkb} b = S_{ivk} \qquad \text{for all } v \text{ in } G_*,\ i = 0\ldots N-1,\ k\in\{0,1\}$$

$$\text{C3: } \sum_{b=0}^{B_{\max}} s_{ivkb} = 1 \qquad \text{for all } v \text{ in } G_*,\ i = 0\ldots N-1,\ k\in\{0,1\}$$

$$\text{C4: } M_{uvkb} \geq s_{ivkb} \qquad \text{for all edges } u\to v \text{ in } G_*,\ i = 0\ldots N-1,\ k\in\{0,1\},\ b = 0\ldots B_{\max}$$

The objective is defined like motivated before and considers all existing edges and adder inputs in the fused graph G_*, as well as all shift values up to a certain limit $B_{\max}$. The property presented in the previous section, saying that the sum of shifts on each path σ_{ip} is a constant for a specific c_i, is directly used as constraints C1. Instead of taking the non-linear relation in (5.2), which follows from a consideration of non-zeros like introduced in Section 2.3, relation (5.1) can be used. The variables s_{ivkb} are considered in constraints C2 and C3. While constraints C2 relate the binary variables for the shift of b to the integer shift values S_{ivk}, Constraints C3 assure that only one shift binary s_{ivkb} is 1 for all b in the final solution to prevent ambiguities in the definition of the integer shift value in C2. Constraints C4 link the multiplexer costs to different shift values and different sources like explained before.

5.2.2 Experimental Evaluation

For the evaluation of the proposed OSR method the benchmarks used in the experimental evaluation of single output RCMs in Section 4.2 were reused. The OSR was applied to the originally not pipelined solutions generated with the DAG fusion algorithm by Tummeltshammer [27] using the original source code [86] of their algorithm. The following steps have to be done to perform the OSR. First, a RCM solution is mapped to the variables given in the ILP formulation above. Then, the ILP solver is used to find an optimal solution. Finally, the result is applied to the original solution and transformed back into a directed acyclic graph (DAG). Therefore, the proposed OSR was implemented using C++ and a C++ library called Scalp [87] to include the ILP solver Gurobi [88]. The resulting OSR tool is also available online within

Table 5.1: Average number of 2:1 multiplexers (MUX) for DAG fusion before and after the proposed optimal shift reassignment. Each value is the average of 100 test cases.

number of configurations	2	3	4	5	6	7	8	9
DAG fusion [27]	4.12	8.44	11.8	14.77	17.29	20.24	22.51	24.71
OSR (proposed here)	3.66	7.06	10.1	12.87	15.10	17.49	19.63	21.43
fewer MUX [%]	11.17	16.35	14.41	12.86	12.67	13.59	12.79	13.27

number of configurations	10	11	12	13	14	15	16
DAG fusion [27]	27.00	28.12	29.86	31.86	33.59	34.83	36.25
OSR (proposed here)	23.48	24.78	26.32	27.97	29.57	30.34	31.62
fewer MUX [%]	13.04	11.88	11.86	12.21	11.97	12.89	12.77

the PAGSuite project [33]. Input to the OSR tool is the PAGSuite standard syntax for PAGs described in Appendix C. After optimization, the DAG of the optimal solution is transformed back to that syntax to use it in the standard flow provided in the PAGSuite. Furthermore, the number of required multiplexers and an estimated chip area are provided as text output. The OSR tool was used to reassign the shift values optimally in terms of required multiplexers using Gurobi 7.0.1. As the number of required adders is not changed by the OSR, comparing the number of required 2:1 multiplexers is the most feasible technology independent measure. However, a change in shift values could increase the word size of adders and multiplexers. Therefore, a theoretical cost evaluation for these cases is provided. For each adder and multiplexer the total word size is dominated by the largest constant. If the largest constant is increased by the OSR in an adder, the adder's implementation costs are increased, too. For multiplexers only the cases in which the difference between shift values is changed by the OSR lead to increased implementation costs. In these cases, the multiplexer has to switch between the signal input or zero for some additional bits. However, this can be implemented by a simple bitwise AND instead of a multiplexer for those bits [24]. That is why the resulting hardware overhead in these cases is very small.

The resulting improvement of required 2:1 multiplexers (MUX) can be found in Table 5.1. Each value is again the average of 100 test cases with an equal number of configurations. On average 11–16 % fewer 2:1 multiplexers are required with an optimal shift distribution.

The same experiment was repeated with the optimal single output RCM results of the PAG fusion algorithm presented in Section 4.2. Multiplexer savings are possible as the input of the proposed optimal fusion were generated using a heuristic (RPAG [81]). The results can be found in Table 5.2. The savings are considerably smaller with

Table 5.2: Average number of 2:1 multiplexers (MUX) for PAG fusion before and after the proposed multiplexer moving. Each value is the average of 100 test cases.

number of configurations	2	3	4	5	6	7	8	9	10	11	12	13	14
PAG fusion (Sec. 4.2)	2.20	4.79	7.63	10.4	12.64	14.63	17.56	20.22	22.55	25.00	27.6	29.11	30.13
OSR (proposed here)	2.13	4.56	7.37	9.87	11.97	14.12	16.77	19.29	21.44	23.87	26.09	27.35	28.54
fewer MUX [%]	3.18	4.80	3.41	5.10	5.30	3.49	4.50	4.60	4.92	4.52	5.47	6.05	5.28

average values ranging from 3 % to 6 %. However, this can be explained by the fact that the number of required multiplexers in the original results from the PAG fusion presented in this work, is much smaller than in the results from DAG fusion [27] for the same benchmark. This means, the potential to save multiplexers by a reassignment in the PAG fusion results is small, as the original solution is already better than in the DAG fusion case. This observation is another argument why the PAG fusion presented in Section 4.2 should be used to generate single output RCMs, rather than DAG fusion.

To have a closer look at the composition of the shown average values a detailed overview over the results of the first experiment with OSR applied to DAG fusion results is provided in Table 5.3. It lists the number of cases in the benchmark for which a certain number of 2:1 multiplexer inputs can be saved. This is shown for the different numbers of configurations separately. The columns are equal to a histogram of 100 values for a specific number of configurations. The last column is the sum of each row of the benchmark and thus a histogram of the overall benchmark. The savings with the largest occurrence for each number of configurations are highlighted by using bold face.

The cases with a rather small number of configurations have a low complexity and thus a lower optimization flexibility. This can be seen in the data, as for these cases large savings are rare. This means, the original solution was already optimal or had equal multiplexer costs in many cases. For more complex solutions (larger number of configurations), 2 to 5 additional multiplexers can be saved in the majority of cases. At the same time, cases without savings are getting fewer. This supports the assumption that the average numbers in Table 5.1 are consistent and meaningful. For one instance of the benchmark a saving of 11 out of 41 multiplexers (26.83 %) is possible, which can be found in the data for 16 configurations. On the other hand, there was no case with 16 configurations without savings.

Table 5.3: Number of cases in which a certain number of 2:1 multiplexers can be saved compared to the original DAG fusion solution using the proposed optimal shift reassignment.

		number of configurations															
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	sum
saved multiplexers	0	**64**	26	20	13	20	7	5	6	5	6	2	2	1	1	-	178
	1	27	**35**	26	31	12	12	15	11	8	9	10	6	8	2	7	219
	2	8	22	**29**	**29**	**28**	**26**	23	20	17	20	17	18	13	13	7	290
	3	1	11	16	10	23	23	**25**	16	**24**	**22**	**25**	**20**	17	14	15	**262**
	4	-	4	8	15	8	19	18	**24**	19	17	16	16	**23**	22	18	227
	5	-	2	0	1	5	12	6	11	12	10	16	17	13	**23**	**24**	152
	6	-	-	1	1	3	1	6	6	6	14	10	15	18	10	12	103
	7	-	-	-	-	1	-	2	6	7	0	1	2	4	8	10	41
	8	-	-	-	-	-	-	-	-	1	1	3	2	3	3	3	16
	9	-	-	-	-	-	-	-	-	1	1	-	2	-	4	2	10
	10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1
	11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1

Table 5.4: Relative resulting logic depth (adders and 2:1 multiplexers) after OSR, considering multiplexers as tree of 2:1 multiplexers.

Relative resulting depth	$+1$	± 0	-1	-2	-3	-4	-5	-6
number of cases	15	539	315	353	187	71	19	1

The adder depth is not affected by the OSR. However, a reassignment of 2:1 multiplexers can change the overall logic depth. Therefore, the logic depth in terms of the number of chained adders and 2:1 multiplexers of the original and the optimized DAG fusion solutions was analyzed. In 63 % of the analyzed DAG fusion solutions, the OSR reduced the logic depth (-1 to -6 in Table 5.4), while the logic depth was increased in only 1 % of the cases. An increase of the logic depth in these cases was caused by an unfavorable distribution of 2:1 multiplexers at different inputs of the same adder.

All results presented here were generated with a run-time below one second even for the largest cases in single thread mode on a 2.2 GHz Intel Core i7-4770HQ CPU.

5.3 Conclusion

In this section the optimal reassignment of shifts in RCMs using ILP was presented. The proposed method can be applied as post-optimization to all existing RCM algorithm results to save additional multiplexer resources. This includes the FPGA-optimized method presented in Section 4.2 as well as the methods targeting ASICs introduced in Section 4.1. An experimental evaluation showed, that even though the original solutions were partly generated in an optimal way, improvements can be achieved by a redistribution of shifts. This can be explained by the fact that the original solutions were based on odd fundamental input graphs. It could be shown that reassigning shift values within a given single output RCM solution can lead to large absolute and average multiplexer savings up to 16 %. A reassignment of shifts in reconfigurable MCM and CMM is also possible and will be considered in future work. This will help to further reduce the implementation costs for run-time reconfigurable constant multipliers.

6 Design Space of Run-Time Reconfiguration Methods

The reconfiguration methods presented in the last chapters use different ways to perform a run-time reconfiguration. As a consequence, the implementations have different properties which are relevant for their comparison and selection as most suitable method for a specific application. This chapter evaluates this design space for the introduced run-time reconfiguration approaches using reconfigurable FIR filter implementations as example. The evaluation is separated into the consideration of the hardware effort, performance, reconfiguration time and configuration memory and the consideration of the required energy per sample and reconfiguration energy. The evaluation of reconfiguration energy for LUT-based RCM circuits and multiplexer-based RCM circuits was not considered in the analyzed previous work so far. Finally, an overview over the design space of RCM implementations on FPGAs is provided in the conclusion.

6.1 Design Space of a Reconfigurable FIR Filter

Based on constant multiplication as the fundamental operation of an FIR filter (see equation 3.11) there are many different ways to realize reconfigurable FIR filters on FPGAs. Besides the LUT-based FIR filter architectures presented in Chapter 3, reconfigurable FIR filters are realized using PAG fusion RMCMs presented in Chapter 4. The relation of MCM and FIR filters is illustrated in Figure 6.1. It shows a block diagram of an FIR filter in transposed form. All constant multipliers are directly driven by the current input x_k. Therefore, optimized MCM realizations can be used to realize the highlighted part in Figure 6.1. In the context of run-time reconfiguration, RMCM solutions (see Figure 4.21) are used instead. In addition to that, optimized MCM realizations generated with the RPAG algorithm [53] are used and reconfigured using Partial Reconfiguration. Furthermore, generic FIR filter implementations using generic logic multipliers and DSP block-based multipliers are taken into consideration.

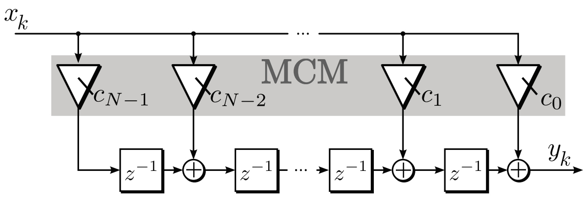


Figure 6.1: Block diagram of an FIR filter in transposed form.

6.1.1 Experimental Setup

The design space exploration was performed using a symmetric mid size FIR filter ($N = 41$, filter: MIRZAEI10_41 [73]) taken from the benchmark set already used in Chapter 3 with an input and coefficient word size of 16 bit. The original benchmark only provides a single filter. Therefore, nine additional filters with the same size and symmetry were designed. Their coefficients can be found in Appendix E.

FIR Filter Implementations

The KCM-based and DA-based FIR filters using CFGLUTs were generated like described in Section 3.3. The configuration update was done with the parallel interface using pre-calculated configurations.

Moreover, the PAG fusion heuristic described in Section 4.2.2 was used to generate the RMCM at the input of the transposed form FIR filter. The heuristic was used with a search width of $w = 64$ for two up to five different filters of the benchmark. The resulting RMCMs were supplemented by structural adders and registers as shown in Figure 6.1.

Again, the Partial Reconfiguration (PR) using ICAP can provide interesting trade-off points. Therefore, the different FIR filter alternatives were generated using the RPAG algorithm [53]. RPAG was chosen as it is a state-of-the-art optimization method for MCM on FPGAs, which provides high performance at low resource consumption. Reconfiguration of the generated filters was assumed to be realized by PR provided by Xilinx (see Section 2.2.1).

Two additional reference designs were generated. A generic VHDL description of an FIR filter in transposed form was used for both designs using generic multipliers. The scaling constants can be changed by switching one of the two multiplier inputs between several constants. The most compact way to provide many constant sets (10 in this case) is to use read-only memory (ROM) realized in the FPGA's distributed RAM blocks. This was assured by using the description recommended by Xilinx [49]. In the first design the input multipliers were created using the Xilinx CORE Generator [66] to enforce their implementation using slices. This is equal to the procedure used in sections 3.2.4 and 4.2.4. The second design was generated by inserting a "$*$" operator in the VHDL description and by enabling DSP block usage in the synthesis tool. The general assumption was so far, that in a large DSP design other parts are competing for DSP resources, like, e.g., in [40], [89] and that, therefore, optimized slice based alternatives should be available. However, in this chapter a DSP-based FIR filter implementation was included as second reference. If the limited quantity of available DSP blocks is not a problem, each of the 16 x16 bit multipliers can be replaced by one DSP block.

The comparison of the DSP block usage to the other methods could be done by relating the two different types of required resources (DSP blocks and slices) according to their relative availability on the considered FPGA by referencing their utilization ratio [53]. Alternatively, a chip area requirement could be estimated [90]. Neither of the described methods addresses the frequent requirement to select the smallest, hence cheapest, FPGA the design fits into. Moreover, all other presented methods are optimized to be realized within the FPGA's slice resources. Therefore, only the required slices are compared here and the number of required DSP blocks is provided to have a complete picture of the required resources.

FPGA and Tool Flow

The implemented FIR filters were synthesized using Xilinx ISE v13.4 for a Virtex 6 FPGA (xc6vlx75t-ff784-2) with identical options. The resource consumption in slices and the maximum clock frequency were taken from the netlist after place and route. The power analysis was done like described Section 2.1 using the Xilinx Power Estimator [56] and the respective estimated maximum clock frequency.

6.1.2 Experimental Results

The results for the different reconfigurable FIR filter implementations are summarized in Table 6.1 and explained in the following. The table lists the resource consumption in slices, the maximum clock frequency ($f_{\max}$) in MHz, the memory requirement for one configuration $\frac{\mu}{\text{set}}$ in bit, the reconfiguration time T_{rec} in nano seconds, the energy required to compute one sample in nano Joule and the reconfiguration energy in nano Joule. The methods introduced in the previous chapters are highlighted by using bold face. Moreover, the slice delay product for the different approaches is shown in Figure 6.2.

Resources, Performance, Reconfiguration Time and Memory

The KCM-based and DA-based FIR filters using CFGLUTs have a nearly equal slice consumption of about 1,100 slices and provide very high maximum clock frequencies $f_{\max}$. Their memory requirement differs by a factor of 8 which is expected from Chapter 3.3.3. However, the required configuration memory is not too critical, as about 70 configurations for the KCM-based CFGLUT implementation could be stored in dis-

Table 6.1: Comparison of different reconfigurable FIR filter implementations with $B_x = B_c = 16\,\text{bit}$ using CFGLUT methods, the proposed PAG fusion heuristic, ICAP reconfiguration, CoreGen multipliers and DSP block multipliers.

Reconf.FIR impl.	slices	f_{max} [MHz]	$\frac{\mu}{\text{set}}$ [bit]	T_{rec} [ns]	$\frac{\text{energy}}{\text{sample}}$ [nJ]	E_{rec} [nJ]
CFGLUT DA	1,071	521.9	1,920	61.3	1.26	58.44
CFGLUT KCM	1,108	487.8	14,784	65.6	1.12	51.64
RMCM (2 conf.)	848	401.3	0	2.5	0.87	0.03
RMCM (3 conf.)	911	372.2	0	2.7	0.98	0.08
RMCM (4 conf.)	968	402.7	0	2.5	0.97	0.08
RMCM (5 conf.)	1,590	340.0	0	2.9	1.30	0.17
RPAG with ICAP	640	386.7	746,496	233,280	0.82	2,799,360
CoreGen multipliers	2,647	343.9	336	2.9	1.94	1.28
DSP multipliers[1]	525	283.13	336	3.5	0.75	0.17

[1] 21 additional DSP blocks are required

tributed RAM and about 380 in block RAM of the analyzed FPGA (xc6vlx75t-ff784-2), which is the smallest Virtex 6 device. The reconfiguration time can be derived from 32 reconfiguration clock cycles for both CFGLUT architectures multiplied by the clock period of the reconfiguration clock $CCLK$ which is equal to the global clock (1.91 ns for CFGLUT DA and 2.05 ns for CFGLUT KCM). Though the required configuration memory is considerably smaller for the DA base approach, similar reconfiguration times of 61.3 ns and 65.6 ns can be achieved. However, this causes a large routing effort for the KCM-based approach. If this routing effort caused by the parallel reconfiguration is a problem, the reconfiguration of the KCM-based FIR filter has to be sequentialized at the expense of an increased reconfiguration time.

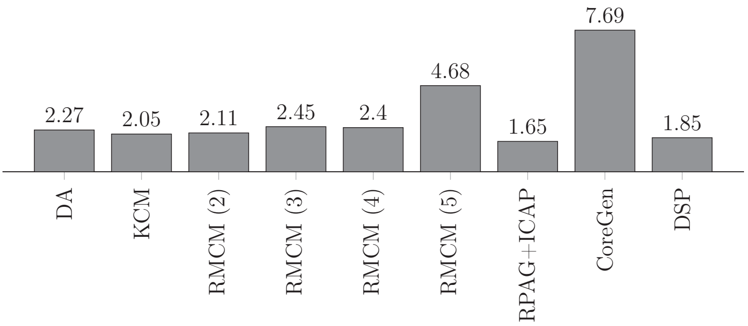


Figure 6.2: Slice delay product of the reconfigurable FIR filters compared in Table 6.1 in slices multiplied by nano seconds.

For the FIR filters using PAG fusion RMCM four solutions are shown in Table 6.1. They differ in the number of implemented configurations the FIR filter can be switched between. For 2–4 configurations the slice consumption is lower than for the CFGLUT architectures. This comes at the cost of a performance degradation. However, the slice delay product for the just discussed cases is similar (see Figure 6.2). As the constant information is contained in the topology of the adder graph, no configuration memory is required. The RMCM-based FIR filter is reconfigured by switching multiplexers in the PAG, which can be done from one clock cycle to the other (2.5–2.7 ns). A large increase in required slices and, at the same time, an additional performance degradation can be seen for the RMCM-based FIR filter with 5 configurations. This can be explained by the fact that 5:1 multiplexers appear in this implementation. As shown in Figure 2.7 their mapping requires one slice LUT more than an up to 4:1 multiplexer on Xilinx Virtex 6 FPGAs. Moreover, an additional slice internal multiplexer is required to link to two required slice LUTs, which could cause the $f_{\max}$ drop. Compared to the CFGLUT-based methods the FIR filter implementation using PAG fusion RMCM can provide a faster reconfiguration time and a lower resource consumption and should be preferred for cases with 2–4 configurations if an $f_{\max}$ of about 400 MHz is suitable.

The number of required slices using RPAG optimized FIR filters ranges from 502 to 569 for the different filter instances. For the reconfiguration using Partial Reconfiguration, a reconfigurable region with at least 569 slices has to be reserved. This means, at least eight frames, each containing 80 slices, have to be reserved (cf. Table 2.1). Assuming that the design is still routable in this limited FPGA region, 640 slices are required. Each Virtex 6 frame contributes with 93,312 bit configuration memory, leading to a memory requirement per filter instance of $\mu_{\text{ICAP}} = 746,496\,\text{bit}$. Assuming that the 32 bit wide ICAP can be run at 100 MHz, the reconfiguration takes $T_{\text{rec}} = \frac{\mu_{\text{ICAP}}}{32} \cdot 10\,\text{ns} = 233\,\mu\text{s}$. The maximum clock frequency $f_{\max}$ of the RPAG optimized FIR filters is between 386.7 MHz and 448.8 MHz. As all RPAG optimized FIR filters have to operate in the same reconfigurable region and thus in the same clock domain, the overall $f_{\max}$ has to be chosen to the $f_{\max}$ of the slowest design which is 386.7 MHz. The reconfigurable FIR filter solution using RPAG with ICAP provides the best slice delay product. However, compared to the CFGLUT-based methods, a factor of up to 388 more reconfiguration memory is necessary and the ICAP reconfiguration is a factor of 3,556 slower. The latter is even worse (factor 80,441 slower reconfiguration) when comparing ICAP reconfiguration to FIR filters using PAG fusion RMCM. The price for the fast reconfiguration times and low memory requirements is paid by a slice overhead of 67 % (DA) and 73 % (KCM) for the CFGLUT-based methods and 33 % (2 conf.), 42 % (3 conf.), 51 % (4 conf.) and 148 % (5 conf.) for the implementation using PAG fusion RMCM.

The FIR filter implementation using CoreGen multipliers is not competitive as it requires at least twice as many resources as all other methods. At the same time

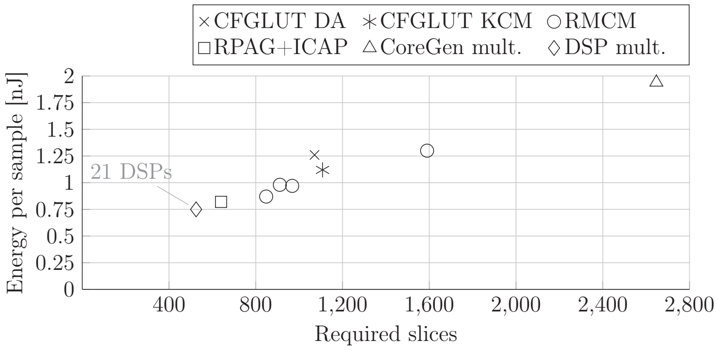


Figure 6.3: Energy per sample over slices for the different run-time reconfigurable FIR filter implementations.

the maximum operation speed is inferior to most of the other approaches. Only the reconfiguration time of one clock cycle could be an advantage for an application with many configurations and tight reconfiguration time constraints.

The DSP block-based implementation has the lowest slice requirement, but 21 additional DSP blocks are required. Moreover, this approach performs worse than RPAG with ICAP in the slice delay product, even without considering the DSP blocks. This results from a low maximum clock frequency caused by the routing delay from the constant set ROM realized as distributed RAM to the multipliers in the DSP blocks. Therefore, this implementation approach is not particularly worthwhile when high performance is required.

To sum it up, the resulting reconfigurable FIR filters of the PAG fusion RMCM-based implementation provide the fastest reconfiguration time with a better resource consumption for 2 to 4 configurations. For 5 to 10 configurations it depends on the required reconfiguration time, if the reconfigurable FIR filter using DA or KCM multipliers together with CFGLUTs or the implementation using RPAG with ICAP should be used.

6.1.3 Energy per Sample and Reconfiguration Energy

The evaluation of the different run-time reconfigurable FIR filter implementations in terms of the required energy to compute one sample and the energy required to switch the coefficient set is provided in this section based on the data already shown in Table 6.1. The required energy per sample is illustrated in Figure 6.3. It shows the relation of the number of required slices to the energy per sample in nJ. The main observation is that the energy per sample grows nearly linear with the number of required slices. This supports the frequently used optimization of a circuit's slice requirement.

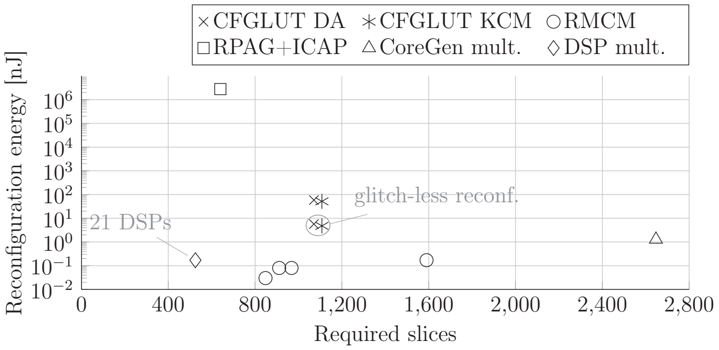


Figure 6.4: Reconfiguration energy over slices for the different run-time reconfigurable FIR filter implementations.

For a final evaluation of the reconfiguration methods in terms of energy, Figure 6.4 provides the reconfiguration energy required to switch between different filter coefficient sets. In addition to the data in Table 6.1, an additional reconfiguration variant called glitch-less reconfiguration was considered for the CFGLUT-based methods. In this variant glitches were prevented by disabling all registers within the reconfigurable filters during reconfiguration. This can be done since the filter results during reconfiguration are invalid anyway. In doing so, the reconfiguration energy is reduced by a factor of 10. Nevertheless, the glitch-less variant requires a 30 times larger reconfiguration energy than the fast reconfiguration methods (multiplexer switching and DSP multipliers). This is caused by a larger reconfiguration time and high switching activity in the CFGLUTs during reconfiguration. In contrast to this high switching activity, only a multiplexer is switched or the input of a multiplier is changed in the cases with fast reconfiguration.

Again, the CoreGen multiplier-based FIR filter implementation does not provide a favorable solution, as it requires the largest amount of energy per sample and a much larger reconfiguration energy (factor of 16) when compared to the RMCM and DSP-based solutions.

The reconfiguration energy of the ICAP-based approach was estimated from an evaluation by Bonamy et al. [54]. They reported a reconfiguration energy of 30 $\frac{\mu J}{\text{kByte}}$ for the ICAP reconfiguration of a Virtex 6 FPGA in their power consideration of reconfiguration controllers. With the memory requirement of $746,496$ bit per coefficient set a reconfiguration energy of $2,799,360$ nJ can be estimated. This is several orders of magnitude larger than for the approaches with fast reconfiguration (PAG fusion RMCM and DSP-based multiplication).

In summary, PAG fusion RMCM should be preferred for 2 to 5 configurations when short reconfiguration times and a frequent switching of configurations is required. The glitch-less CFGLUT-based approach should be taken for 6-10 configurations for an intermediate switching of configurations. For applications with infrequent reconfiguration the implementation using RPAG solutions and ICAP could be a good choice. If no limitation in available DSP blocks is present, using DSP-based multipliers provides an alternative.

6.2 Conclusion

This chapter provided a design space evaluation for the application of reconfigurable constant multiplication to reconfigurable FIR filter implementations. The main focus was on the three methods considered in the previous chapters, namely the LUT-based reconfiguration, multiplexer-based reconfiguration and Partial Reconfiguration. While the energy per sample was in the same range for the most important implementation approaches, there were some considerable differences in resource consumption, performance, memory requirement, reconfiguration time and reconfiguration energy. As nearly all non-reconfigurable parts are equal in the different FIR filter implementations (pre-adders for symmetry usage, structural adders of final sum) some general properties of the three different reconfigurable constant multiplication methods can be derived from the FIR filter evaluation. These properties are summarized in the following.

Logic Reconfiguration using Run-Time Reconfigurable LUTs

The KCM-based RCM implementation using CFGLUTs provides a low resource consumption and high performance. The reconfiguration time is relatively short when a parallel reconfiguration can be performed. Otherwise the reconfiguration time scales linear with the number of serialized CFGLUTs. The reconfiguration energy is moderate provided that glitches are prevented during reconfiguration. The number of different constants which can be loaded is only limited by the available memory for the pre-calculated configurations or the constant memory for the online configuration update.

Routing Reconfiguration using Multiplexers

The RCMs generated with PAG fusion are slice-efficient solutions with high performance for a limited number of configurations. This limitation comes due to increasing multiplexer size and increasing optimization effort with an increasing number of fused input adder graphs. An additional slice efficiency can be provided when multiple output RCMs are required (MCM, CMM) due to a reuse of intermediate results. Moreover, an optimal shift reassignment (OSR) can be applied to further reduce the number of required multiplexers. The multiplexer-based reconfiguration provides a reconfiguration within one clock cycle and thereby a low reconfiguration energy requirement.

Partial Reconfiguration using ICAP

Partial Reconfiguration of PAG implementations generated with RPAG using ICAP provides the best solutions in terms of slices with a good performance. However, the large reconfiguration time and required reconfiguration energy limits its applicability to applications in which an infrequent reconfiguration is acceptable.

7 Conclusion

Different ways to perform run-time reconfigurable constant multiplication on FPGAs were developed and evaluated in this thesis. They were divided into logic reconfiguration using reconfigurable LUTs and data path reconfiguration using multiplexers.

A new method to generate LUT-based RCMs for a single output multiplication based on KCMs [30] was presented. It provides hardware efficient RCM solutions which can be reconfigured within a few clock cycles. The number of different configurations is only limited by the available memory. Its application to an otherwise not realizable adaptive control application [20] shows that this method makes a considerable contribution to the realization of run-time reconfigurable constant multipliers on FPGAs.

Moreover, an algorithm to generate RCMs using multiplexers was presented. In contrast to previous methods, pipelined realizations were considered during optimization. This is particularly important to realize fast FPGA implementations. In addition to that, solutions for reconfigurable SCM, reconfigurable MCM and reconfigurable CMM can be generated by using either the optimal or the heuristic version of the algorithm. While the resulting solutions are only valuable for a limited number of different configurations, a considerable amount of hardware resources can be saved for these cases. In addition to that, reconfiguration can be performed energy efficient within one clock cycle, which makes the resulting RCMs perfectly suitable for the time-multiplexed realization of linear DSP transforms, for time-multiplexed resource sharing and for multi-stage filters for decimation or interpolation. The open-source C++ implementation of the proposed algorithm and its interface to the HLS flow Origami HLS [34] make further research, the reproducibility of results and the integration of low-level optimized reconfigurable components into high-level system development possible.

The post-optimization for multiplexer-based RCMs described in Chapter 5 is applicable to all existing RCM solutions for FPGAs and ASICs and helps to further reduce the required hardware resources for multiplexer-based RCMs.

The different methods for run-time reconfiguration presented in this thesis add some important trade-off points to the design space of run-time reconfigurable constant multiplication on FPGAs. This is best seen in Table 6.1 on page 86. The rows representing solutions contributed by this thesis (bold face) provide high performance, low resource consumption and short reconfiguration times. This means, this thesis provides an important contribution to the realization of run-time reconfigurable constant multiplication on FPGAs.

8 Future Work

The solutions presented in this thesis lead to further research questions and accessible application fields. Some of the ideas were already presented in the last chapters. Some other ideas for future work are presented in the following.

8.1 Extension of the RPAG Algorithm

The results of Chapter 5 indicate that there is still room for improvement in the two-step process of generating PAGs using the RPAG algorithm and fusing them using the presented PAG fusion algorithm (see Chapter 4). This results from the fact that the shift values were not considered during PAG generation. As the PAG fusion (second step) can be done optimally, it is necessary to improve the PAG generation with regard to using its results for PAG fusion, to improve the overall solution. The number of multiplexers introduced during the PAG fusion can be reduced by an alignment of the fused adders' input shifts. The number of different shifts is not considered in RPAG so far. Therefore, the RPAG algorithm should be extended to consider the shift values during optimization. This can be achieved by keeping multiple best predecessors sets in the predecessor evaluation phase (see step III in Section 2.4.1). In doing so, this makes a rating of predecessor sets based on their reuse frequency and their shift values possible. The target is to select the predecessor set with the highest reuse frequency, which leads to the fewest number of different shifts. As it is not clear which criterion is the most important one, a weighting of both criteria could be included to further analyze this aspect. More formal, let the gain of a predecessor set selection be

$$G_{\text{pred\_set}} = \alpha \, \frac{1}{\eta_{\text{shifts}}} + (1-\alpha) \, \frac{1}{\eta_{\text{pred}}} \, , \tag{8.1}$$

while η_{shifts} is the number of different shifts and η_{pred} the number of different predecessors in the set pred_set. Then, α can be used to weight the two criteria to select the predecessor set with the highest gain. This extension of the RPAG algorithm has the potential to decrease the number of different shifts in the input PAGs of PAG fusion and thereby the number of introduced multiplexers in the final RCM.

8.2 Reuse of Linear Subcircuits in Time-Multiplex

The results of the PAG fusion algorithm presented in Chapter 4 can be transformed into Matlab/Simulink models using the Origami HLS tool [34]. The target of Origami HLS is to develop and optimize algorithms for the efficient transformation of model-based system descriptions into highly optimized FPGA implementations. In this context the time-multiplexed reuse of resources, also referred to as folding [91], is an efficient method to save hardware resources if computation latency and throughput requirements can be fulfilled. If single operation reuse is considered, the presented multipler-less constant multipliers can be directly used to replace a multiplication by several constants, which is otherwise typically realized by a multiplier with a multiplexer at one input. Replacing this multiplier-multiplexer combination is beneficial as the results of chapters 3, 4, and 6 show that the resource usage and performance of the presented low-level optimized implementations outperform standard VHDL implementations. This is another application which profits from the low-level optimizations presented in this thesis. It is therefore planned to additionally enable high-level support for the LUT-based constant multipliers presented in Chapter 3 within Origami HLS in future work.

Besides replacing single operations, it was shown to be beneficial to use combined common subcircuits for folding [92, 93]. Instead of using a single operation, isomorphic subcircuits are used to be shared in a time-multiplexed fashion. At the same time, Origami HLS provides an algorithm to detect linear systems within a Matlab/Simulink model and returns their transfer function as constant matrix. This feature will be used to replace these system by optimized CMM solutions provided by RPAG. Putting these two aspects together will be evaluated in future work. This is, folding of linear subcircuits using the optimized reconfigurable constant multiplication solutions provided by the algorithm presented in Chapter 4. This can be achieved by finding equally large linear subsystems in a first step. Then, these linear systems are considered for folding. Finally, their transfer functions will be merged and used as different configurations in optimized reconfigurable CMM solutions provided by the proposed PAG fusion. This procedure will help to reduce the implementation costs of folded DSP dominated high-level system descriptions.

A Detailed Complexity Consideration of Optimal PAG Fusion

A.1 Determining the Number of Solutions and Decisions

One key measure of PAG fusion (Chapter 4) complexity is the number of possible solutions to combine the nodes in one stage to N-tuples (N constants which are realized in that specific node) while N ist the number of configurations. The number of nodes in the considered stage s of the input adder trees is K_s. The total number of combinations which are possible for K_s N-tuples is

$$(K_s!)^N \ . \tag{A.1}$$

This results from K_s N-tuples with decreasing number of possible nodes to select, which are

$$\underbrace{(K_s; \ldots; K_s)}_{N}(K_s - 1, \ldots, K_s - 1) \ldots (1; \ldots; 1) \ . \tag{A.2}$$

Only the mapping of nodes of different configurations to N-tuples is important and not the order of the N-tuples itself. Therefore, the order of the N-tuples can be ignored. As there are $K_s!$ combinations to order the N-tuples, the total number of solutions L_s for a specific stage s is

$$L_s = \frac{(K_s!)^N}{K_s!} = (K_s!)^{N-1} \ . \tag{A.3}$$

For the whole problem consisting of S stages the number of solutions is multiplied, leading to

$$L = \prod_{s=1}^{S-1} (K_s!)^{N-1} \tag{A.4}$$

possible solutions.

In order to evaluate the run-time of the algorithm the number of decisions D (which is the number of nodes in the decision tree) is an important number. For one stage s of the input adder trees the number of decisions is calculated as

$$D_s = (K_s!)^{N-1} \underbrace{\sum_{j=0}^{K_s-1} \left(\frac{1}{j!}\right)^{N-1}}_{\leq e(\text{Euler Number})} \leq (K_s!)^{N-1} e = L_s e \ . \tag{A.5}$$

Explanation: For each stage the number of decisions D_s grows in the substages and sums up to

$$\begin{aligned} D_s &= K_s^{N-1} + K_s^{N-1}(K_s-1)^{N-1} + K_s^{N-1}(K_s-1)^{N-1}(K_s-2)^{N-1} + \ldots \\ &= K_s^{N-1} + (K_s(K_s-1))^{N-1} + (K_s(K_s-1)(K_s-2))^{N-1} + \ldots \\ &= \left(\frac{K_s!}{(K_s-1)!}\right)^{N-1} + \left(\frac{K_s!}{(K_s-2)!}\right)^{N-1} + \left(\frac{K_s!}{(K_s-3)!}\right)^{N-1} + \ldots \\ &= (K_s!)^{N-1} \sum_{j=0}^{K_s-1} \left(\frac{1}{j!}\right)^{N-1} \ . \end{aligned} \tag{A.6}$$

For the whole decision tree the total number of decisions D adds up to:

$$D = \sum_{i=1}^{S-1} D_i \prod_{j=1}^{i-1} L_j \tag{A.7}$$

A.2 Upper Bound for the Number of Decisions

If it is assumed that there is a K such that

$$K = \max(K_s) \tag{A.8}$$

an upper bound for the number of decisions $D_{\max}$ using (A.3) and (A.6) is

$$\begin{aligned} D_{\max} &= \sum_{i=1}^{S-1} \left[(K!)^{N-1} \sum_{j=0}^{K-1} \left(\frac{1}{j!}\right)^{N-1} \right] \prod_{p=1}^{i-1} (K_p!)^{N-1} \\ &\leq \sum_{i=1}^{S-1} (K!)^{N-1} e (K!)^{(N-1)(i-1)} = e \sum_{i=1}^{S-1} (K!)^{(N-1)i} \\ &= e(K!)^{N-1} \frac{1-(K!)^{(S-1)(N-1)}}{1-(K!)^{N-1}} \ . \end{aligned} \tag{A.9}$$

To give an example, the upper bound of decisions is 133 for two configurations and $6 \cdot 10^{11}$ for eight configurations for 16 bit constants, where a typical maximum number of adders per stage is 4 and a typical number of stages is 3.

B Data of the Experimental Evaluation

Table B.1: Synthesis results in required slices and maximum clock frequency $f_{\max}$ in MHz for the method proposed in Section 4.2 and DAG fusion. These results are shown in Figure 4.19. Device: Virtex 6 FPGA (xc6vlx75t-2ff484-2). Tool: Xilinx ISE v13.4.

	PAG fus. pip.		PAG fus. pip. ternary		DAG fus. pip.		DAG fus. not pip.	
#conf.	slices	$f_{\max}$	slices	$f_{\max}$	slices	$f_{\max}$	slices	$f_{\max}$
2	63	442	43	347	67	475	35	206
3	82	418	58	325	93	479	64	177
4	96	437	71	323	105	476	76	161
5	113	451	84	316	131	462	100	136
6	122	460	100	313	146	451	112	126
7	140	454	120	308	163	451	127	116
8	157	461	130	305	175	453	138	110
9	168	447	147	312	186	439	181	102
10	177	432	156	312	198	440	189	99
11	187	439	168	309	202	437	195	101
12	197	433	173	307	210	442	206	97
13	201	430	179	307	217	432	217	96
14	209	425	178	325	224	434	227	93
CoreGen mult and BRAM: 107 slices, 443 MHz								

Table B.2: Synthesis results in required ALMs and maximum clock frequency $f_{\max}$ in MHz for the method proposed in Section 4.2. These results are shown in Figure 4.20. Device: Intel Stratix V (5SGSMD3E3H29C4). Tool: Quartus Prime 15.1.

	PAG fus. pip.	
#conf.	ALMs	$f_{\max}$
2	125	494
3	170	455
4	210	415
5	247	394
6	269	379
7	296	368
8	333	357
CoreGen mult and BRAM: 213 ALMs, 425 MHz		

Table B.3: Synthesis results in required slices and the maximum clock frequency ($f_{\max}$ in MHz) for the RMCM method proposed in Section 4.2. These results are shown in Figure 4.21 and compared to a generic CoreGen multiplier. Device: Virtex 6 FPGA (xc6vlx75t-2ff484-2). Tool: Xilinx ISE v13.4.

		PAG fusion pip.		PAG fusion pip. ternary	
	#conf.	slices	$f_{\max}$	slices	$f_{\max}$
2 output RMCM	2	113	388	80	313
	4	169	401	134	296
	6	256	416	207	296
	8	338	403	272	293
	10	383	387	313	276
4 output RMCM	2	200	363	142	299
	4	340	389	236	273
	6	535	381	406	277
	8	681	372	546	274
	10	757	368	615	271
6 output RMCM	2	282	359	200	280
	4	509	367	346	252
	6	787	361	619	250
	8	994	369	791	259
	10	–	–	863	266
8 output RMCM	2	354	351	254	268
	4	638	358	436	232
	6	1023	366	832	236
	8	1267	355	1025	246
	10	–	–	–	–
10 output RMCM	2	447	347	300	261
	4	765	381	519	217
	6	1233	351	1047	226
	8	–	–	1249	238
	10	–	–	–	–

Table B.4: Synthesis results (required slices, maximum clock frequency $f_{\max}$ in MHz) for the comparison of the reconfigurable LUT multiplier, a generic multiplier, and a RCM using ICAP in Section 3.2.4 in Figure 3.3. Device: Virtex 6 FPGA (xc6vlx75t-2ff484-2). Tool: Xilinx ISE v13.4.

	rec. LUT multiplier (prop.)		generic multiplier		constant multiplier + ICAP	
$B_x \times B_c$	slices	$f_{\max}$	slices	$f_{\max}$	max. slices raw	min. $f_{\max}$
8x8	9	563	22	361	13	421
12x8	24	553	32	527	26	324
16x8	21	571	40	364	30	339
20x8	43	415	57	320	45	328
24x8	51	405	63	471	48	311
28x8	62	415	82	367	60	301
32x8	65	386	93	217	67	287
8x12	9	622	29	366	18	403
12x12	24	545	49	266	28	332
16x12	34	547	61	338	37	287
20x12	60	404	78	402	52	274
24x12	57	398	95	522	61	326
28x12	85	411	109	349	69	253
32x12	76	405	120	299	80	260
8x16	14	568	36	242	18	416
12x16	29	524	59	268	33	325
16x16	40	533	82	238	42	256
20x16	73	377	105	327	60	277
24x16	69	406	121	373	68	289
28x16	97	410	133	362	76	259
32x16	90	389	147	340	101	243
8x20	15	518	44	197	25	347
12x20	33	530	72	224	36	416
16x20	44	478	96	346	50	335
20x20	92	406	140	263	70	272
24x20	101	401	158	255	77	317
28x20	97	412	175	293	84	278
32x20	114	391	196	241	110	236
8x24	21	497	50	295	26	364
12x24	45	479	81	339	43	261
16x24	52	448	112	231	59	334
20x24	89	405	160	281	81	195
24x24	93	389	180	269	88	318
28x24	111	410	205	273	106	251
32x24	123	410	227	262	110	254
8x28	21	494	57	340	29	296
12x28	46	449	93	263	49	277
16x28	55	472	127	301	61	277
20x28	103	410	181	269	86	188
24x28	101	353	203	242	97	281
28x28	123	406	242	259	107	264
32x28	122	385	282	256	128	250
8x32	27	467	64	248	32	339
12x32	54	464	103	251	58	313
16x32	70	427	143	274	82	215
20x32	119	404	198	248	90	222
24x32	126	411	229	274	101	276
28x32	143	400	272	221	122	239
32x32	153	409	306	175	146	195

C Syntax for Reconfigurable Pipelined Adder Graphs

The syntax to describe reconfigurable pipelined adder graphs (PAGs) is an extension of the syntax used to describe PAGs in RPAG [16]. The extension is necessary as multiplexers with an arbitrary number of inputs appear in reconfigurable PAGs. This is not the case for RPAG PAGs consisting of registers and 2-input adders only which are easy to be distinguished. Hence, for reconfigurable PAGs the operation type is defined by a string. Moreover, brackets are included to clearly indicate each operation's output factor. As these changes are important, the complete syntax, which is now the standard syntax for PAGs within the PAGSuite [33], is explained. A graph G as shown in Figure C.1 consists of at least one registered adder/subtractor node. However, in most of the cases it consists of several registered adder/subtractor nodes, register nodes and multiplexer nodes. This is represented as

$$G = \{node, [node]*\}, \tag{C.1}$$

while the square brackets and the '*' in this equation symbolize that the first node is followed by an arbitrary number of nodes (including 0) separated by a comma. The curly brackets are part of the syntax. Each node in equation C.1 can be an adder, subtractor, register or multiplexer. An adder or subtractor node is given as

$$\{'A', [\mathrm{factor}_o], \mathrm{s}_o, [\mathrm{factor}_{p1}], \mathrm{s}_{p1}, \mathrm{shift}_{p1}, [\mathrm{factor}_{p2}], \mathrm{s}_{p2}, \mathrm{shift}_{p2}\},$$

while *factor* is a node's output factor, s is the stage of a node, and *shift* is the input shift assigned to one of the predecessors $p1$ or $p2$. For a subtraction the factor of the subtracted input is negated. The square brackets and the curly brackets are part of the syntax. The square brackets indicate that *factor* can be a vector in the reconfigurable SCM and MCM case and a matrix in the reconfigurable CMM case. The assignment of a factor to a configuration for reconfigurable SCM and MCM is done using a semicolon:

$$[\mathrm{factor}] = [\mathrm{factor}_{c1}; \mathrm{factor}_{c2}; \mathrm{factor}_{c3}; \ldots].$$

For the reconfigurable CMM case a comma is used to separate factors of different inputs and a semicolon to distinguish configurations:

$$[\mathrm{factor}] = [\mathrm{factor}_{c1i1}, \mathrm{factor}_{c1i2}, \ldots; \mathrm{factor}_{c2i1}, \mathrm{factor}_{c2i1}, \ldots; \mathrm{factor}_{c3i1}, \mathrm{factor}_{c3i1}, \ldots].$$

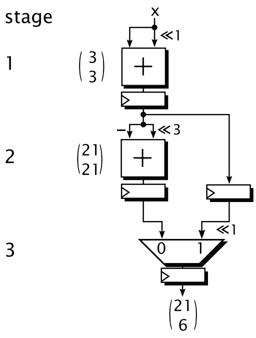


Figure C.1: Example of a reconfigurable PAG G computing $21x$ or $6x$.

A register node has only one input and is given as

$$\{'R', [\mathrm{factor}_o], \mathrm{s}_o, [\mathrm{factor}_{p1}], \mathrm{s}_{p1}\}.$$

A multiplexer node can have as many inputs as configurations. For the sake of clarity the syntax given in the following is for a multiplexer with 3 different input sources. A multiplexer node is represented as

$$\{'M', [\mathrm{factor}_o], \mathrm{s}_o, [\mathrm{factor}_{p1}], \mathrm{s}_{p1}, [\mathrm{shifts}_{p1}], [\mathrm{factor}_{p2}], \mathrm{s}_{p2}, [\mathrm{shifts}_{p2}], [\mathrm{factor}_{p3}], \mathrm{s}_{p3}, [\mathrm{shifts}_{p3}]\},$$

while the predecessors can be shifted in a different way and are given as vector. The vector *shifts* is used similarly to the vector *factor*:

$$[\mathrm{shifts}] = [\mathrm{shift}_{c1}; \mathrm{shift}_{c2}; \mathrm{shift}_{c3}; \ldots].$$

Each multiplexer input can only be driven by one predecessor. Therefore, only the shift values of predecessors are required, for which the predecessor is selected by the multiplexer. For all other configurations the shift value is set to the string "NaN" (not a number).

The example graph G in Figure C.1 consists of all available registered operations types (one adder, one subtractor, one register and one multiplexer). Note that the input is always assumed to be $1x$ for each configuration in stage 0. Using the syntax above, G can be unambiguously represented as

$$\begin{aligned} G = \{&\{'A', [3;3], 1, [1;1], 0, 0, [1;1], 0, 1\},\\ &\{'R', [3;3], 2, [3;3], 1\},\\ &\{'A', [21;21], 2, [-3;-3], 1, 0, [3;3], 1, 3\},\\ &\{'M', [21;6], 3, [21;21], 2, [0;\mathrm{NaN}], [3;3], 2, [\mathrm{NaN};1]\}\} \end{aligned}$$

Further examples can be found by running the PAG fusion code and in the benchmark sets published on the PAGSuite project website [33].

D Reconfigurable Constant Multiplier Integration in Origami HLS

The pipelined adder graph (PAG) fusion algorithm presented in Chapter 4 outputs a file containing the resulting RCM adder graph in the standard syntax for PAGs within the PAGSuite (see Appendix C). This representation can be used within the Origami HLS project [34] to generate a Matlab HDL Coder [94] compatible Matlab/Simulink model of the reconfigurable PAG. As an example the single output RCM shown in Figure 4.13 is realized as Matlab/Simulink model using Origami's LUA scripting language [95] command line with the following input:

```
s = system.read("empty.slx")
s:pagToSystem("{{'R',[1;1;1],1,[1;1;1],0},
{'A',[17;17;17],1,[1;1;1],0,0,[1;1;1],0,4},
{'M',[128;1;128],2,[1;1;1],1,[7;0;7]},
{'R',[17;17;17],2,[17;17;17],1},
{'A',[239;19;239],3,[128;1;128],2,1,[-17;17;-17],2,0},
{'R',[NaN;1;1],2,[1;1;1],1},
{'A',[NaN;273;273],3,[NaN;1;1],2,0,[17;17;17],2,4},
{'R',[NaN;273;273],4,[NaN;273;273],3},
{'M',[1912;19;239],4,[239;19;239],3,[3;0;0]},
{'A',[1912;1111;1331],5,[0;273;273],4,2,[1912;19;239],4,0}}",16)
s:writeSLX("example_RSCM.slx")
```

First, an empty Matlab/Simulink model is initialized by reading a provided empty system. Then, the reconfigurable PAG is added to the model by providing its string in the PAGSuite standard syntax and the required input word size (16 bit at the end of *pagToSystem*). Finally, the Matlab/Simulink model is saved in the Matlab/Simulink standard **.slx* format.

The resulting Matlab/Simulink realization is shown in Figure D.1. The two inputs x and conf. and the output y are drawn as circles. All other block types are annotated in gray using their functionality within the circuit and their Matlab/Simulink names in brackets. Besides the standard elements like adders, multiplexers and registers (z^{-1}), there are several blocks which are used in a specific manner. However, their intended functionality is detected by the synthesis tools. A left shift by l is realized

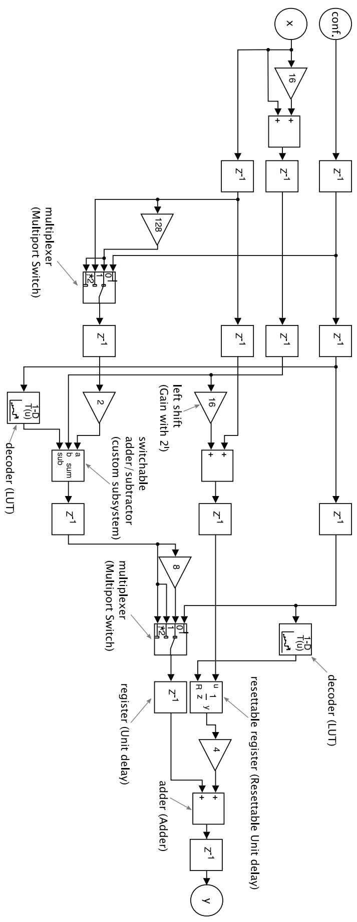

x
conf.
16
128
multiplexer (Multiport Switch)
left shift (Gain with 2^l)
switchable adder/subtractor (custom subsystem)
decoder (LUT)
register (Unit delay)
resettable register (Resettable Unit delay)
adder (Adder)
y

Figure D.1: Matlab/Simulink realization of the RCM shown in Figure 4.13.

using a constant gain block with the gain value 2^l. A switchable adder/subtractor is modeled by a custom block containing an adder, a subtractor and a multiplexer to switch between addition and subtraction, respectively. The decoder to select the right operation (addition/subtraction) based on the active configuration is modeled as LUT. Moreover, a resettable register (see Section 4.2.2) is already provided in the Matlab HDL Coder standard Simulink block set. Using the presented interface enables the integration of low-level optimized reconfigurable constant multipliers into high-level system development.

E FIR Benchmark for the Design Space Exploration

Table E.1: Coefficients of the benchmark for the design space exploration in Chap. 6.

MIRZAEI10_41_alt1 - lowpass with transition band 0.05..0.15 f_N: 1421, -940, -1338, -1966, -2679, -3323, -3738, -3750, -3200, -1960, 50, 2841, 6350, 10431, 14866, 19378, 23659, 27394, 30297, 32137, 32768, 32137, 30297, 27394, 23659, 19378, 14866, 10431, 6350, 2841, 50, -1960, -3200, -3750, -3738, -3323, -2679, -1966, -1338, -940, 1421
MIRZAEI10_41_alt2 - lowpass with transition band 0.1..0.2 f_N: -881, 843, 1169, 1568, 1780, 1582, 840, -454, -2124, -3825, -5080, -5364, -4218, -1370, 3173, 9082, 15725, 22263, 27783, 31472, 32768, 31472, 27783, 22263, 15725, 9082, 3173, -1370, -4218, -5364, -5080, -3825, -2124, -454, 840, 1582, 1780, 1568, 1169, 843, -881
MIRZAEI10_41_alt3 - lowpass with transition band 0.15..0.25 f_N: 546, -808, -1027, -1144, -845, -26, 1139, 2210, 2622, 1923, 21, -2634, -5087, -6109, -4599, -24, 7279, 16074, 24494, 30560, 32768, 30560, 24494, 16074, 7279, -24, -4599, -6109, -5087, -2634, 21, 1923, 2622, 2210, 1139, -26, -845, -1144, -1027, -808, 546
MIRZAEI10_41_alt4 - lowpass with transition band 0.2..0.3 f_N: -350, 769, 860, 690, 21, -929, -1580, -1331, -12, 1830, 3063, 2566, 14, -3623, -6238, -5483, -15, 9559, 20605, 29412, 32768, 29412, 20605, 9559, -15, -5483, -6238, -3623, 14, 2566, 3063, 1830, -12, -1331, -1580, -929, 21, 690, 860, 769, -350
MIRZAEI10_41_alt5 - lowpass with transition band 0.25..0.35 f_N: 226, -724, -679, -262, 536, 1109, 781, -477, -1751, -1760, -8, 2436, 3407, 1332, -3051, -6453, -4878, 3478, 16332, 28043, 32768, 28043, 16332, 3478, -4878, -6453, -3051, 1332, 3407, 2436, -8, -1760, -1751, -477, 781, 1109, 536, -262, -679, -724, 226
MIRZAEI10_41_alt6 - lowpass with transition band 0.3..0.4 f_N: -142, 677, 493, -90, -755, -677, 347, 1325, 934, -842, -2193, -1179, 1801, 3630, 1383, -3909, -6759, -1518, 11909, 26473, 32768, 26473, 11909, -1518, -6759, -3909, 1383, 3630, 1801, -1179, -2193, -842, 934, 1325, 347, -677, -755, -90, 493, 677, -142
MIRZAEI10_41_alt7 - lowpass with transition band 0.35..0.45 f_N: 81, -627, -313, 335, 665, 2, -945, -693, 813, 1552, 2, -2154, -1582, 1888, 3714, 3, -5913, -4977, 7571, 24726, 32768, 24726, 7571, -4977, -5913, 3, 3714, 1888, -1582, -2154, 2, 1552, 813, -693, -945, 2, 665, 335, -313, -627, 81
MIRZAEI10_41_alt8 - lowpass with transition band 0.4..0.5 f_N: -36, 576, 146, -458, -366, 526, 716, -474, -1173, 226, 1707, 316, -2272, -1299, 2807, 3045, -3248, -6705, 3538, 22825, 32768, 22825, 3538, -6705, -3248, 3045, 2807, -1299, -2272, 316, 1707, 226, -1173, -474, 716, 526, -366, -458, 146, 576, -36
MIRZAEI10_41_alt9 - lowpass with transition band 0.45..0.55 f_N: 0, -524, 0, 464, 0, -670, 0, 942, 0, -1306, 0, 1813, 0, -2574, 0, 3875, 0, -6772, 0, 20800, 32768, 20800, 0, -6772, 0, 3875, 0, -2574, 0, 1813, 0, -1306, 0, 942, 0, -670, 0, 464, 0, -524, 0
MIRZAEI10_41 - original filter from [73]: 157, 100, -264, -147, 430, 135, -666, -31, 960, -217, -1298, 675, 1648, -1463, -1977, 2841, 2245, -5779, -2422, 19057, 32768, 19057, -2422, -5779, 2245, 2841, -1977, -1463, 1648, 675, -1298, -217, 960, -31, -666, 135, 430, -147, -264, 100, 157

Bibliography

[1] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[2] A. D. Booth, “A Signed Binary Multiplication Technique,” *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[3] R. I. Hartley, “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, pp. 677–688, Oct. 1996.

[4] M. Faust and C. H. Chang, “Minimal Logic Depth Adder Tree Optimization for Multiple Constant Multiplication,” in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 457–460.

[5] A. G. Dempster and M. D. Macleod, “Constant Integer Multiplication Using Minimum Adders,” *Circuits, Devices and Systems, IEE Proceedings*, vol. 141, no. 5, pp. 407–413, Oct. 1994.

[6] O. Gustafsson, A. G. Dempster, and L. Wanhammar, “Extended Results for Minimum-Adder Constant Integer Multipliers,” in *IEEE International Symposium on Circuits and Systems*, 2002, pp. I–73–I–76.

[7] J. Thong and N. Nicolici, “A Novel Optimal Single Constant Multiplication Algorithm,” in *ACM/IEEE Design Automation Conference (DAC)*, 2010, pp. 613–616.

[8] Y. Voronenko and M. Puschel, “Multiplierless Multiple Constant Multiplication,” *Transactions on Algorithms (TALG)*, vol. 3, no. 2, May 2007.

[9] O. Gustafsson, “Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach,” in *Asilomar Conference on Signals, Systems and Computers*, 2008, pp. 1805–1809.

[10] L. Aksoy, E. O. Gunes, and P. Flores, “An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem,” in *IEEE NORCHIP*, 2008, pp. 41–46.

[11] L. Aksoy, E. O. Güneş, and P. Flores, "Search Algorithms for the Multiple Constant Multiplications Problem: Exact and Approximate," *Microprocessors and Microsystems*, vol. 34, no. 5, pp. 151–162, Aug. 2010.

[12] M. Kumm, D. Fanghänel, K. Möller, P. Zipf, and U. Meyer-Baese, "FIR Filter Optimization for Video Processing on FPGAs," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, p. 111, May 2013.

[13] A. G. Dempster and M. D. Macleod, "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 9, pp. 569–577, Sep. 1995.

[14] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 2, pp. 151–165, Feb. 1996.

[15] O. Gustafsson, "A difference based adder graph heuristic for multiple constant multiplication problems," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 1097–1100.

[16] M. Kumm, P. Zipf, M. Faust, and C.-H. Chang, "Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication," in *IEEE International Symposium on Circuits and Systems*, 2012, pp. 49–52.

[17] M. Kumm and P. Zipf, "High Speed Low Complexity FPGA-based FIR Filters Using Pipelined Adder Graphs," in *International Conference on Field-Programmable Technology (FPT)*, 2011, pp. 1–4.

[18] P. Lowenborg and H. Johansson, "Minimax Design of Adjustable-Bandwidth Linear-Phase FIR Filters," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 2, pp. 431–439, Feb. 2006.

[19] S. S. Demirsoy, I. Kale, and A. G. Dempster, "Reconfigurable Multiplier Blocks: Structures, Algorithm and Applications," *Circuits, Systems and Signal Processing*, vol. 26, no. 6, pp. 793–827, 2007.

[20] K. Möller, M. Kumm, P. Zipf, K. Groß, D. Lens, and H. Klingbeil, "FPGA Based Tunable Digital Filtering for Closed Loop RF Control in Synchrotons," *GSI Scientific Report*, 2013.

[21] L. Aksoy, P. Flores, and J. Monteiro, "Multiplierless Design of Folded DSP Blocks," *ACM Transactions on Design Automation of Electronic Systems*, vol. 20, no. 1, pp. 1–24, Nov. 2014.

[22] F. Qureshi and O. Gustafsson, "Low-Complexity Reconfigurable Complex Constant Multiplication for FFTs," in *IEEE International Symposium on Circuits and Systems*, 2009, pp. 1137–1140.

[23] M. Garrido, F. Qureshi, and O. Gustafsson, "Low-Complexity Multiplierless Constant Rotators Based on Combined Coefficient Selection and Shift-and-Add Implementation (CCSSI)," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 7, pp. 2002–2012, Jul. 2014.

[24] M. Faust, O. Gustafsson, and C.-H. Chang, "Reconfigurable Multiple Constant Multiplication using Minimum Adder Depth," in *Asilomar Conference on Signals, Systems and Computers*, 2010, pp. 1297–1301.

[25] S. S. Demirsoy, A. G. Dempster, and I. Kale, "Design Guidelines for Reconfigurable Multiplier Blocks," in *IEEE International Symposium on Circuits and Systems*, 2003, pp. IV–293–IV–296.

[26] R. H. Turner and R. F. Woods, "Highly Efficient, Limited Range Multipliers for LUT-Based FPGA Architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 10, pp. 1113–1118, Oct. 2004.

[27] P. Tummeltshammer, J. C. Hoe, and M. Püschel, "Time-Multiplexed Multiple-Constant Multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 9, pp. 1551–1563, Sep. 2007.

[28] J. Chen and C. H. Chang, "High-Level Synthesis Algorithm for the Design of Reconfigurable Constant Multiplier," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1844–1856, Dec. 2009.

[29] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, "Reconfigurable Constant Multiplication for FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 927–937, Jun. 2017.

[30] K. Chapman, "Constant Coefficient Multipliers for the XC4000E," *Xilinx Application Note*, 1996.

[31] A. Crosisier, D. J. Esteban, M. E. Levilio, and V. Riso, "Digital Filter for PCM Encoded Signals," *United States Patent No. 3777130*, 1973.

[32] S. Zohar, "New Hardware Realizations of Nonrecursive Digital Filters," *IEEE Transactions on Computers*, vol. 22, no. 4, pp. 328–338, Apr. 1973.

[33] M. Kumm, K. Möller, and P. Zipf, "PAGSuite Project Website," 2017. [Online]. Available: http://www.uni-kassel.de/go/pagsuite

[34] Digital Technology Group, University of Kassel, "Origami HLS: The art of folding," 2017. [Online]. Available: http://www.uni-kassel.de/go/origami

[35] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Springer US, 1992.

[36] *Virtex-6 FPGA Configurable Logic Block User Guide UG364 (v1.2)*, Xilinx Inc., 2012.

[37] Intel Corporation, *Stratix V Device Handbook Volume 1: Device Interfaces and Integration*, Dec. 2016.

[38] Xilinx, Inc., *Virtex-6 FPGA DSP48E1 Slice User Guide*, Feb. 2011.

[39] M. Kumm, S. Abbas, and P. Zipf, "An Efficient Softcore Multiplier Architecture for Xilinx FPGAs," in *IEEE Symposium on Computer Arithmetic*, 2015, pp. 18–25.

[40] F. de Dinechin and B. Pasca, "Large Multipliers with Fewer DSP Blocks," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 250–255.

[41] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, "Optimal Design of Large Multipliers for FPGAs," in *IEEE International Symposium on Computer Arithmetic*, accepted for presentation in 2017.

[42] U. Meyer-Baese, J. Chen, C. H. Chang, and A. G. Dempster, "A Comparison of Pipelined RAG-n and DA FPGA-based Multiplierless Filters," in *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2006, pp. 1555–1558.

[43] S. J. E. Wilton, S. shin Ang, and W. Luk, "The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays," in *International Workshop on Field-Programmable Logic and Applications (FPL)*. Springer-Verlag, 2004, pp. 719–728.

[44] Mentor, a Siemens Business, "ModelSim," 2017. [Online]. Available: https://www.mentor.com/products/fv/modelsim/

[45] Xilinx, Inc., *Partial Reconfiguration User Guide*, Oct. 2010.

[46] Intel Corporation, *Quartus II Handbook Volume 1: Design and Synthesis*, May 2015.

[47] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys, "Power Consumption Model for Partial and Dynamic Reconfiguration," in *International Conference on Reconfigurable Computing and FPGAs*, 2012, pp. 1–8.

[48] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 498–502.

[49] Xilinx, Inc., *Xilinx 7 Series FPGA Libraries Guide for HDL Designs*, Mar. 2011.

[50] K. Wiatr and E. Jamro, “Implementation of Multipliers in FPGA Structures,” in *International Symposium on Quality Electronic Design*, 2001, pp. 415–420.

[51] P. Jamieson and J. Rose, “Mapping Multiplexers onto Hard Multipliers in FPGAs,” in *International IEEE-NEWCAS Conference*, 2005, pp. 323–326.

[52] K. Chapman, “Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources,” *Application Note: Spartan-6, Virtex-6 Family, 7 Series FPGAs, Xilinx Inc.*, 2014.

[53] M. Kumm, “Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays,” Ph.D. dissertation, University of Kassel, Springer Fachmedien Wiesbaden, ISBN 978-3-658-13323-8, 2016.

[54] R. Bonamy, H. M. Pham, S. Pillement, and D. Chillet, “UPaRC–Ultra-Fast Power-Aware Reconfiguration Controller,” in *Design, Automation Test in Europe (DATE)*, 2012, pp. 1373–1378.

[55] J. Becker, M. Huebner, and M. Ullmann, “Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations,” in *Symposium on Integrated Circuits and Systems Design*, 2003, pp. 283–288.

[56] Xilinx, Inc., *Xilinx Power Estimator User Guide*, Oct. 2013.

[57] P. R. Cappello and K. Steiglitz, “Some Complexity Issues in Digital Signal Processing,” *Acoustics*, vol. 32, no. 5, pp. 1037–1041, Oct. 1984.

[58] M. Kumm, M. Hardieck, and P. Zipf, “Optimization of Constant Matrix Multiplication with Low Power and High Throughput,” *IEEE Transaction on Computers*, accepted for publication in 2017.

[59] R. Hartley, “Optimization of Canonic Signed Digit Multipliers for Filter Design,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1991, pp. 1992–1995.

[60] A. Dempster, O. Gustafsson, and J. O. Coleman, “Towards an Algorithm for Matrix Multiplier Blocks,” in *European Conference on Circuit Theory and Design (ECCTD)*, 2003, pp. 1–4.

[61] O. Gustafsson, H. Ohlsson, and L. Wanhammar, “Low-Complexity Constant Coefficient Matrix Multiplication Using a Minimum Spanning Tree Approach,” in *Nordic Signal Processing Symposium (NORSIG)*, 2004, pp. 141–144.

[62] M. J. Wirthlin, “Constant Coefficient Multiplication Using Look-Up Tables,” *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, vol. 36, no. 1, pp. 7–15, Jan. 2004.

[63] M. Kumm, K. Möller, and P. Zipf, “Dynamically Reconfigurable FIR filter Architectures with Fast Reconfiguration,” in *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2013, pp. 1–8.

[64] E. Jamro, “Parameterised Automated Generation of Convolers Implemented in FPGAs,” Ph.D. dissertation, Krakow University of Mining and Metallurgy, 2001.

[65] J. Hormigo, G. Caffarena, J. Oliver, and E. Boemo, “Self-Reconfigurable Constant Multiplier for FPGA,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 6, no. 3, p. 14, Oct. 2013.

[66] Xilinx, Inc., *LogiCORE IP Multiplier v11.2, Product Specification*, Mar. 2011.

[67] ——, *Virtex-5 Family Overview*, Feb. 2009.

[68] ——, *7 Series FPGAs Overview*, Nov. 2012.

[69] M. Kumm, K. Möller, and P. Zipf, “Reconfigurable FIR Filter using Distributed Arithmetic on FPGAs,” in *IEEE International Symposium on Circuits and Systems*, 2013, pp. 2058–2061.

[70] S. A. White, “Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review,” *IEEE ASSP Mag.*, vol. 6, no. 3, pp. 4–19, Jul. 1989.

[71] S. Mirzaei, R. Kastner, and A. Hosangadi, “Layout Aware Optimization of High Speed Fixed Coefficient FIR Filters for FPGAs,” *Int. Journal of Reconfigurable Computing*, vol. 2010, pp. 1–17, 2010.

[72] U. Meyer-Baese, G. Botella, D. Romero, and M. Kumm, “Optimization of High Speed Pipelining in FPGA-based FIR Filter Design Using Genetic Algorithm,” in *Proceedings of SPIE*, 2012, pp. 84 010R–84 010R–12.

[73] FIRsuite, “Suite of Constant Coefficient FIR Filters,” 2017. [Online]. Available: http://www.firsuite.net

[74] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, “Pipelined Reconfigurable Multiplication with Constants on FPGAs,” in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[75] K. Möller, M. Kumm, M. Garrido, and P. Zipf, “Optimal Shift Reassignment in Reconfigurable Multiplication Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, accepted for publication in 2017.

[76] K. Möller, M. Kumm, B. Barschtipan, and P. Zipf, “Dynamically Reconfigurable Constant Multiplication on FPGAs,” in *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014, pp. 159–169.

[77] D. R. Reddy, “Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort. Department of Computer Science,” 1977.

[78] G. Baeckler, M. Langhammer, J. Schleicher, and R. Yuan, “Logic Cell Supporting Addition of Three Binary Words,” *US Patent No 7565388, Altera Coop.*, 2009.

[79] J. M. Simkins and B. D. Philofsky, “Structures and Methods for Implementing Ternary Adders/Subtractors in Programmable Logic Devices,” *US Patent No 7274211, Xilinx Inc.*, 2006.

[80] M. Kumm, J. Willkomm, “OpenCores Ternary Adder,” 2017. [Online]. Available: http://opencores.org/project,ternary_adder

[81] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf, and U. Meyer-Baese, “Multiple Constant Multiplication with Ternary Adders,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[82] I. Free Software Foundation, “Using the GNU Compiler Collection,” 2017. [Online]. Available: https://gcc.gnu.org/

[83] U. of Illinois, “A C Language Family Frontend for LLVM,” 2017. [Online]. Available: http://clang.llvm.org

[84] F. de Dinechin and B. Pasca, “Designing Custom Arithmetic Data Paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Apr. 2012.

[85] FloPoCo Contributors, “Source Code Repository for FloPoCo Floating-Point Core Generator,” 2017. [Online]. Available: https://gforge.inria.fr/scm/?group_id=1030

[86] SPIRAL-Project. (2017). [Online]. Available: http://www.spiral.net

[87] M. Kumm, K. Möller, T. Schönwälder, P. Sittel, and P. Zipf, “Scalp Project Website,” 2017. [Online]. Available: https://digidev.digi.e-technik.uni-kassel.de/scalp/

[88] Gurobi Optimization, Inc., “Gurobi Optimizer Reference Manual,” 2017. [Online]. Available: http://www.gurobi.com

[89] T. H. Pham, S. A. Fahmy, and I. V. McLoughlin, “Low-Power Correlation for IEEE 802.16 OFDM Synchronization on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 8, pp. 1549–1553, Aug. 2013.

[90] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, “Architectural Modifications to Enhance the Floating-Point Performance of FPGAs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 177–187, Jan. 2008.

[91] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation.* John Wiley & Sons, 1999.

[92] K. Möller, M. Kumm, C.-F. Müller, and P. Zipf, “Model-based Hardware Design for FPGAs using Folding Transformations based on Subcircuits,” in *International Workshop on FPGAs for Software Programmers*, 2015, pp. 7–12.

[93] P. Sittel, M. Kumm, K. Möller, M. Hardieck, and P. Zipf, “High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits,” in *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2017.

[94] The MathWorks, Inc., “Generate Verilog and VHDL Code for FPGA and ASIC Designs,” 2017. [Online]. Available: https://www.mathworks.com/products/hdl-coder.html

[95] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “Lua 5.3 Reference Manual,” 2017. [Online]. Available: https://www.lua.org

This book addresses the question how run-time reconfigurable constant multipliers (RCMs) can be efficiently implemented on field programmable gate arrays (FPGAs). RCMs calculate the multiplication of an input number by one out of several constants which can be selected during run-time. This is important as constant multiplication is an essential operation in digital signal processing (DSP) applications. The evaluation of RCMs is done by considering reconfiguration using reconfigurable look-up tables (LUTs), reconfiguration using multiplexers (MUXs) and Partial Reconfiguration (PR). This book contributes two new methods to generate RCMs using the first two reconfiguration principles. First, a LUT-based constant multiplier is extended to be reconfigurable. Second, optimized constant multipliers without reconfiguration are fused using MUXs. Moreover, a general post-optimization for MUX-based RCMs is proposed. Finally, the design space produced in this way is analyzed using synthesis experiments. The contributed methods provide important trade-off points in the design space of RCMs on FPGAs.

ISBN 978-3-7376-0376-8

9 783737 603768 >