Springer Series in Advanced Microelectronics 51

Ayan Palchaudhuri
Rajat Subhra Chakraborty

High Performance Integer Arithmetic Circuit Design on FPGA

Architecture, Implementation and Design Automation

Springer

Springer Series in Advanced Microelectronics

Volume 51

The Springer Series in Advanced Microelectronics provides systematic information on all the topics relevant for the design, processing, and manufacturing of microelectronic devices. The books, each prepared by leading researchers or engineers in their fields, cover the basic and advanced aspects of topics such as wafer processing, materials, device design, device technologies, circuit design, VLSI implementation, and subsystem technology. The series forms a bridge between physics and engineering and the volumes will appeal to practicing engineers as well as research scientists.

More information about this series at http://www.springer.com/series/4076

Ayan Palchaudhuri · Rajat Subhra Chakraborty

High Performance Integer Arithmetic Circuit Design on FPGA

Architecture, Implementation and Design Automation

Springer

Ayan Palchaudhuri
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal
India

Rajat Subhra Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal
India

ISSN 1437-0387 ISSN 2197-6643 (electronic)
Springer Series in Advanced Microelectronics
ISBN 978-81-322-2519-5 ISBN 978-81-322-2520-1 (eBook)
DOI 10.1007/978-81-322-2520-1

Library of Congress Control Number: 2015943832

Springer New Delhi Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer (India) Pvt. Ltd. is part of Springer Science+Business Media (www.springer.com)

To Ma, for her continuous prayers,
support and inspiration.

Ayan Palchaudhuri

To Bunto, for her patience, love
and understanding.

Rajat Subhra Chakraborty

Preface

Realization of high performance arithmetic circuits targeted towards a specific family of the high-end Field Programmable Gate Arrays (FPGAs) continue to remain a challenging problem. Many fast arithmetic circuits proposed over the decades may not be amenable to efficient realization on a selected FPGA architecture. Experience has shown that current CAD tools for FPGAs are often unable to infer the native architectural components efficiently from the given input Hardware Description Language (HDL) specification of the circuit, as they explore only a small design space close to the input architectural description. The logic synthesis techniques inherent to the CAD tools are also often unable to apply the proper Boolean identities and perform appropriate algebraic factoring and sub-expression sharing, especially when intermediate signals are tapped out or registered. *Primitive instantiation* is an effective approach for optimization of designs on the Xilinx FPGA platform, and is often simpler than rewriting the Register Transfer Level (RTL) code to coax the logic synthesis tool to infer the desired architectural components. In addition, the FPGA CAD tools often fail to achieve an efficient placement of logic blocks on the FPGA fabric, resulting in higher routing delays.

In this book, we describe the optimized implementations of several arithmetic datapath, controlpath, and pseudorandom sequence generator circuits. We explore regular, modular, cascadable, and bit-sliced architectures for these circuits, by directly instantiating the target FPGA-specific primitives in the HDL specifications of the circuits. We justify every proposed architecture with detailed mathematical analyses. We improve performance by enforcing a constrained placement of the circuit building blocks, by placing the logically related hardware primitives in close proximity to one another, thereby minimizing the routing delay. This is accomplished by supplying relevant placement constraints in the Xilinx proprietary "User Constraints File" (.ucf) format to the FPGA CAD tool.

Taking advantage of the regularity of the architectures of the circuits proposed by us, the HDL specifications of the circuits as well as the placement constraints can be automatically generated. We have implemented a GUI-based CAD tool named *FlexiCore* integrated with the *Xilinx ISE* (Integrated Software Environment)

design environment for design automation of platform-specific high performance arithmetic circuits from user-level specifications. This tool was used to implement the proposed circuits, as well as hardware implementations of two integer arithmetic algorithms (Greatest Common Divisor (GCD) using Binary GCD algorithm and matrix multiplication using Distributed Arithmetic (DA)) where several of the proposed circuits were used as building blocks. Implementation results demonstrate higher performance and superior operand-width scalability at acceptable power-delay product (PDP) for the proposed circuits, with respect to implementations derived through other existing approaches.

Kharagpur

Ayan Palchaudhuri
Rajat Subhra Chakraborty

Acknowledgments

The authors thank Prof. Anindya Sundar Dhar, Department of Electronics and Electrical Communication Engineering, IIT Kharagpur, and Dr. Debdeep Mukhopadhyay, Department of Computer Science and Engineering, IIT Kharagpur, for their valuable insights into the work. The authors also acknowledge two undergraduate students of the Department of Computer Science and Engineering, IIT Kharagpur, Mohammad Salman and Sreemukh Kardas, for their contributions in developing the proposed CAD tool, *FlexiCore*.

This research was funded by a start-up research grant provided by IIT Kharagpur to Rajat Subhra Chakraborty.

Contents

Acronyms

ASIC	Application-Specific Integrated Circuit
BIST	Built-In Self-Test
CA	Cellular Automata
CAA	Cellular Automata Array
CAD	Computer Aided Design
CDI	Configuration Data In
CDO	Configuration Data Out
CE	Clock Enable
CL	Combinational Logic
CLB	Configurable Logic Block
CORDIC	COordinate Rotation DIgital Computer
DA	Distributed Arithmetic
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FloPoCo	Floating-Point Cores
FF	Flip-Flop
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCD	Greatest Common Divisor
GUI	Graphical User Interface
HDL	Hardware Description Language
HKMG	High-K Metal Gate
HPL	High Performance Low Power
IP	Intellectual Property
ISE	Integrated Software Environment
LFSR	Linear Feedback Shift Register
LNS	Logarithmic Number System
LUT	Look-Up Table
MACC	Multiply Accumulate

MCNC	Microelectronics Center of North Carolina
MISR	Multiple-Input Signature Register
PDP	Power-Delay Product
PIPO	Parallel-In Parallel-Out
PP	Partial Product
RCA	Ripple Carry Adder
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
TCL	Tool Command Language
TSMC	Taiwan Semiconductor Manufacturing Company Limited
UCF	User Constraints File
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XST	Xilinx Synthesis Tool

About the Authors

Ayan Palchaudhuri is a Ph.D. student in the Department of Electronics and Electrical Communication Engineering (E&ECE) of Indian Institute of Technology (IIT) Kharagpur. He has received the M.S. degree from the Department of Computer Science and Engineering (CSE), IIT Kharagpur, in 2015. He has over 2.5 years of work experience as a Junior Project Assistant in the Department of CSE, IIT Kharagpur. His research interests include VLSI Architecture Design and Computer Arithmetic. He is the co-author of two conference papers, one journal, one book chapter, and a patent has been filed based on his research work. His research work has been recognized with the Best Poster Award in the Student Research Symposium of the 21st IEEE International Conference on High Performance Computing (HiPC) 2014.

Rajat Subhra Chakraborty is Assistant Professor in the Computer Science and Engineering Department of Indian Institute of Technology Kharagpur. He has a Ph.D. in Computer Engineering from Case Western Reserve University (Ohio, USA) and a B.E. (Hons.) in Electronics and Telecommunication Engineering from Jadavpur University (India) in 2005. He has work experience at National Semiconductor and AMD. His research interests include: Hardware Security, VLSI Design and Design Automation, and Reversible Watermarking for digital content protection. He is the co-author of two published books, four book chapters, and over 50 publications in international journals and conferences of repute. He is one of the recipients of the "IBM Faculty Award" for 2012, and a "Royal Academy of Engineering (UK) Fellowship" in 2014. He holds one U.S. patent, and two more international patents and three Indian patents have been filed based on his research work. Dr. Chakraborty is a member of IEEE and ACM.

Chapter 1
Introduction

Abstract This chapter presents a concise overview of FPGA-based architecture design. Certain existing research work behind proposing new FPGA architectures and CAD heuristics to overcome the design limitations have been discussed. It also unfolds the limitations of the FPGA CAD tool that are currently popular for arithmetic core generation. A methodology that uses the target FPGA specific primitive instantiation-based approach and constrained placement exercise has been proposed as a superior alternative in comparison to design implementations available in literature. The major contributions of this book have also been listed.

1.1 Background of FPGA-Based Design

Researchers over the years have proposed various Field Programmable Gate Array (FPGA) architectures for high performance realization of digital circuits, particularly those that have long cascading delay, e.g., ripple carry adder. In [19], an enhanced cascade circuit with dedicated routing structures for fast propagation of signals between cells was proposed to achieve a significant speedup. In [11], the authors had considered a folding method of logic functions and its adaptation to map to proposed Look-Up Table (LUT) architectures, such that the memory requirement for realization of the logic functions can be significantly reduced. It has also been studied that reducing the number of stages of programmable routing (e.g., routing through switchboxes) can significantly reduce the critical path [17]. This can be achieved through use of LUTs which support maximum functionality and dedicated routing along with other logic blocks. Experiments have confirmed that 6-input LUTs provide the right trade-off between critical path delay and design die size [3]. With culmination of all such similar research paradigms, modern FPGAs have architectural features amenable to the efficient implementation of combinational functions of arbitrary complexity in the form of LUTs, along with fast special-purpose logic and routing resources for propagating *carry* signals between adjacent logic blocks in form of *carry chain* [9].

However, with a significant increase in circuit complexity for FPGA-based designs, even the most sophisticated Computer Aided Design (CAD) tools often result in circuit implementations with unsatisfactory performance and resource

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_1

requirements owing to their inability to optimally exploit the underlying FPGA architecture and their dedicated routing fabric. Hence, implementations derived through the standard automatic logic synthesis-based design flow starting with the behavioral Hardware Description Language (HDL) as the mode of design entry of the circuit, can be outperformed by more low-level "custom" design techniques. Extensive experimentation with the Xilinx FPGA CAD tool has shown that though the tool is able to infer the desired hardware primitives for many combinational logic circuits, it generally fails to do so when we synthesize sequential or pipelined architectures. In addition, the current FPGA CAD tools often fail to place the technology mapped sub-circuits at locations such that circuit delay is minimized. To overcome these limitations, a designer needs to identify the basic building blocks of the required circuit, and make an effort to optimally construct them from the hardware primitives available on the FPGA and ensure their proper placement. This book discusses several FPGA-specific design techniques that need to be adopted for optimal realization of high performance circuits and presents relevant case studies.

The rest of the chapter is organized as follows. In Sect. 1.2, we discuss the limitations of FPGA CAD tools for realization of high performance circuits. In Sect. 1.3, we describe the design philosophy that must be adopted to overcome the limitations posed by current FPGA CAD tools. Section 1.4 discusses two popular FPGA CAD tools—the standard *Xilinx Integrated Software Environment* (ISE)-based Graphical User Interface (GUI) for IP Core Generator and *FloPoCo* (in short for Floating–Point Cores) where user inputs circuit specifications and the CAD tool generates the corresponding synthesizable HDL specification of the circuit. In Sect. 1.5, we mention some of the recent works on architectural realization of arithmetic circuits on Xilinx FPGAs. We state the major contributions of this book in Sect. 1.6. The organization of the book is presented in Sect. 1.7.

1.2 Limitations of FPGA CAD Tools

Modern CAD tools for FPGA platform facilitate the automatic mapping of binary addition logic to carry-chain structures to accelerate signal propagation, thereby achieving a speedup in the critical paths. In addition, existence of carry-chain simplifies routing by avoiding switch-boxes in the general routing fabric, and also increases the functional capability of the FPGA logic slices. Such benefits have been experimentally evaluated on the basis of combinational MCNC benchmarks in [15]. Though synthesis tools for Xilinx Virtex architectures are able to map the binary addition logic or a homogeneous wide AND and OR gates to *carry chain* fabric [16], or can infer the *wide function multiplexers* native to an FPGA slice, it fails to do so as soon as the final carry outputs of individual slices [14] or intermediate LUT outputs are tapped out or registered to facilitate pipelining of the architecture. A possible reason might be that Xilinx FPGA logic slices do not support dedicated hardware for registering such signals, and the CAD tools fail to infer the desired implementation. In such a scenario, the designer has to spell out special directives in the HDL of

the design or associated "constraints files" to the FPGA CAD tool, so that in the *packing* [1] or *clustering* step (as known in the FPGA CAD literature) of the FPGA design flow, the technology mapped circuit is efficiently "packed" into the available hardware resources.

1.3 Overview of Design Philosophy for FPGAs

Most current FPGA vendors allow direct instantiation of the available primitives in the HDL code [21], and a "mixed" style of HDL coding, where high-level behavioral code is intermingled with relatively "low-level" structural code. Placement steps also need to be constrained and controlled, as the CAD placement tools, if allowed to perform unconstrained placement and routing, often result in large routing delays. This happens because the technology mapped logic elements get unevenly distributed across the FPGA fabric, resulting in greater routing and interconnect delays. This contributes to a major portion of the circuit critical path delay. Although most modern FPGA vendors provide special hardware IPs for common integer arithmetic operations which can be directly instantiated in the HDL code, we would demonstrate that we can do better by adopting careful design techniques.

To follow the methodology promoted above, the designer must possess a thorough understanding of the target FPGA architecture, the routing fabric, the available design elements in the form of primitives and macros, and how to configure them to achieve the required functionality. Also, the structure of the circuits must have sufficient regularity to automate their design, and to allow efficient placement and routing. Arithmetic circuits and Finite State Machines (FSMs) with regular structures (e.g., counters and shift registers) are thus the ideal candidates for this methodology.

1.3.1 Target FPGA-Specific Hardware Primitive Instantiation

With sufficient modularity in the circuit architecture, it becomes easy to automate the generation of the HDL code, and associated constraint files which are themselves very regular in their grammar. Target FPGA-specific *primitive instantiation* is an effective approach for optimization of designs on the Xilinx FPGA platform [7], and is often the only approach. The primitive instantiation-based methodology is also simpler than rewriting the Register Transfer Level (RTL) code to coax the logic synthesis tool to infer the desired architectural components. However, in general, the entire circuit might not be amenable to the primitive instantiation-based design approach. In such cases, this approach can be adopted to design only those parts of the circuit that are amenable to such a methodology, and contribute significantly to the critical path delay. The only disadvantage of using such a design methodology is that the design becomes less portable, and becomes harder to maintain. In spite of this, the methodology is very effective in practice, considering the facts that (a) often

the target FPGA platform is known before the circuit is designed and that (b) FPGAs from a related family from the same vendor are often backward compatible regarding the design elements (primitives and macros) supported. For example, newer versions of FPGAs of the "Virtex" family from Xilinx are expected to support primitives supported in some older Virtex versions. Thus, the HDL code for instantiating primitives targeting the older versions, and the constraints file to control the placement, can be reused in the newer version after small tweaks, if necessary. Added with the advantage of being able to automatically generate the circuit descriptions and the constraints, this is an attractive methodology.

1.4 Existing FPGA CAD Tools

The most popular FPGA CAD tools for arithmetic core generation are the *Xilinx IP Core Generator* and an open source tool called *FloPoCo*. We briefly describe their main features and the relevant shortcomings.

1.4.1 Xilinx IP Core Generator

The Xilinx IP Core Generator is part of the standard Xilinx ISE distribution, It includes a GUI-based utility, through which synthesizable HDL code (for common integer arithmetic circuits such as adders, multipliers, accumulators, counters, etc.) can be automatically generated. Designers can generate both combinational and pipelined versions of the arithmetic circuits, where the user enters the parameter *latency* as input for pipelined architecture realizations. Although such HDL automatically generated by the Xilinx software is functionally correct, it fails to give high performance when implemented, because the synthesis tool performs an inefficient technology mapping of the circuit, and the inferred logic elements are usually scattered in an apparently random fashion across the FPGA fabric, thereby causing large routing delays and affecting critical paths. Xilinx also allows the direct instantiation of "Digital Signal Processing" (DSP) hardware macros in the HDL targeted for FPGAs, which are highly customized dedicated arithmetic circuits. Individual Virtex-5 DSP slices can operate at a maximum frequency of 550 MHz [20], Virtex-6 DSP slices can operate at a maximum frequency of 600 MHz [22]; such operating frequencies are attainable at very low latency and are also suitable for low power applications. However, as we would demonstrate, they have performance limitations for large operand widths, and can be outperformed by the proposed circuits with aggressive pipelining, coupled with compact and constrained placement of logic primitives.

1.4.2 FloPoCo *(Floating-Point Cores)*

FloPoCo is an open-source C++ framework for generating arithmetic cores for FPGAs [8]. *FloPoCo* provides a command-line interface through which the user can input operator specifications, and the program generates the corresponding synthesizable VHDL (Very High Speed Integrated Circuit Hardware Description Language) code. The main features of *FloPoCo* as listed in [6] are as follows:

- Supports integer, fixed point, floating point, and *Logarithmic Number System* (LNS) arithmetic.
- Supports pipelining by allowing the user to specify the desired operating frequency.
- Allows the user to specify the target FPGA implementation platform, and generates synthesizable VHDL code optimized for that target platform. In addition, *FloPoCo* performs target platform-specific pipelining, as its frequency-directed pipelining takes into consideration the timing information about the target FPGA platform [6]. *FloPoCo* comes with such models for the main FPGA families from both Xilinx and Altera.

However, detailed experimentation with the latest released version of *FloPoCo* (v 2.5.0) [8], and implementation and characterization of the integer arithmetic circuit descriptions generated by it indicate the following drawbacks:

- *FloPoCo* only generates pure behavioral VHDL code which cannot correctly infer the desired hardware primitives of the target FPGA platform. Consequently, it has no control over the inference and placement of logic blocks on the FPGA fabric. This makes the performance of the circuit post-synthesis worse than the target frequency specified by the user. Thus, *FloPoCo* provides no guarantee that the target frequency specified would be met in the final implementation.
- *FloPoCo* at times create very deep pipelines, apparently to meet input frequency constraints, but post place-and-route implementations do not guarantee that the delay constraints are met.
- Pipelining behavior of *FloPoCo* is very inconsistent. It was observed that for adder circuit implementations, *FloPoCo* creates very deep pipelines, whereas it creates fairly unbalanced and irregular pipelines for dual subtractor implementations, where each of the pipeline stages have different complexities. On the contrary, it completely avoids pipelining the integer multipliers, but, however, generates an erroneous comment in the VHDL code that it has achieved single stage pipelining. Similar observations have been made for the squarer circuits where *FloPoCo* is unable to pipeline squarer circuits whose input operand bit-widths are lower than 14, but create highly inefficient pipelines for squarer circuits of higher bit-widths. Our observations about these inconsistencies in the pipelining behavior of the current version of *FloPoCo* have been concurred with by the creators of *FloPoCo* through personal correspondence. They have acknowledged that a bug exists in their program, which they have filed and would probably be taken care of in future releases.

Irrespective of the behavior of existing options for automatic generation of arithmetic circuit cores targeting FPGAs, the important approach to note is that most or all of them are agnostic of the "low–level" architecture of the target FPGA platform, and its routing issues. Consequently, they can be predicted to be unable to take advantage of the hardware primitives, and to generate the most optimal circuit descriptions for the target FPGA.

1.5 Recent Works on High Performance Circuit Realization on Xilinx FPGAs

In the recent literature, design of fast adders [18, 24] and absolute difference circuits [13] on the Xilinx Virtex-5 family of FPGAs have been reported. These works discuss the design of fast and efficient architectures by exploiting the *carry chain* and 6-input LUTs of the Virtex-5 family. In [2, 5], the authors have proposed methods to integrate DSP blocks along with fabric logic for realization of multiplier circuits. Similar approaches have been reported in [23] for realization of large integer squarers on FPGA. Researchers have also been proposing heuristics for efficient synthesis of FPGA-based circuits, notably on the class of circuits called "compressor trees" that generalizes multioperand addition, and the partial product reduction trees of parallel multipliers using carry-save arithmetic [12].

1.6 Major Contributions of the Book

In this book, we explore the architecture of several high performance integer arithmetic circuits, built using primitive instantiation and constrained placement of target-specific primitives on the Xilinx Virtex-5 and Virtex-6 FPGA fabric, and a CAD tool to automate their design. The architectures are essentially "bit-sliced", and conceived in a way such that the bit-slices can be directly mapped to hardware primitives available on the FPGA. In addition, the placement of the primitives is carefully constrained to improve the critical path delay, and throughput is increased further by appropriate pipelining [10]. We call the CAD tool *FlexiCore*, in short for "**Flexi**ble Arithmetic Soft **Core** Generator." It is flexible in a sense that the operand widths for the mapped circuits can be varied, and the CAD tool allows partial control to the user over the placement of the circuits on the FPGA fabric. The CAD tool is integrated into the standard Xilinx ISE design environment, thus making it extremely convenient to a large user community, although nothing prevents our CAD tool (with minor modifications) to be part of other FPGA CAD frameworks. The circuits currently supported by the CAD tool are the ones most widely used in the domains of integer arithmetic algorithms, digital signal processing, and digital image processing. The use of the tool can also be easily extended for Xilinx FPGA platforms other than Virtex-5, after

taking care of small modifications (e.g., changes in the "library primitive" names supported by Xilinx).

To summarize, the following are the main contributions of our work:

- We develop a methodology to design high performance widely used integer arithmetic circuits and pseudorandom binary sequence generators for Xilinx Virtex-5 and Virtex-6 FPGAs, while utilizing optimum hardware resources and having acceptable power-delay product (PDP) wherever possible. The main insight is to make optimal use of certain hardware primitives of the target FPGA platform, and constrained placement of the primitives, combined with pipelining (as required).
- We elaborate on certain useful observations which act as important guidelines for compact and high-performance realization of circuits realized using fabric logic on modern high-end FPGAs from Xilinx. These involve manipulation of the Boolean logic equations a priori in the HDL circuit descriptions, to forms that can be optimally mapped to the native target architecture by the CAD software.
- We adopt a *bit-sliced* design paradigm where an entire arithmetic circuit is built using identical modules of smaller bit width. As the stages of the building blocks are identical, it is easier to perform an optimal, fine-grained, and forward path pipelining. For some (two) circuits, we optimize the performance of the implementations derived using previously proposed architectures, while for the other circuits, we develop the architectures from first principles. In all cases, we provide in detail the Boolean logic-based mathematical analyses and proofs of correctness leading to the architectures, at times supplementing the analyses presented in the original sources of the architectures.
- The designs generated by our CAD framework show better operand-width scalability in comparison to previously proposed designs [24], i.e., lesser decrease of performance with the increase in operand width. However, any performance (speed) deterioration observed with increase in operand width is mainly attributed to the geometry of the FPGA devices and certain complex routing issues (either due to the fabric architecture or due to the complex digital logic circuitry), both of which are practically unavoidable by the user.
- Our designs outperform the circuits built using the GUI-based circuit generator utility in-built in *Xilinx ISE*, or by using the DSP slice hard macros, or the arithmetic cores generated by *FloPoCo*.
- Our CAD tool is integrated in the *Xilinx ISE* design environment to automate the design of the circuits, including automatic generation of the necessary placement constraint files.
- We demonstrate the effectiveness of the proposed design methodology by the complete "bottom-up" design of a 32-bit Greatest Common Divisor (GCD) calculation circuit, and a Distributed Arithmetic (DA)-based Matrix Multiplication circuit that utilizes several circuit building blocks generated by our CAD utility.

To the best of our knowledge, custom level implementation of logic elements on FPGA fabric for obtaining superior speed performances has not been extensively studied or reported in research articles. The previous literature reports high level HDL modeling for FPGA-based arithmetic circuit design [4], however, our design

philosophy for FPGAs is slightly different in nature, as would be revealed in more detail in the upcoming chapters. This book aims to address this design philosophy and validate the entire exercise by presenting suitable examples and case studies.

1.7 Organization of the Book

- Chapter 1 is the introductory chapter which discusses some of the existing work behind proposing new FPGA architectures, mapping of logic functions into proposed LUT architectures, along with limitations of the CAD algorithms for optimized mapping of logic into the FPGA fabric. It also presents the modern-day FPGA architectural components available in the high-end FPGA families coming from Xilinx. The existing Xilinx FPGA CAD tools available to designers and their limitations have been mentioned. A methodology to overcome these limitations using the art of primitive instantiation and constrained placement exercise has been proposed.
- Chapter 2 provides architectural details for the Virtex-5 and Virtex-6 FPGA platforms. It also discusses the different modes of implementations that *Xilinx ISE* provides to generate arithmetic circuit descriptions.
- Chapter 3 provides the fabric component-based approach for design of high performance integer arithmetic circuits for FPGAs. The Boolean logic manipulation and restructuring involved to map circuits optimally into the available FPGA hardware primitives has been presented.
- Chapter 4 presents the pipelined implementations of common arithmetic datapath circuits such as integer adder, absolute difference circuit, combined unsigned and two's complement integer multiplier and squarer, and universal shift register. We also present their Boolean logic-based mathematical analyses, proofs of correctness, and the post place-and-route implementation results clearly reveal the superiority and advantages of our proposed design philosophy.
- Chapter 5 presents the pipelined implementations of arithmetic controlpath circuits such as integer comparator and loadable bidirectional counter along with their Boolean logic-based mathematical analyses, and the post place-and-route implementation results. Once again, the results clearly reveal the superiority and advantages of our proposed design philosophy.
- Chapter 6 discusses the FPGA-based implementation of cellular automata (CA)-based pseudorandom binary sequence generator.
- Chapter 7 introduces the CAD tool for design automation, *FlexiCore*, for automating the HDL of arithmetic circuits along with the placement constraint related files developed by us. The CAD tool can support the generation of multiple module-based designs which has been put to use for the design of a Greatest Common Divisor (GCD) circuit and a Distributed Arithmetic (DA)-based matrix multiplication circuit. The superior performance of the designs whose descriptions have been generated by *FlexiCore* have been tabulated.
- Chapter 8 summarizes the contributions and draws the future research directions.

1.8 Summary

In this chapter, we have presented a concise overview of the limitations of the existing CAD tools for FPGA realization regarding optimal synthesis, technology mapping, and placement for realization of high performance arithmetic pipelined blocks. The overview of the design philosophy adopted by us for FPGA realization of arithmetic circuits has been provided with some recent works on high performance FPGA-based arithmetic circuit design. We have also spelt out the major contributions of the book. In the next chapter, we will discuss the architecture of the building blocks of our target FPGA platform.

References

1. Ahmed, T., Kundarewich, P.D., Anderson, J.H.: Packing techniques for virtex-5 FPGAs. ACM Trans. Reconfig. Technol. Syst. (TRETS). **2**(18), 18:1–18:24 (2009)
2. Athow, J.L., Al-Khalili, A.J.: Implementation of large–integer hardware multiplier in Xilinx FPGA. In: 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 1300–1303 (2008)
3. Cosoroaba, A., Rivoallon, F.: Xilinx Inc., White Paper: Virtex-5 Family of FPGAs, Achieving Higher System Performance with the Virtex-5 Family of FPGAs, WP245 (v1.1.1). http://www.xilinx.com/support/documentation/white_papers/wp245.pdf. Cited 7 July 2006
4. Deschamps, J.-P., Sutter, G.D., Canto, E.: Guide to FPGA Implementation of Arithmetic Functions. Ser. Lecture Notes in Electrical Engineering. Springer, vol. 149 (2012)
5. de Dinechin, F., Pasca, B.: Large Multipliers With Fewer DSP Blocks. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 250–255 (2009)
6. de Dinechin, F., Pasca, B.: Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. **28**(3), 18–27 (2011)
7. Ehliar, A.: Optimizing Xilinx designs through primitive instantiation. In: Proceedings of the 7th FPGAworld Conference, pp. 20–27 (2010)
8. FloPoCo.: Arithmetic Core Generator. http://flopoco.gforge.inria.fr/. Cited 12 Sep 2013
9. Hauck, S., Hosler, M.M., Fry, T.W.: High-Performance Carry Chains for FPGA's. In: IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **8**(2), 138–147 (2000)
10. Jump, J.R., Ahuja, S.R.: Effective pipelining of digital systems. IEEE Trans. Comput. **27**(9), 855–865 (1978)
11. Kimura, S., Horiyama, T., Nakanishi, M., Kajihara, H.: Folding of logic functions and its application to look up table compaction. In: IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 694–697 (2002)
12. Parandeh-Afshar, H., Neogy, A., Brisk, P., Ienne, P.: Compressor Tree Synthesis on Commercial High-Performance FPGAs. ACM TRETS. **4**(4), 39:1–39:19 (2011)
13. Perri, S., Zicari, P., Corsonello, P.: Efficient absolute difference circuits in Virtex-5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 309–313 (2010)
14. Preuβer, T.B., Spallek, R.G.: Mapping basic prefix computations to fast carry-chain structures. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 604–608 (2009)
15. Preuβer, T.B., Spallek, R.G.: Enhancing FPGA device capabilities by the automatic logic mapping to additive carry chain. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 318–325 (2010)
16. Preuβer, T.B., Zabel, M., Spallek, R.G.: Accelerating computations on FPGA carry chains by operand compaction. In: 20th IEEE Symposium on Computer Arithmetic (ARITH), pp. 95–102 (2011)

17. Singh, S., Rose, J., Chow, P., Lewis, D.: The effect of logic block architecture on FPGA performance. IEEE J. Solid-State Circ. **27**(3), 281–287 (1992)
18. Vazquez, M., Sutter, G., Bioul, G., Deschamps, J.P.: Decimal adders/subtractors in FPGA: Efficient 6-input LUT implementations. In: International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 42–47 (2009)
19. Woo, N.-S.: Revisiting the cascade circuit in logic cells of lookup table based FPGAs. In: Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays, pp. 90–96 (1995)
20. Xilinx Inc.: Virtex-5 FPGA XtremeDSP Design Considerations User Guide, UG193 (v3.5). http://www.xilinx.com/support/documentation/user_guides/ug193.pdf. Cited 26 Jan 2012
21. Xilinx Inc.: Virtex-5 Libraries Guide for HDL Design, UG621 (v11.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex5_hdl.pdf. Cited 6 Sep 2009
22. Xilinx Inc.: Virtex-6 FPGA DSP48E1 Slice User Guide, UG369 (v1.3). http://www.xilinx.com/support/documentation/user_guides/ug369.pdf. Cited 14 Feb 2011
23. Xu, S., Fahmy, S.A., McLoughlin I.V.: Efficient large integer squarers on FPGA. In: 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 198–201 (2013)
24. Zicari, P., Perri, S.: A fast carry chain adder for Virtex-5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 304–308 (2010)

Chapter 2
Architecture of Target FPGA Platform

Abstract This chapter provides an insight into the architecture of Configurable Logic Blocks (CLBs), the basic building blocks of a FPGA, including details of the Look-Up Tables, wide function multiplexers, carry chains, flip-flops, and DSP slices. It also gives an overview of the different modes of implementation supported by Xilinx ISE to realize arithmetic functions.

2.1 Introduction

Current FPGAs such as the advanced Virtex and Spartan families of Xilinx FPGAs (e.g., Virtex-5, Virtex-6 and Spartan-6), promise immense hardware logic support, at higher integration, lower power consumption, and maximum performance. Maximum system performance requires a balanced mix of performance-efficient FPGA components: logic fabric (Look-Up Tables (LUTs), special functions like carry chains and dedicated multiplexers, flip-flops (FFs)), on-chip RAM, DSP blocks, and I/Os. Virtex-5 FPGAs have been the first FPGA device fabricated at the 65 nm CMOS technology node. Switching from 90 nm (for Virtex-4 FPGAs) to 65 nm (for Virtex-5 FPGAs) [4] have promised the above-mentioned advantages. Spartan-6 FPGAs, on the other hand, are built on a mature 45 nm low-power copper process technology [3] that delivers the optimal balance of cost, power and performance, whereas Virtex-6 FPGAs are built using a 40 nm state-of-the-art copper process technology, and are a programmable alternative to custom ASIC technology [9]. Xilinx 7 series FPGAs leverage the unprecedented power, performance, and capacity enabled by TSMC's (Taiwan Semiconductor Manufacturing Company Limited) 28 nm [2] high-k metal gate (HKMG), high performance, low power (HPL) process technology, and the unparalleled scalability afforded by the FPGA industry's first scalable, optimized architecture to provide a comprehensive platform base for next-generation systems.

The Configurable Logic Blocks (CLBs) of FPGAs are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix as shown in Fig. 2.1. Each CLB element for Virtex-5 and Virtex-6 series of FPGAs (that have been our target platform for implementation) contain a pair of slices. These two slices do not have direct connections to each other, and each slice is organized as a column.

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_2

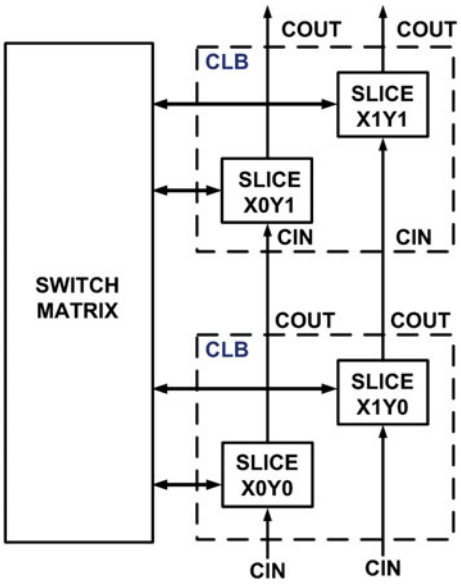

Fig. 2.1 Arrangement of slices within the CLB [5]

The Xilinx tools designate slices with the following definitions [5]. An "X" followed by a number identifies the position of each slice in a pair as well as the column position of the slice. A "Y" followed by a number identifies a row of slices. The number remains the same within a CLB, but counts up in sequence from one CLB row to the next CLB row.

The rest of the chapter is organized as follows. In Sect. 2.2, we present the slice architecture for Xilinx Virtex-5 FPGAs. In Sect. 2.3, we present the additional and modified features that Virtex-6 FPGAs offer in comparison to Virtex-5 FPGAs. A brief overview of the DSP slice architecture has been presented in Sect. 2.4. The different modes of implementation—fabric and DSP slice logic have been discussed in Sect. 2.5. We conclude in Sect. 2.6.

2.2 Fabric Slice Architecture for Virtex-5 FPGAs

The CLBs of Xilinx FPGA are the main logic resources for implementing combinational and sequential circuits. A typical CLB of Virtex-5 FPGA contains 2 "slices," with each slice (called a "SLICEL" or "SLICEM" in Xilinx terminology depending on the nature of LUTs) comprising of four 6-input logic-function generators or LUTs, four storage elements or FFs, three wide function multiplexers, and a length-4 carry chain comprising of multiplexers and XOR gates [1, 5] as shown in Fig. 2.2. All these elements are used by the slices for realization of arithmetic, logic, and memory functions.

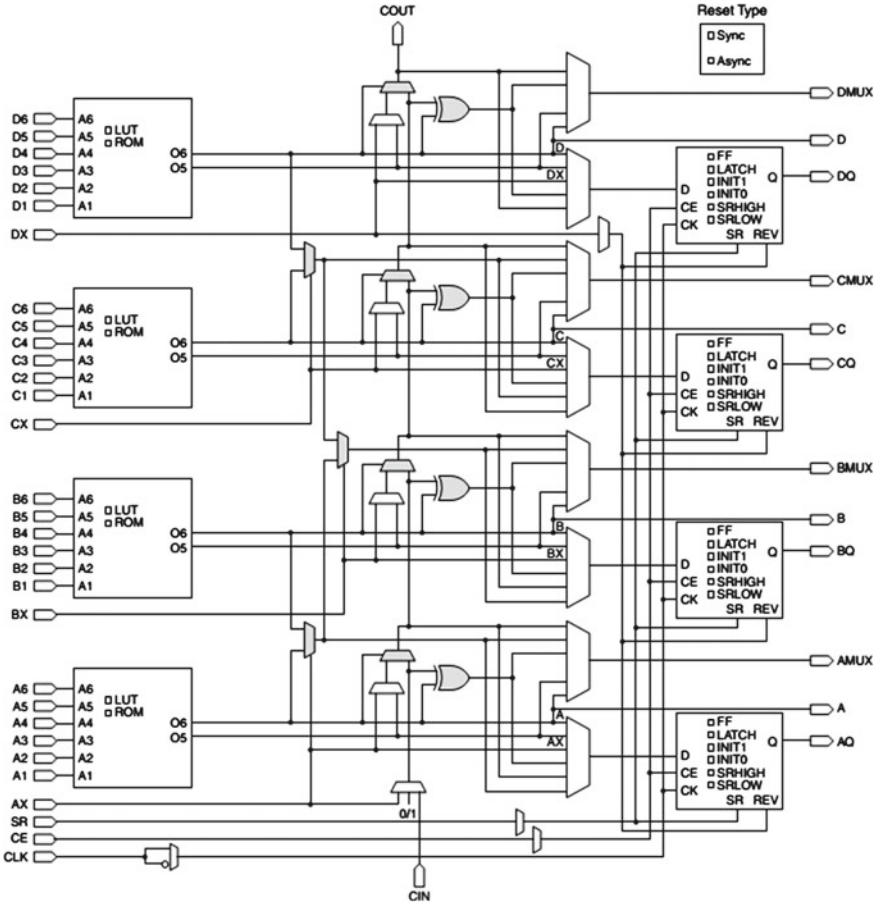

Fig. 2.2 Xilinx Virtex-5 slice architecture [5]

The Xilinx Virtex-5 family has been the first FPGA platform to offer a true 6-input LUT, with fully independent (not shared) inputs. This allows the implementation of functions with higher operand width and reduces the number of logic levels between registers. The 6-input LUT can also be configured as a 5 (or less) input, 2-output logic function with shared inputs, thereby reducing the requirement in the number of LUTs from two to one for certain logic expressions elaborated in Chap. 3. The LUTs present in SLICEL can implement any arbitrary combinational logic, whereas the LUTs in SLICEM can be implemented as a synchronous RAM resource called a distributed RAM element. The carry chain represents the fast carry propagation logic and the LUTs in the slice can be optionally connected to the carry chain via dedicated routes to implement complex logic functionality [7]. The storage elements in a slice can be configured as either edge-triggered D-type FFs or level-sensitive latches. Each FF can be controlled using the control signals *set*, *reset*, *clock*, and *clock enable* signals.

2.3 Fabric Slice Architecture for Virtex-6 FPGAs

Virtex-6 slice architecture is quite similar to Virtex-5 slice architecture, other than the fact that it offers four additional storage elements in every slice in comparison to Virtex-5 to facilitate more efficient pipelining and improved routing. However, every storage element has one control signal less, i.e., it does not have independent set and reset pins, as compared to Virtex-5 architecture. The slice architecture for Virtex-6 FPGAs is shown in Fig. 2.3. Other than that, it supports a higher bandwidth with

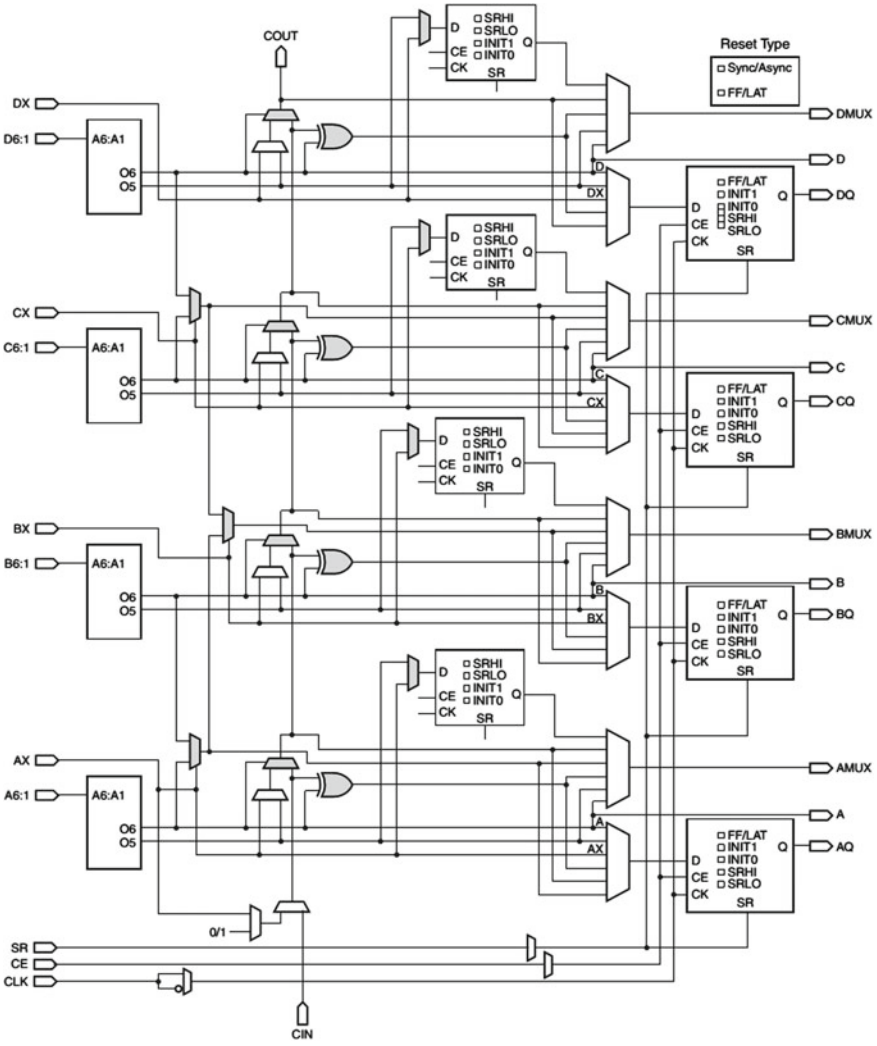

Fig. 2.3 Xilinx Virtex-6 slice architecture [8]

greater number of serial transceivers that can deliver at a higher Gbps rate, along with a faster global clocking with lower skew, improved jitter and faster clock trees in comparison to Virtex-5 FPGA platforms.

2.4 DSP Slice Architecture for Virtex-5 and Virtex-6 FPGAs

DSP slices in FPGAs are typically designed for low power applications as it significantly avoids fabric routing, but provides a reasonable speed of operation. The Virtex-5 FPGA DSP48E slice, as shown in Fig. 2.4, supports several independent arithmetic functionalities. Such functional units include a 25×18 two's complement multiplier, multiply accumulate (MACC) unit, multiply adder, three-input adder, barrel shifter, wide-bus multiplexer, magnitude comparator, bitwise logic functions, pattern detector, and wide counter. The slice has internal pipeline stages which must be used for achieving maximum performance up to 550 MHz.

For Virtex-6 FPGAs, the DSP slice available, DSP48E1, has all the features of a Virtex-5 FPGA DSP48E slice with certain additional features [10]. When all pipeline stages are used, Virtex-6 DSP slices can achieve a 600 MHz speed of operation. It supports an additional 25-bit pre-adder and register with another additional control unit.

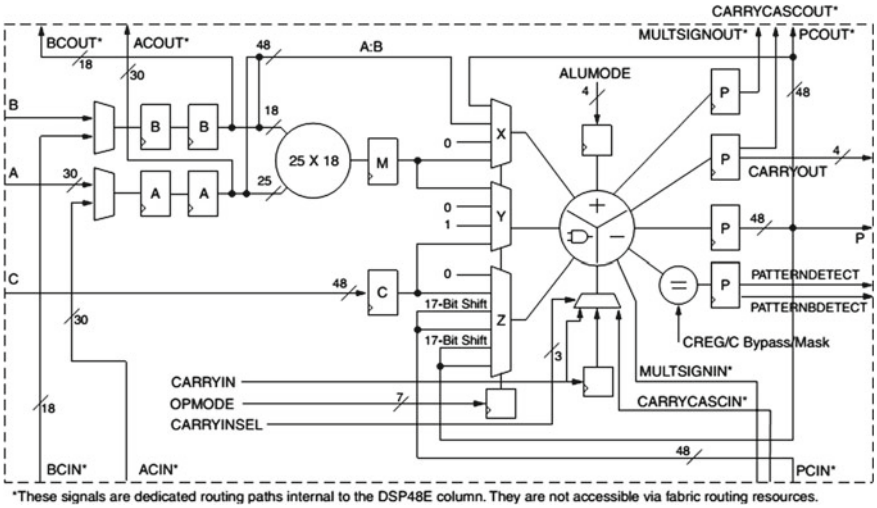

Fig. 2.4 Xilinx Virtex-5 DSP48E slice [6]

2.5 Implementation Overview

Native platform-dependent primitives such as 6-input LUT and carry chain can be directly instantiated in the HDL circuit description, and they appear unchanged in the final implementation mapped on the FPGA. For efficient and high performance design, the designer must ensure maximum utilization of the logic elements within each slice, and place the logically related slices into adjacent locations.

Designs targeted toward Virtex-5 and Virtex-6 FPGAs can also exploit IP cores available specifically for various arithmetic and logic functions. The Xilinx *LogiCORE*® hard IP provides two modes of implementation: *Fabric* and *XtremeDSP*®. Fabric implementation involves utilization of LUTs, FFs, multiplexers and carry chains, whereas *XtremeDSP* implementation involves utilization of the special *DSP* slice which can also be instantiated as a primitive. DSP slice-based implementations guarantee lower power consumption in comparison to fabric logic, but as we would find, cannot always match the performance achievable by circuits implemented using constrained placement of Xilinx fabric logic.

Most of our performance results have been reported using the Virtex-5 FPGA as the implementation platform. However, for certain circuits, like the cellular automata-based pseudorandom binary sequence generator or the matrix multiplication circuit, where it is necessary to register the dual outputs of the LUTs, Virtex-6 have been chosen as the implementation platform to ensure a compact implementation and pack more registers into a single slice, thereby freeing up resources that would otherwise have spanned across multiple slices and depleted adjacent slices of their register resources.

The circuits described in Chap. 4–7 were implemented either on a Xilinx Virtex-5 FPGA, device family XC5VLX330T, package FF1738 and speed grade -2 or Xilinx Virtex-6 FPGA, device family XC6VLX550T, package FF1760 and speed grade -2 using the Xilinx ISE 12.4 design environment. The speed of operation, resource utilization, and power–delay product (PDP) of the architectures have been compared with those reported in existing literature (if any) and with different modes of implementation have been tabulated. The designs have been evaluated in terms of speed, resource consumption in terms of FFs, LUTs, slices and DSP hard macros (whenever applicable), and power–delay product (PDP). Power–delay product has been calculated as the product of the power dissipation (sum of clock, logic, signal, and DSP power dissipation), the (minimum) clock-period (toggle rate of 12.5 %), and the latency (in terms of the number of clock cycles required to complete the computation).

Functions implemented using the DSP slices consume less power than those implemented in general FPGA fabric [6], and this would be evident from the results. However, it would also be evident that the proposed methodology based on fabric logic, combined with careful and constrained placement can outperform the DSP slice-based design with respect to speed. To achieve maximum performance using the DSP slices, it is desirable to use all the pipeline stages within the DSP slice.

2.6 Summary

In this chapter, we have introduced the modern and advanced families of Xilinx FPGAs that provide immense logic integration facilities and architectural support for high-performance implementations. The slice architectures for Virtex-5 and Virtex-6 FPGA families were described. An overview of the different modes of implementation were presented.

The next chapter will address a fabric component-based approach for realization of arithmetic circuits on modern FPGA families, where certain guidelines for manipulation and decomposition of Boolean logic level equations describing the implemented circuits will be discussed so that they can be easily and efficiently mapped to the physical fabric logic primitives of the target FPGA platform. Such an approach also allows the designer to predict the overall hardware cost and ensure a careful and compact placement of the logic architectures on the FPGA fabric. Certain examples of useful and practical circuits have also been described to illustrate the application of such guidelines for logic design.

References

1. Cosoroaba, A., Rivoallon, F.: Xilinx Inc., White Paper: Virtex-5 Family of FPGAs, Achieving Higher System Performance with the Virtex-5 Family of FPGAs, WP245 (v1.1.1). http://www.xilinx.com/support/documentation/white_papers/wp245.pdf. Cited 7 July 2006
2. Mehta, N.: Xilinx Inc., Xilinx 7 series FPGAs: the logical advantage, WP405 (v1.0). http://www.xilinx.com/support/documentation/white_papers/wp405-7Series-Logical-Advantage.pdf. Cited 6 Mar 2012
3. Xilinx Inc.: Spartan-6 family overview, DS160 (v2.0). http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf. Cited 25 Oct 2011
4. Xilinx Inc.: Virtex-5 family overview, DS100 (v5.0). http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. Cited 6 Feb 2009
5. Xilinx Inc.: Virtex-5 FPGA user guide, UG190 (v5.4). http://www.xilinx.com/support/documentation/user_guides/ug190.pdf. Cited 16 Mar 2012
6. Xilinx Inc.: Virtex-5 FPGA xtremeDSP design considerations user guide, UG193 (v3.5). http://www.xilinx.com/support/documentation/user_guides/ug193.pdf. Cited 26 Jan 2012
7. Xilinx Inc.: Virtex-5 libraries guide for HDL designs, UG621 (v11.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex5_hdl.pdf. Cited 6 Sep 2009
8. Xilinx Inc.: Virtex-6 FPGA configurable logic block, UG364 (v1.2). http://www.xilinx.com/support/documentation/user_guides/ug364.pdf. Cited 3 Feb 2012
9. Xilinx Inc.: Virtex-6 family overview, DS150 (v2.4). http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. Cited 19 Jan 2012
10. Xilinx Inc.: Virtex-6 FPGA DSP48E1 slice user guide, UG369 (v1.3). http://www.xilinx.com/support/documentation/user_guides/ug369.pdf. Cited 14 Feb 2011

Chapter 3
A Fabric Component Based Design Approach for High-Performance Integer Arithmetic Circuits

Abstract This chapter elaborates on some useful guidelines that can be helpful for compact and high-performance realization of circuits on modern high-end FPGAs from Xilinx. It involves manipulation of the Boolean equations *a priori* in the HDL circuit descriptions to forms that can be optimally mapped to the native target architecture by the CAD software. Although the guidelines are relatively simple, they are extremely useful in the efficient realization of numerous arithmetic circuits which can be constructed using the "bit-sliced" design paradigm.

3.1 Introduction

Implementation of highly optimized arithmetic circuits targeted toward a specific family of high-end FPGAs continue to remain a challenging problem. This is because the architecture of many fast arithmetic circuits that have been proposed over the decades may not be amenable to a much optimized implementation for a selected FPGA. Also, often the logic synthesis CAD software tools are unable to infer the desired native building-block components from the given input HDL specification of the circuit, as they explore only a small design space close to the input architectural description [5]. In addition, the logic synthesis algorithms used internally by the CAD tools are unable to apply the logic identities and perform appropriate algebraic factoring and subexpression sharing in many cases, especially when intermediate signals are tapped out [3] or registered to facilitate pipelining of the architecture.

It is in general a nontrivial computational problem to decompose the Boolean equations describing the implemented circuit, to forms such that the resultant subexpressions can be mapped easily and efficiently to the physical primitives of the fabric logic on the target FPGA. It helps if the designer manipulates the Boolean equations *a priori* in the HDL circuit descriptions, to forms that can be optimally mapped to the native target architecture by the CAD software.

The rest of the chapter is organized as follows. In Sect. 3.2, we discuss an existing work that estimates the hardware cost to map a Boolean function of x variables using k-input LUTs. We also discuss certain limitations behind the philosophy of making

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_3

such an estimate. In Sect. 3.3, we elaborate on certain important guidelines for high-performance realization of circuits on modern high-end FPGAs from Xilinx. We conclude in Sect. 3.4.

3.2 Existing Work

A previous work [4] had reported area requirements (for LUT-based FPGAs) in terms of the total number of k-input LUTs required to map a function of x variables, as

$$lut(x) = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 & \text{if } 1 < x \leq k \\ \lfloor \frac{x-k}{k-1} \rfloor + 2 & \text{if } x > k \text{ and } (k-1) \not| (x-k) \\ \frac{x-k}{k-1} + 1 & \text{if } x > k \text{ and } (k-1) \mid (x-k) \end{cases} \tag{3.1}$$

The above estimates of LUT requirements are based on the fact that LUTs are perhaps the most important and viable option for implementing combinational logic in FPGAs. Combinational logic blocks with higher number of inputs are expected to be implemented by a cascade of LUTs, with permitted amount of parallel processing, along the signal propagation path. However, for hardware-efficient implementations, the designer must explore the additional logical capabilities that the LUTs of modern day FPGAs provide. In addition, there are other hardware primitives available in the target FPGA platform to reduce the LUT requirement, such as the wide function multiplexers and the carry chains which must be considered for the purpose of implementation. The above closed form expression in (3.1) for estimating hardware resource requirements, therefore, has the following limitations:

- It assumes that all LUTs provide single outputs, whereas modern FPGAs from Xilinx provides dual-output LUTs that can significantly reduce hardware cost, provided the logic functions to be mapped satisfy certain criteria.
- It must be remembered that certain logic expressions can be factored appropriately to form subexpressions that can be realized using a combination of hardware primitives such as LUTs, wide function multiplexers and carry chains, thereby providing multiple outputs out of a single slice. The closed form expressions in (3.1) possibly hint at an approximate upper bound on the number of LUTs required.
- The number of LUTs occupied is not an accurate estimate of the area requirements, it is the number of slices spanned by the logic elements which give an accurate estimate of the total area required for logic realization.

The philosophy behind estimating the hardware resource requirement in terms of the number of LUTs used [4] may not reflect its actual implementation on hardware. For example, let us consider the Boolean logic functionality of an $8:1$ multiplexer, which is essentially an 11-input 1-output combinational logic function.

$$f(s_2, s_1, s_0, a, b, c, d, e, f, g, h) = s_2' s_1' s_0' a + s_2' s_1' s_0 b + s_2' s_1 s_0' c + s_2' s_1 s_0 d + s_2 s_1' s_0' e + s_2 s_1' s_0 f + s_2 s_1 s_0' g + s_2 s_1 s_0 h \quad (3.2)$$

Going by (3.1), $lut(x) = 2$ for $k = 6$. This information indicates that the first six variables go as input to the first LUT and the remaining five variables along with the output of the first LUT go as input to the second LUT. However on examining (3.2) closely, it can be observed that there is no possible way to decompose it to the following form comprising of two functions f_1 and f_2 which could have actually realized it using two LUTs:

$$f(s_2, s_1, s_0, a, b, c, d, e, f, g, h) = f_2 \underbrace{(f_1 \overbrace{(x_1, x_2, x_3, x_4, x_5, x_6)}^{\text{implemented using 1 LUT}}, x_7, x_8, x_9, x_{10}, x_{11})}_{\text{implemented using 1 LUT}}$$

where x_i can be any one of the variables of the function f.

3.3 Guidelines for High-Performance Realization

We list certain elementary but useful observations below, which should act as important guidelines for compact and high-performance realization of circuits on modern high-end FPGAs from Xilinx. These particular forms of the Boolean functions were chosen because they are relevant in the optimal realization of several arithmetic functions of interest on the Virtex-5 and Virtex-6 platform, as demonstrated later in this work.

1. A six-input LUT can implement any arbitrary combinational logic function f, having a maximum of six inputs and a single output.

$$y = f(x_1, \ldots, x_n) \quad where \quad 2 \leq n \leq 6 \quad (3.3)$$

2. A six-input LUT can implement any arbitrary five (or less)-input two-output function where each of the single-output functions may or may not have shared inputs. For example, consider two functions g and h, where

$$g = f(x_1, \ldots, x_n) \quad with \quad X = \{x_1, \ldots, x_n\}, \quad (3.4)$$
$$h = f(y_1, \ldots, y_m) \quad with \quad Y = \{y_1, \ldots, y_m\} \quad (3.5)$$

Here, the sets X and Y are called the *support* [2] of the functions g and h. For packing g and h into a single LUT, any one of the conditions must be satisfied:

- $4 \leq |X| + |Y| \leq 5$; if $X \cap Y = \emptyset$ (i.e., g and h are *orthogonal*)
- $2 \leq |X| + |Y| \leq 10$; if $X \cap Y \neq \emptyset$

where $|X|$ and $|Y|$ are the cardinality of the sets X and Y.

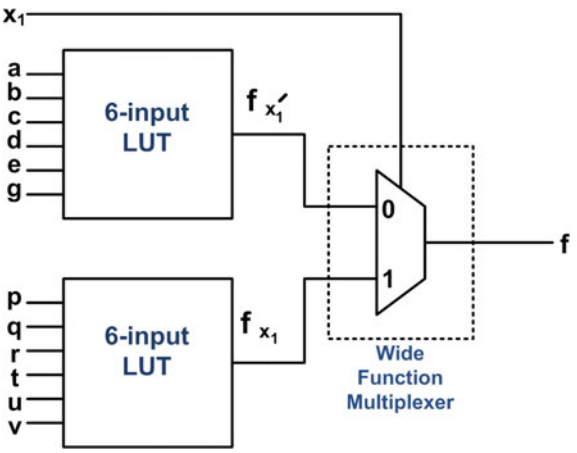

Fig. 3.1 Architecture mapping for Boolean logic that can be decomposed with respect to a single variable

3. Let f be a Boolean function of n variables ($8 \leq n \leq 13$) which can be represented in the following form (see Fig. 3.1):

$$f(i_1, i_2, \ldots, i_n) = x_1' f_{x_1'} + x_1 f_{x_1} \tag{3.6}$$

- Here, f_{x_1} and $f_{x_1'}$ are each six (or less)-input combinational functions that can be individually realized using one LUT each.
- The wide function multiplexer present in the same slice as that of the LUTs computes the final expression, as shown in Fig. 3.1.
- Equation (3.1) however evaluates to $lut(x) = 3$, where $x = x_{max} = 13$ (6×2 (two six-input LUTs) + 1 (select line)) and $k = 6$.
- If there exist p functions of the form as in f, the design requires $\lceil p/2 \rceil$ slices, and $2p$ LUTs.
- An $8:1$ multiplexer can be realized using this logic where the 6-input LUTs of Fig. 3.1 are configured as $4:1$ multiplexers each (sharing the same select lines), and the wide function multiplexer selecting one of the LUT outputs.

4. Let f be a function of n variables ($17 \leq n \leq 26$) such that we can apply recursive decomposition twice on it as shown below:

$$\begin{aligned} f(i_1, i_2, \ldots, i_n) &= x_1' f_{x_1'} + x_1 f_{x_1} \\ &= x_1'(x_2' f_{x_1' x_2'} + x_2 f_{x_1' x_2}) + x_1(x_3' f_{x_1 x_3'} + x_3 f_{x_1 x_3}) \\ &= x_1' x_2' f_{x_1' x_2'} + x_1' x_2 f_{x_1' x_2} + x_1 x_3' f_{x_1 x_3'} + x_1 x_3 f_{x_1 x_3} \end{aligned} \tag{3.7}$$

- Here, $f_{x_1' x_2'}$, $f_{x_1' x_2}$, $f_{x_1 x_3'}$ and $f_{x_1 x_3}$ are each 6 (or less)-input combinational functions that can individually be realized using one LUT each.
- Three wide function multiplexers present in the same slice as that of the LUTs computes the final expression as shown in Fig. 3.2.

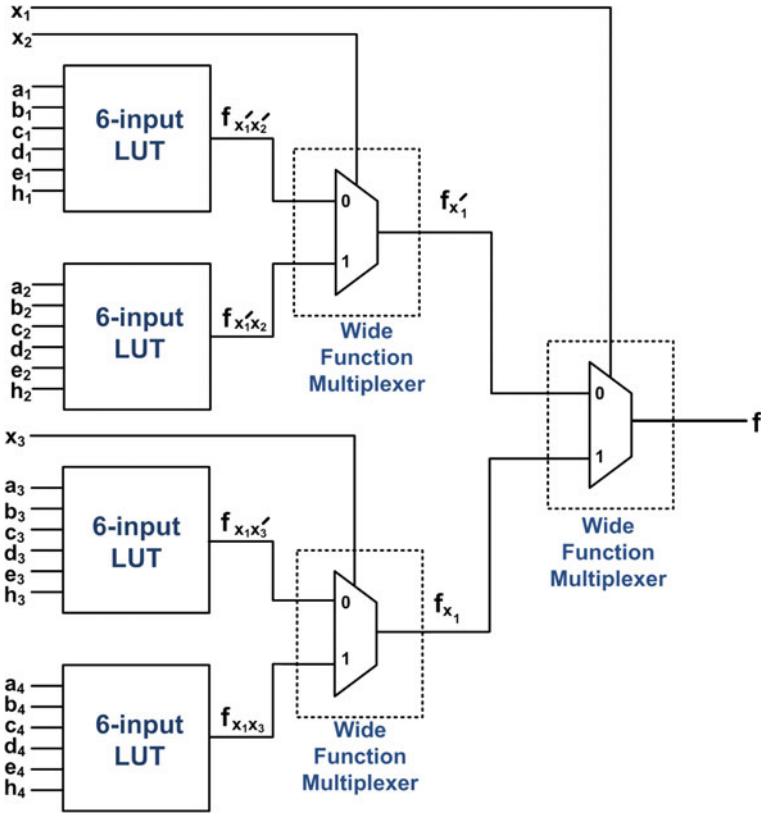

Fig. 3.2 Architecture mapping for Boolean logic that can be decomposed with respect to two variables

- Equation (3.1) however evaluates to $lut(x) = 6$, where $x = x_{max} = 27$ (6×4 (four six-input LUTs) + 3 (select lines)) and $k = 6$.
- If there exists p functions of the form as in f, the design requires p slices and $4p$ LUTs.
- A 16 : 1 multiplexer can thus be mapped in a single slice using four LUTs, and three wide function multiplexers.

5. Consider any expression R of the following form:

$$R = a'b + a[X] \tag{3.8}$$
$$= a'b + a[c'd + c(Y)] \tag{3.9}$$
$$= a'b + a[c'd + c(e'f + e\{Z\})] \tag{3.10}$$
$$= a'b + a[c'd + c(e'f + e\{g'h + gi\})] \tag{3.11}$$

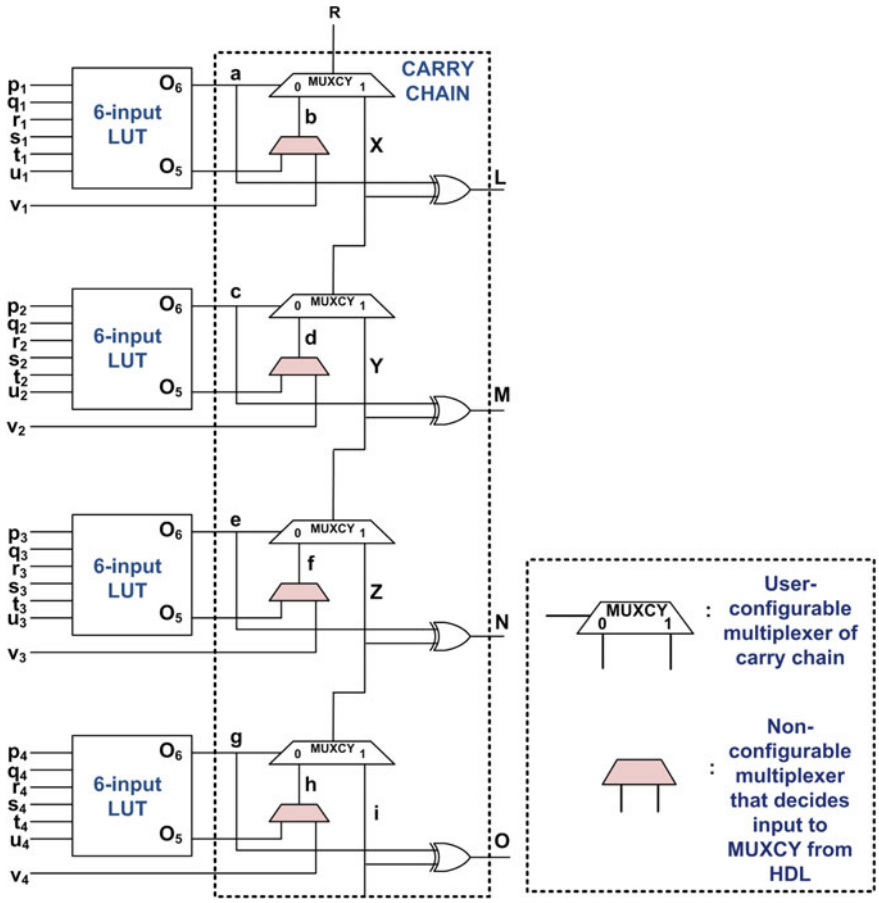

Fig. 3.3 Architecture mapping for Boolean logic that can exploit the carry chain

where $X = c'd + cY$, $Y = e'f + eZ$ and $Z = g'h + gi$. Here i is a single input variable, and for every member of the pair (a, b), (c, d), (e, f), and (g, h), there can be either of the following possibilities:

a. Both the members of the pair can individually be a maximum five (or less)-input function.
b. First member can be a six (or less)-input function and second member can be a single variable function.

The above expression can be realized using four LUTs and a single carry chain, thereby occupying only a single slice in a modern high-end Xilinx FPGA, as shown in Fig. 3.3.

- The Boolean logic equation (3.11) essentially represents a cascade of 2:1 multiplexer functions.

- Here, R can have a maximum of 29 (6×4 (four six-input LUTs) + 4 inputs $v_{4:1}$ external to the logic slice + 1 external input to the bottom MUXCY of the carry chain) input variables.
- If the Boolean logic function to be realized is of the form such that the variable i in (3.11) can be substituted by another expression bearing a similar resemblance to (3.11), and in such a way, if a total of n such substitutions can be carried out only at the position of variable i, then the entire expression can be realized using $(n + 1)$ slices and a maximum of $4(n + 1)$ LUTs.
- From (3.1), we obtain $lut(x) = 6$, where $x = 29$ and $k = 6$ for which a minimum FPGA area of two slices are required. However, with the help of carry chain fabric, the entire architecture can be compacted within a single slice.
- A wide input AND and OR gate can be realized by the function R.
- Example of a wide (24-input) AND gate where the logic equation can be manipulated to fit the form of (3.11) as shown below:

$$\begin{aligned} R &= a_1a_2a_3a_4a_5a_6a_7\cdots a_{18}a_{19}a_{20}a_{21}a_{22}a_{23}a_{24} \\ &= \overline{a_{19}\cdots a_{24}} \cdot 0 + (a_{19}\cdots a_{24})[a_{18}a_{17}\cdots a_1a_0] \\ &= \overline{a_{19}\cdots a_{24}} \cdot 0 + (a_{19}\cdots a_{24})[\overline{a_{13}\cdots a_{18}}.0 + (a_{13}\cdots a_{18})[a_{12}\cdots a_0]] \\ &= \overline{a_{19}\cdots a_{24}} \cdot 0 + (a_{19}\cdots a_{24})[\overline{a_{13}\cdots a_{18}} \cdot 0 + (a_{13}\cdots a_{18}) \\ &\quad [\overline{a_7\cdots a_{12}} \cdot 0 + (a_7\cdots a_{12})[\overline{a_1\cdots a_6} \cdot 0 + (a_1\cdots a_6) \cdot 1)]]] \end{aligned} \tag{3.12}$$

  Thus, going by Fig. 3.3, $a = a_{19}a_{20}a_{21}a_{22}a_{23}a_{24}$, $c = a_{13}a_{14}a_{15}a_{16}a_{17}a_{18}$, $e = a_7a_8a_9a_{10}a_{11}a_{12}$ and $g = a_1a_2a_3a_4a_5a_6$, $b = d = f = h = 0$, and $i = 1$. Hence each 6-input LUT realizes a 6-input AND gate and the outputs of the 6-input LUTs are AND-ed using the carry chain.
- Example of a wide (24-input) OR gate where the logic equation can be manipulated to fit the form of (3.11) as shown below:

$$\begin{aligned} R &= a_1 + a_2 + a_3 + a_4 + \cdots + a_{21} + a_{22} + a_{23} + a_{24} \\ &= (a_{19} + \cdots + a_{24}) \cdot 1 + (\overline{a_{19} + \cdots + a_{24}})[a_{18} + \cdots + a_1] \\ &= (a_{19} + \cdots + a_{24}) \cdot 1 + (\overline{a_{19} + \cdots + a_{24}})[(a_{13} + \cdots + a_{18}) \cdot 1 \\ &\quad + (\overline{a_{13} + \cdots + a_{18}})[a_{12} + \cdots + a_1]] \\ &= (a_{19} + \cdots + a_{24}) \cdot 1 + (\overline{a_{19} + \cdots + a_{24}})[(a_{13} + \cdots + a_{18}) \cdot 1 \\ &\quad + (\overline{a_{13} + \cdots + a_{18}})[(a_7 + \cdots + a_{12}).1 + (\overline{a_7 + \cdots + a_{12}}) \\ &\quad [(a_1 + \cdots + a_6) \cdot 1 + (\overline{a_1 + \cdots + a_6}) \cdot 0)]]] \end{aligned} \tag{3.13}$$

  Thus, going by Fig. 3.3, $a = \overline{a_{19} + a_{20} + a_{21} + a_{22} + a_{23} + a_{24}}$, $c = \overline{a_{13} + a_{14} + a_{15} + a_{16} + a_{17} + a_{18}}$, $e = \overline{a_7 + a_8 + a_9 + a_{10} + a_{11} + a_{12}}$ and $g = \overline{a_1 + a_2 + a_3 + a_4 + a_5 + a_6}$, $b = d = f = h = 1$, and $i = 0$. Hence each 6-input LUT realizes a 6-input NOR gate and the outputs of the 6-input LUTs are fed to the carry chain and a wide input OR gate is realized by following the absorption law $a + \overline{a}b = a + b$.

As illustrative examples of application of the preceding observations, we describe certain practical circuits where a wide input AND and OR gate are necessary for realization.

- Consider the design of a *priority encoder* which arbitrates among N units that are all requesting access to a shared resource. Access is to be granted to a single unit with highest priority where the least significant bit of the input corresponds to the highest priority. The corresponding logic equations can be described as

$$\begin{aligned} Y_1 &= N_1 \\ Y_2 &= N_2 \cdot \overline{N_1} \\ Y_i &= \underbrace{N_i \cdot \overline{N_{i-1}} \cdot \overline{N_{i-2}} \cdot \cdots \cdot \overline{N_2} \cdot \overline{N_1}}_{\text{wide input AND gate}} \end{aligned}$$

- Architecture of an *incrementer* that adds 1 to an input word N can be described by the following logical equations:

$$\begin{aligned} Y_0 &= \overline{N_0} \\ Y_1 &= N_1 \oplus N_0 \\ Y_i &= N_i \oplus \underbrace{(N_{i-1} \cdot N_{i-2} \cdot \cdots \cdot N_1 \cdot N_0)}_{\text{wide input AND gate}} \end{aligned}$$

- Architecture of a *decrementer* that subtracts 1 from an input word N can be described by the following logical equations:

$$\begin{aligned} Y_0 &= \overline{N_0} \\ Y_1 &= N_1 \odot N_0 \\ Y_i &= N_i \odot \underbrace{(N_{i-1} + N_{i-2} + \cdots + N_1 + N_0)}_{\text{wide input OR gate}} \end{aligned}$$

- $K = A + B$ Comparator [1]
  To design a circuit to detect $A + B = K$, the usual approach is to design an adder that adds inputs A and B, and feed the sum and input K to an equality comparator. However to significantly reduce hardware and computational overhead, a methodology was proposed in [1]. The key observation is the fact that if A and B are known, the carry into each bit to make $K = A + B$ can be determined. Thus, it is sufficient to check adjacent pairs of bits to verify that the carry-out produced by the previous bit and the carry-in required by the current bit are both same. The truth Table 3.1 shows the *required* and *generated* carries.
  The required carry-in cr_{i-1} for bit i and the generated carry-out cp_{i-1} for bit $i-1$ are obtained as follows:

Table 3.1 Required and generated carries [6]

A_i	B_i	K_i	cr_{i-1} (required)	cp_i (produced)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

$$cr_{i-1} = A_i \oplus B_i \oplus K_i \tag{3.14}$$

$$cp_{i-1} = (A_{i-1} \oplus B_{i-1})\overline{K_{i-1}} + A_{i-1} \cdot B_{i-1} \tag{3.15}$$

Equality check for the i-th bit position is performed using a single LUT as it can be computed using six distinct variables-A_i, B_i, K_i, A_{i-1}, B_{i-1} and K_{i-1}.

$$EQ_i = cr_{i-1} \odot cp_{i-1} \tag{3.16}$$

Final equality check is done using carry chain where all the outputs corresponding to equality checks at every bit position are AND-ed together.

$$EQ = \underbrace{EQ_j \cdot EQ_{j-1} \cdot \cdots \cdot EQ_i \cdot \cdots \cdot EQ_1 \cdot EQ_0}_{\text{wide input AND gate}} \tag{3.17}$$

- Additionally, we can obtain the following outputs from the XOR gates of the carry chain: $L = a \oplus X$, $M = c \oplus Y$, $N = e \oplus Z$ and $O = g \oplus i$.
  - The XOR gates of the carry chain can compute the sum bits of an adder.
  - *Propagate* function p_i is computed using LUT; $p_i = a_i \oplus b_i$.
  - i-th sum bit is computed by XORCY gate of carry chain; $s_i = p_i \oplus c_i$.
  - Carry-out bit of each stage is computed using MUXCY of carry chain; $c_{i+1} = \overline{p_i}a_i + p_i c_i$.
  - An n-bit adder can be realized using $\lceil n/4 \rceil$ slices and a maximum of n LUTs.

Additionally, LUTs of SLICEM can also be configured as shift registers. Each SLICEM LUT can be configured as a variable 1 to 32 clock cycle shift register [7] whose length can be fixed, static, or dynamically adjusted by controlling $A[4:0]$ as shown in Fig. 3.4. The LUT can be described as a $32:1$ multiplexer with the five inputs serving as binary select lines, and the values programmed into the LUT serving as the data being selected. Such LUTs can be cascaded with FFs and other LUTs of SLICEM to realize greater shift lengths. Presence of these special LUTs

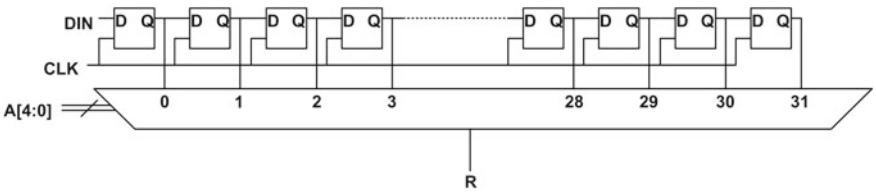

Fig. 3.4 Configuration of an LUT (of SLICEM) as a shift register

reduces FPGA resource utilization compared to implementations using FFs only. Since each SLICEM in a Virtex-5 FPGA contains four LUTs and four FFs, a 1 to 132 clock cycle shift register can be realized in a single slice; and a 1 to 136 clock cycle register can be realized in a single slice for Virtex-6 FPGAs as it contains four additional FFs. For realization of an n-clock cycle shift register, we require $\lceil n/132 \rceil$ slices, with a maximum of $4\lceil n/132 \rceil$ LUTs, and $4\lceil n/132 \rceil$ FFs for Virtex-5 FPGA platform, and $\lceil n/136 \rceil$ slices, with a maximum of $4\lceil n/136 \rceil$ LUTs and $8\lceil n/136 \rceil$ FFs for Virtex-6 FPGA platform. Shift registers are implemented typically for Linear Feedback Shift Register (LFSR) circuits.

3.4 Summary

All the above-mentioned guidelines are critical for consideration of efficient mapping of Boolean logic equations on target FPGA fabric. Such guidelines have manifold applications in realization of numerous arithmetic circuits which can be constructed using bit-sliced design paradigm. In the next few chapters, we will show such arithmetic datapath and controlpath circuits where we will be frequently referring to these guidelines to explain the architectures. We will discuss the pipelined implementations of adders, fast *carry lookahead* logic for adder circuits, absolute difference circuits, multipliers, squarers, universal shift comparators, counters and pseudorandom sequence generators, and validate our design philosophy by tabulating the superior performance improvements that has been achieved in comparison to implementations carried out using other design philosophies. The next chapter discusses pipelined implementations of arithmetic datapath circuits.

References

1. Cortadella, J., Llabería, J.: Evaluation of $A + B = K$ conditions without carry propagation. IEEE Trans. Comput. **41**(11), 1484–1487 (1992)
2. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Kluwer Academic Publisher, Boston (1996)

3. Preuβer, T.B., Spallek, R.G.: Mapping basic prefix computations to fast carry–chain structures. In: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), pp. 604–608 (2009)
4. Roy, S.S., Rebeiro, C., Mukhopadhyay, D.: Theoretical modeling of the Itoh-tsujii inversion algorithm for enhanced performance on k–LUT based FPGAs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1–6 (2011)
5. Verma, A.K., Brisk, P., Ienne, J.P.: Challenges in automatic optimization of arithmetic circuits. In: Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH), pp. 213–218 (2009)
6. Weste, N.H.E., Harris, D., Banerjee, A.: CMOS VLSI Design: A Circuits and Systems Perspective, 3rd edn. Pearson (2011)

Chapter 4
Architecture of Datapath Circuits

Abstract This chapter discusses some common arithmetic datapath circuits which can significantly contribute to the critical path delay, either due to their long, cascading path delay, or undesirable inference of logic elements and their irregular placement on the Xilinx fabric logic. We present pipelined implementations of arithmetic datapath circuits, which when combined with their constrained and careful placement on the fabric logic, significantly improve their performance. Simultaneously, we present the associated mathematical analyses and proofs of correctness for the proposed architecture.

4.1 Introduction

We now describe the architectures of the arithmetic datapath circuits supported by the *FlexiCore* CAD tool. Currently, the CAD tool supports the following datapath circuits: **Integer Adder/Subtractor**, **Integer Absolute Difference Circuit**, **Combined Two's Complement and Unsigned Integer Multiplier**, **Combined Two's Complement and Unsigned Integer Squarer**, and **Universal Shift Register**. Among these circuits, the Xilinx Virtex-5-based optimized architectures for the adder and absolute difference circuits have been proposed previously [16, 24]—we have improved the performance by pipelining at appropriate locations and using constrained placement. The architecture of the remaining circuits as well as the Boolean algebraic justifications and formal proofs to justify all the architectures have been explained by us in detail. The design automation of all the circuits (including the ones previously proposed) have been discussed and implemented by us.

The rest of the chapter is organized as follows. In Sect. 4.2, we present the pipelined implementation of integer adders. In Sect. 4.3, we present the pipelined implementation of absolute difference circuits. Section 4.4 discusses the *hybrid, carry-save, pipelined* implementation of combined unsigned and two's complement integer multipliers. Section 4.5 presents the implementation of squarers using similar design philosophy proposed for multipliers. Universal Shift Register implementations have been proposed in Sect. 4.6. We conclude in Sect. 4.7.

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_4

4.2 Integer Adder/Subtractor Architecture

4.2.1 Hybrid Ripple Carry Adder (Hybrid RCA)

The basic building block for realization of a pipelined "hybrid ripple carry adder" (Hybrid RCA), using the carry chain, LUTs, and FFs available in a Xilinx slice, has been depicted in Fig. 4.1. The term "hybrid" signifies that pipelining has been done after every 4-bit addition. The outputs of the "XORCY" gates generate the sum bits, whereas the output of each MUXCY calculates the intermediate carries. Latches can be inserted on the carry propagation path for pipelining the design. The LUTs compute the *propagate* function $p_i = a_i \oplus b_i$. Let $g_i = a_i b_i$ be the corresponding *generate* function. The ith sum bit is then calculated by XOR-ing the LUT and MUXCY outputs as

$$s_i = p_i \oplus c_i = a_i \oplus b_i \oplus c_i \tag{4.1}$$

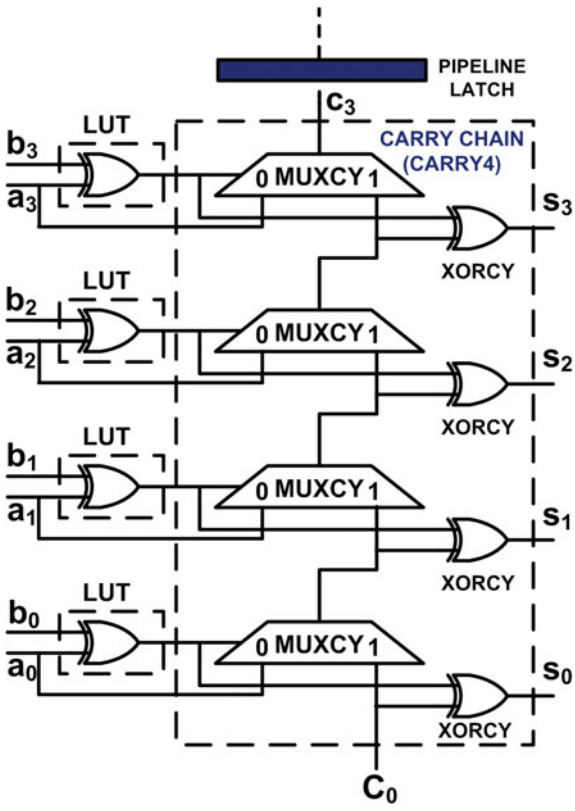

Fig. 4.1 Basic building bock for pipelined implementation of hybrid Ripple Carry Adder (RCA)

where c_i is the ith carry-in bit. The output of each MUXCY gate computes the ith carry-out as:

$$\begin{aligned} c_{oi} &= g_i + p_i c_i = a_i b_i + (a_i \oplus b_i) c_i \\ &= (\overline{a_i}\,\overline{b_i} + a_i b_i) a_i + (a_i \oplus b_i) c_i = \overline{p_i} a_i + p_i c_i \end{aligned} \tag{4.2}$$

If pipelined, this Hybrid RCA results in $\lceil n/4 \rceil - 1$ pipeline stages.

Staging registers to align the inputs and outputs can be realized through instantiation of the SRLC32E primitive, which is a 1-to-32 clock cycle shift register implemented using a single SLICEM LUT [22]. Since the priority for our designs is speed, we implement an n-bit delay by configuring the LUTs to function as an $(n-1)$-bit shift register. It also uses a FF available in the same slice of the LUT to realize the last unit of delay, as depicted in Fig. 4.2, that simultaneously pipelines the architecture without any overall increase in latency. Similar implementation of staging delays by interchanging the SLICEM LUT and FF position (for maximum speed) is valid for aligning the output bits. However, additional staging registers can be reduced drastically for a system implementation by adjusting the structure that uses them, and hence staging registers have been kept optional.

The extension of the functionality of the adder is relatively straightforward: the adder can also be configured as a subtractor using a mode control input M, which

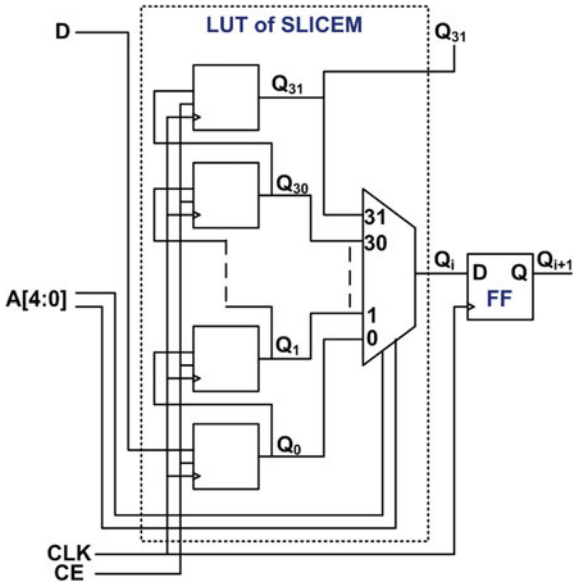

Fig. 4.2 Staging delay implementation through shift registers and FFs

is set to 0 for addition and 1 for subtraction. The modified sum and carry output equations become:

$$s_i = a_i \oplus (b_i \oplus M) \oplus c_i \tag{4.3}$$

$$c_{oi} = a_i(b_i \oplus M) + (a_i \oplus b_i \oplus M)c_i \tag{4.4}$$

The LUTs in Fig. 4.1 now compute the *modified propagate* function as $p_{m_i} = a_i \oplus b_i \oplus M$. The carry-out to the next successive stage is determined by the following (4.5) with the initial carry set to M:

$$c_{oi} = a_i(b_i \oplus M) + (a_i \oplus b_i \oplus M)c_i = \overline{p_{m_i}}a_i + p_{m_i}c_i \tag{4.5}$$

4.2.2 Xilinx DSP Slice-Based Adder

As mentioned previously, adders can also be realized using generic configurable logic like embedded DSP48E slices. Each slice can accept operands of width 48 bits. To realize larger adders with higher operand width $n(>48)$, we require $\lceil n/48 \rceil$ DSP48E slices, and such designs can be pipelined by activating the pipeline registers internal to the slices to provide full speed operation. Thus, in such cases, the Xilinx Synthesis Tool (*XST*) does not report the number of pipeline registers consumed which are internal to the DSP slice. An $n(>48)$-bit pipelined DSP slice-based adder requires $\lceil n/48 \rceil - 1$ pipeline stages. In order to perform the addition operation, the attributes "ALUMODE" and "OPMODE" have to be set to "0000" and "0001111", respectively [21], whereas for subtraction operation, the attributes "ALUMODE" and "OPMODE" have to be set to "0011" and "0110011", respectively [21]. ALUMODE and OPMODE are special control signals of the DSP slice. ALUMODE controls the selection of the logic function in the DSP48E slice, whereas, OPMODE controls the inputs being fed to the logic units present inside the DSP slice. Figure 4.3 illustrates the DSP adder architecture.

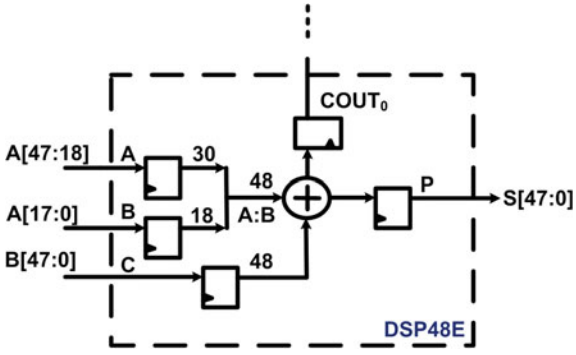

Fig. 4.3 Xilinx DSP slice-based adder [21]

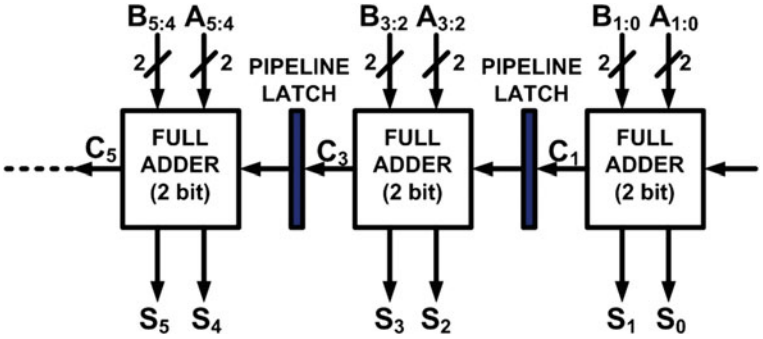

Fig. 4.4 Architecture for *FloPoCo* generated adder

4.2.3 FloPoCo-Based Adder

The behavior of *FloPoCo*, when asked to generate behavioral VHDL code for adders targeted toward Virtex-5 platform implementation, and same frequency of operation as obtained through our approach of constrained placement, is interesting. From the generated VHDL code, it was observed that the circuit description generated is pipelined, with each stage performing 2-bit addition, as shown in Fig. 4.4. This in itself proves that the architecture cannot enjoy the benefit of utilizing a complete length-4 carry chain natively available in the Virtex-5 family to realize the architecture. As expected, when the generated code from *FloPoCo* is synthesized using Xilinx *XST*, the synthesis tool is unable to map each 2-bit adder to its fast carry chain fabric, thereby resulting in a complete LUT-based implementation, and is usually unable to satisfy the frequency constraints.

4.2.4 Fast Carry Adder Using Carry-Lookahead Mechanism

The novel adder proposed in [24] had been designed using carry-lookahead mechanism by splitting an n-bit adder into two independent, identical portions L-RCA and H-RCA, each of which calculates $n/2$ sum bits (assuming n to be even). The H-RCA receives its carry input from a *fast carry generator* circuit. Both the L-RCA and H-RCA are architecturally identical to the pipelined implementation of the Hybrid RCA shown in Fig. 4.1. The architecture of the fast adder architecture proposed in [24] for 64-bit operands is shown in Fig. 4.5. The reformulation of the carry chain computation [12] has been addressed in the fast carry generator as follows:

Let $P_{i:j}$ and $G_{i:j}$ denote the *group-propagated carry* and the *group-generated carry* functions, respectively, for a group of bit positions $i, i-1, \ldots j$ (with $i \geq j$). $P_{i:j}$ equals 1 when an incoming carry into the least significant position j, c_j, is allowed to propagate through all $i - j + 1$ bit positions. $G_{i:j}$ equals 1 when a carry

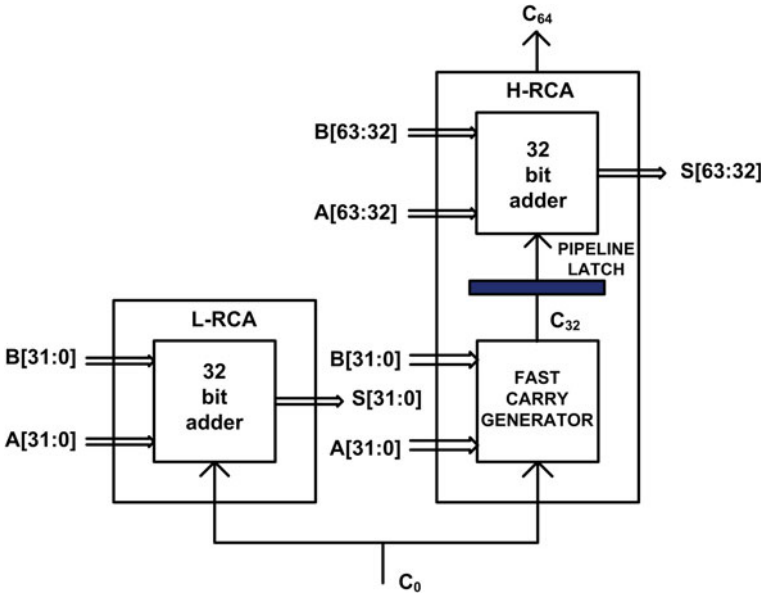

Fig. 4.5 Fast adder architecture proposed in [24]

is generated in at least one of the bit positions from j to i (both inclusive), and propagates to bit position $i+1$, i.e., the outgoing carry c_{i+1}=1 [2].

$$P_{i:j} = \begin{cases} P_i, & \text{if } i = j. \\ P_i P_{i-1:j} & \text{if } i \geq j. \end{cases} \tag{4.6}$$

$$G_{i:j} = \begin{cases} G_i, & \text{if } i = j. \\ G_i + P_i G_{i-1:j} & \text{if } i \geq j. \end{cases} \tag{4.7}$$

where $P_i = a_i \oplus b_i$ and $G_i = a_i b_i$.

The recursive equaions (4.6)–(4.7) can be further generalized to

$$P_{i:j} = P_{i:m} P_{m-1:j} \tag{4.8}$$

$$G_{i:j} = G_{i:m} + P_{i:m} G_{m-1:j} \tag{4.9}$$

where $i \geq m \geq j+1$.

For the mth bit position, $i \geq m \geq j$, we have

$$c_m = G_{m-1:j} + P_{m-1:j} c_j \tag{4.10}$$

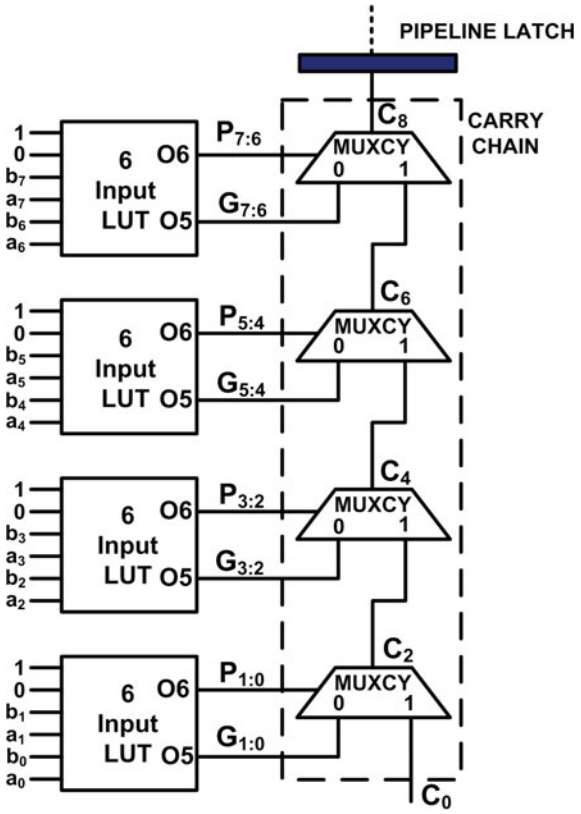

Fig. 4.6 Architecture for fast carry generator [24]

In the architecture depicted in Fig. 4.6, the logic functions $G_{i:j}$ and $P_{i:j}$ are calculated using the 6-input LUTs where $i = j + 1$, and $m = i + 1 = j + 2$ and c_m is calculated using the carry chain. Thus,

$$P_{i:j} = P_i P_j = (a_i \oplus b_i)(a_j \oplus b_j) \tag{4.11}$$

$$\begin{aligned} G_{i:j} &= G_i + P_i G_{i-1:j} = a_i b_i + (a_i \oplus b_i) a_j b_j \\ c_m &= G_{m-1:j} + P_{m-1:j} c_j = G_{m-1:m-2} + P_{m-1:m-2} c_{m-2} \\ &= G_{i:j} + P_{i:j} c_{m-2} = \overline{P_{i:j}} G_{i:j} + P_{i:j} c_{m-2} \end{aligned} \tag{4.12}$$

Hence, c_m can obtained from c_{m-2} using only a single multiplexer of the carry chain in the fast carry generator, which is in contrast to the standard ripple carry adder

chain that computes c_m from c_{m-2} using two multiplexers of the carry chain. Hence, c_8 can be obtained from c_0 using carry-lookahead mechanism within a single slice which is shown as follows:

$$\begin{aligned} c_8 &= \overline{P_{7:6}}G_{7:6} + P_{7:6}c_6 \\ &= \overline{P_{7:6}}G_{7:6} + P_{7:6}[\overline{P_{5:4}}G_{5:4} + P_{5:4}c_4] \\ &= \overline{P_{7:6}}G_{7:6} + P_{7:6}[\overline{P_{5:4}}G_{5:4} + P_{5:4}[\overline{P_{3:2}}G_{3:2} + P_{3:2}c_2]] \\ &= \overline{P_{7:6}}G_{7:6} + P_{7:6}[\overline{P_{5:4}}G_{5:4} + P_{5:4}[\overline{P_{3:2}}G_{3:2} + P_{3:2}[\overline{P_{1:0}}G_{1:0} + P_{1:0}c_0]]] \end{aligned} \tag{4.13}$$

Thus (4.13) assumes the same form as (3.11) and can be compacted into a single slice.

Equations (4.6)–(4.7) can be modified to perform a subtraction operation by introducing the mode control input M where M is set to 0 for addition and 1 for subtraction. The mode control input goes as an additional input to the dual output LUTs and the input carry of the carry chain in the L-RCA and the fast carry generator architecture with no increase in hardware. In this case, the equations are re-modified as $P_i = a_i \oplus b_i \oplus M$ and $G_i = a_i(b_i \oplus M)$. Thus,

$$P_{i:j} = P_i P_j = (a_i \oplus b_i \oplus M)(a_j \oplus b_j \oplus M) \tag{4.14}$$

$$\begin{aligned} G_{i:j} &= G_i + P_i G_{i-1:j} = a_i(b_i \oplus M) + (a_i \oplus b_i \oplus M)a_j(b_j \oplus M) \\ &= a_i(\overline{a_i \oplus b_i \oplus M}) + a_j(\overline{a_j \oplus b_j \oplus M})(a_i \oplus b_i \oplus M) \end{aligned} \tag{4.15}$$

$$\begin{aligned} c_m &= G_{i:j} + P_{i:j}c_{m-2} \\ &= a_i(\overline{a_i \oplus b_i \oplus M}) + a_j(\overline{a_j \oplus b_j \oplus M})(a_i \oplus b_i \oplus M) + P_{i:j}c_{m-2} \\ &= a_i(\overline{a_i \oplus b_i \oplus M}) + a_i(\overline{a_i \oplus b_i \oplus M})(\overline{a_j \oplus b_j \oplus M}) \\ &\quad + a_j(\overline{a_j \oplus b_j \oplus M})(a_i \oplus b_i \oplus M) + P_{i:j}c_{m-2} \\ &= \left[(a_i \oplus b_i \oplus M) + (\overline{a_j \oplus b_j \oplus M})\right][a_i(\overline{a_i \oplus b_i \oplus M}) \\ &\quad + a_j(\overline{a_j \oplus b_j \oplus M})(a_i \oplus b_i \oplus M)] + P_{i:j}c_{m-2} \\ &= \overline{P_{i:j}}G_{i:j} + P_{i:j}c_{m-2} \end{aligned} \tag{4.16}$$

The pipeline latency of the adder on a whole depends on the number of pipeline stages for the H-RCA and the fast carry generator. The $n/2$-bit H-RCA requires $\lceil n/8 \rceil - 1$ pipeline stages, while the $n/2$-bit fast carry generator requires $\lceil n/16 \rceil - 1$ pipeline stages. Overall, an n-bit fast carry adder requires $\lceil 3n/16 \rceil - 1$ pipeline stages, including the pipeline stage between the fast carry generator and the H-RCA.

4.2.5 Adder Implementation Results

The adder circuit has been compared for five design styles—the fast carry generator-based adder [24], Hybrid RCA, FPGA fabric-based adder automatically generated by the GUI utility of IP Core Generator in ISE, DSP slice-based adder, and the *FloPoCo* generated adder.

The dashed entries in Table 4.1 indicate that the equivalent results are either not applicable or not reported in [24]. Note that our designs were pipelined, with the number of pipeline stages as described in previous subsections. Although not explicitly mentioned in [24], the authors informed us through personal correspondence that they inserted register banks in the fast carry adder only at the input and output ports of the circuit to measure the frequency of the circuit.

In case of constrained placement, for adder designs upto 64 bits, the hybrid RCA outperforms the fast carry generator-based adder design both in terms of speed and area, but when it comes to realizing adders with higher operand widths (e.g., 96 and 128), which results in an increased vertical column height of slices, the fast carry generator-based adder design gives marginally better speed performance at the cost of extra hardware. The fast carry generator-based adder also has lower latency than the Hybrid RCA. The important trend to note here is that in all cases, constrained placement adders give substantially better performance than the corresponding unconstrained placed adders of the same operand width. The FPGA fabric adder generated using the GUI utility results in the worst performance, whereas the DSP adders consume only DSP48E slices and operates at a reasonable speed. *FloPoCo* adders have been generated by giving the same frequency of operation (as constraints) which was achieved for the Pipelined Hybrid RCA on Virtex-5 platform with constrained placement. It was however observed that the *FloPoCo* adders generated require double latency in comparison to the proposed circuits with increase in hardware and PDP and is unable to satisfy the frequency constraints.

The compact implementation made possible through our design methodology is not achievable for Xilinx IP Core-based designs for any latency or for *FloPoCo* implementation with any frequency as obtained through our approach. In spite of the available length-4 carry chain functionality, the two above-mentioned CAD tools fail to register the carry chain output using a FF available in the same slice of the carry chain, thereby resulting in an unoptimized inference of logic elements. This can be observed from the Floorplan Editor of the Xilinx ISE software, where the carry signal is made to enter the global routing network to be routed to another slice to use its flip–flop as a pipeline latch, and then route it to another slice to restart the carry chain. This results in a random, haphazard floorplanning, whereby the logic elements are arbitrarily placed into any available vacant slice. A possible reason might be that a Virtex-5 logic slice does not support dedicated hardware for registering those signals, and the CAD tool fails to infer the desired implementation.

With respect to power consumption, it is clearly evident from the results in Table 4.1, that for the same latency, fabric implementation of adders using Xilinx IP Core realizes a circuit which has a higher PDP than the Hybrid RCA. The reasons

Table 4.1 Integer adder implementation results

Operand width	Design style	Design with unconstrained placement							Pipelined design with constrained placement[b]						
		Freq (MHz)	Latency #clk cycles	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
32	Fabric adder (IP core)	378.36	7	0.67	98	116	0	44	–	–	–	–	–	–	–
	DSP slice adder	550.00	2	0.08	0	0	1	0	–	–	–	–	–	–	–
	FloPoCo adder[c]	714.29	16	1.56	138	133	0	51	–	–	–	–	–	–	–
	Fast carry adder[a] [24]	521.00	–	–	98	41	0	–	**808.41**	**5**	**0.18**	**9**	**40**	**0**	**10**
	Hybrid RCA	–	–	–	–	–	–	–	**809.06**	**7**	**0.26**	**8**	**32**	**0**	**8**
48	Fabric adder (IP core)	348.43	11	1.47	157	191	0	78	–	–	–	–	–	–	–
	DSP slice adder	550.00	2	0.10	0	0	1	0	–	–	–	–	–	–	–
	FloPoCo adder[c]	664.45	24	3.66	210	205	0	112	–	–	–	–	–	–	–
	Fast carry adder[a] [24]	472.00	–	–	146	61	0	–	**789.27**	**8**	**0.53**	**14**	**60**	**0**	**15**
	Hybrid RCA	–	–	–	–	–	–	–	**806.45**	**11**	**0.76**	**12**	**48**	**0**	**12**

(continued)

Table 4.1 (continued)

Operand width	Design style	Design with unconstrained placement							Pipelined design with constrained placement[b]						
		Freq (MHz)	Latency #clk cycles	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
64	Fabric adder (IP core)	468.60	15	2.48	217	263	0	117	–	–	–	–	–	–	–
	DSP slice adder	500.00	2	0.15	0	0	2	0	–	–	–	–	–	–	–
	FloPoCo adder[c]	595.59	32	6.43	282	277	0	166	–	–	–	–	–	–	–
	Fast carry adder[a] [24]	397.00	–	–	194	81	0	–	**788.02**	**11**	**1.08**	**19**	**80**	**0**	**20**
	Hybrid RCA	–	–	–	–	–	–	–	**806.45**	**15**	**1.32**	**16**	**64**	**0**	**16**
96	Fabric adder (IP core)	413.22	23	6.05	337	407	0	182	–	–	–	–	–	–	–
	DSP slice adder	500.00	2	0.17	0	0	2	0	–	–		–	–	–	–
	FloPoCo adder[c]	471.48	48	15.05	426	507	0	208	–	–	–	–	–	–	–
	Fast carry adder[a] [24]	303.00	–	–	290	121	0	–	**787.40**	**17**	**2.45**	**29**	**120**	**0**	**30**
	Hybrid RCA	–	–	–	–	–	–	–	**754.15**	**23**	**3.30**	**24**	**96**	**0**	**24**

(continued)

Table 4.1 (continued)

Operand width	Design style	Design with unconstrained placement							Pipelined design with constrained placement[b]						
		Freq (MHz)	Latency #clk cycles	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
128	Fabric adder (IP core)	414.94	31	11.80	457	551	0	271	–	–	–	–	–	–	–
	DSP slice adder	500.00	3	0.35	0	0	3	0	–	–	–	–	–	–	–
	FloPoCo adder[c]	522.19	64	25.91	570	747	0	305	–	–	–	–	–	–	–
	Fast carry adder[a] [24]	–	–	–	–	–	–	–	**787.40**	**23**	**4.42**	**39**	**160**	**0**	**40**
	Hybrid RCA	–	–	–	–	–	–	–	**760.46**	**31**	**8.16**	**32**	**128**	**0**	**32**

[a]In the fast carry adder reported in [24], register banks were inserted only at the input and output of the circuit, as revealed by the authors through personal correspondence

[b]In our designs with constrained placement, an n-bit fast carry adder had $\lceil 3n/16 \rceil - 1$ pipeline stages, while an n-bit Hybrid RCA had $\lceil n/4 \rceil - 1$ pipeline stages. For the DSP slice-based adder, no pipelining was done if $n < 48$, while if $n > 48$, number of pipeline stages internal to the DSP slices were $\lceil n/48 \rceil - 1$, and no FF resource consumption was reported by the Xilinx synthesis tool

[c]*FloPoCo* adders have been generated by giving the same frequency of operation (as constraints) which was achieved for the fast carry adder on Virtex-5 platform

attributed to this observation are the inefficient placement and routing of the adder circuit, leading to higher delay and greater on-chip power dissipation. The fast carry adder has a lower PDP due to shorter latency and lesser column height of slices mapped on to the FPGA fabric, though the amount of hardware involved is more in comparison to Hybrid RCA implementation. *FloPoCo* adders have higher PDP due to higher latency and inefficient packing of logic into the FPGA fabric. The DSP slice-based mode of implementation consumes the least power and has the lowest latency, but cannot outperform the proposed architectures in terms of speed.

4.3 Absolute Difference Circuit Architecture

Absolute difference operation is required for certain image and video processing algorithms, as well as for some arithmetic operations like finding out the Greatest Common Divisor (GCD) of two numbers. For this circuit, we would assume that both the input operands are unsigned integers.

4.3.1 Proposed Absolute Difference Circuit

The authors in [16] had proposed an architecture which is based on the compact implementation on the FPGA fabric of the following expression:

$$AD = \mid A - B \mid = \begin{cases} (A - B), & \text{if } A - B \geq 0 \\ (B - A), & \text{otherwise} \end{cases} \tag{4.17}$$

If an n-bit less-than comparator generates a high signal if $A < B$, $B - A$ is computed, else $A - B$ is computed by the absolute difference circuit. Each LUT accepts 2-bit sub-words $A_{i:i-1}$ and $B_{i:i-1}$, each of which has no more than four distinct inputs, and outputs two signals $AeqB_{i:i-1}$ and $AlessB_{i:i-1}$. $AeqB_{i:i-1} = 1$ if $A_{i:i-1} = B_{i:i-1}$ and $AlessB_{i:i-1} = 1$ if $A_{i:i-1} < B_{i:i-1}$. $AeqB_{i:i-1}$ drives the select line of the multiplexer of the carry chain and $AlessB_{i:i-1}$ is an input to the multiplexer which is selected if $AeqB_{i:i-1} = 0$. If $x_i = A_i \odot B_i$ and $x_{i-1} = A_{i-1} \odot B_{i-1}$, then

$$AlessB_{i:i-1} = \overline{A_i} B_i + x_i \overline{A_{i-1}} B_{i-1} \tag{4.18}$$

$$AeqB_{i:i-1} = x_i x_{i-1} \tag{4.19}$$

The architecture for the module establishing if $A < B$ is shown in Fig. 4.7. The output of this module A_l_B decides upon the operation $A - B$ or $B - A$. For an n-bit less-than comparator, its output $A\_l\_B_n$ is obtained using the following recurrence relation:

$$A\_l\_B_n = \overline{AeqB_{n:n-1}} AlessB_{n:n-1} + AeqB_{n:n-1} A\_l\_B_{n-2} \tag{4.20}$$

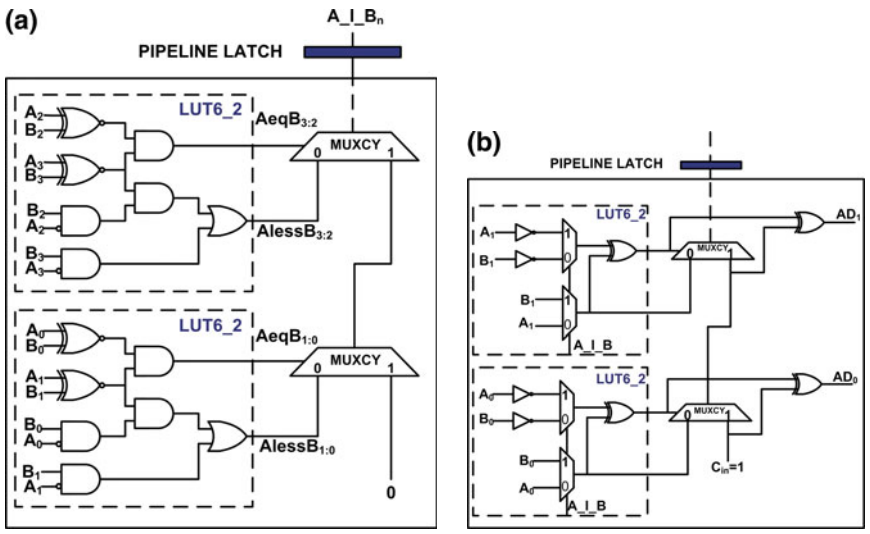

Fig. 4.7 Proposed absolute difference circuit [16]. **a** Module computing if $A < B$. **b** Module computing absolute difference

where the *base* condition is $A\_l\_B_0 = 0$. This recurrence relation bears exact resemblance to (3.11) making it an ideal candidate for carry chain implementation. When $A\_l\_B_n$=1, B+$\overline{A}$+1 is computed, else A+$\overline{B}$+1 is computed, as shown in Fig. 4.7b, where $\overline{A}$ and $\overline{B}$ are the 1's complement of A and B, respectively.

4.3.2 DSP Slice-Based Absolute Difference Circuit

For a DSP slice-based implementation, as shown in Fig. 4.8, there are two (or more) DSP slices, among which half the slices are configured as a subtractor ($C-(A:B)$), and half the slices as an adder ($C+(A:B)$) which acts in a two's complement mode. The sign bit of the subtractor unit of DSP slice is used as the multiplexer output to choose between the subtractor or the adder (two's complement) output. These 2:1 multiplexers and the NOT gates for realizing the two's complement outputs are implemented using fabric logic. However, to realize an absolute difference circuit for numbers with less than 25 bits, a single DSP slice can be implemented as an adder/logic unit in the *Single Instruction Multiple Data* (SIMD) Arithmetic mode [21], which can be used to implement small add–subtract functions, with high performance, less hardware, and lower power consumption. The adder-cum-logic unit when used in the SIMD mode has two 24-bit fields, with one field computing $IN1-IN2$ and the other field computing $IN2-IN1$. The sign bit of the first subtractor result is used as the multiplexer select, and the output of the multiplexer is the absolute difference value. The multiplexer is implemented in the fabric.

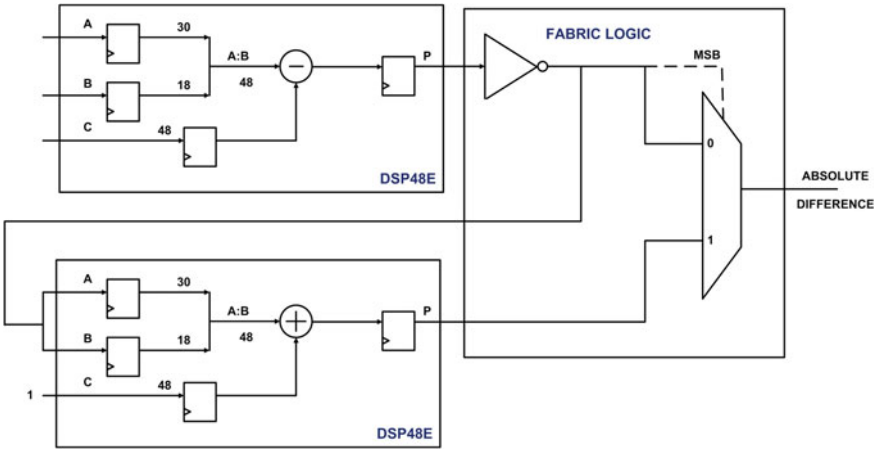

Fig. 4.8 Xilinx Virtex-5 DSP slice-based absolute difference circuit [21]

The pipelined implementation of the absolute difference circuit has been shown in Fig. 4.7a, b. The n-bit module establishing if $A < B$ requires $\lceil n/8 \rceil - 1$ pipeline stages, while the n-bit module computing the absolute difference circuit requires $\lceil n/4 \rceil - 1$ pipeline stages. Overall, an n-bit absolute difference circuit requires $\lceil 3n/8 \rceil - 1$ pipeline stages, including the pipeline stage between the two modules. The $n(>48)$-bit DSP slice implementation requires $\lceil n/48 \rceil - 1$ pipeline stages for each DSP slice that has either been configured as an adder or a subtractor.

4.3.3 FloPoCo-Based Absolute Difference Circuit

FloPoCo (as of v 2.5.0) does not generate a dedicated absolute difference circuit. However, it supports the functionality of a dual subtractor that takes X and Y as inputs and computes both $X - Y$ and $Y - X$. The MSB (sign bit) of either result is input to a 2:1 multiplexer and the absolute difference is obtained as depicted in Fig. 4.9. It can be observed from the generated VHDL code that *FloPoCo* is unable to pipeline a 32-bit dual subtractor, whereas it can pipeline higher operand width dual subtractors, though the pipelining is very irregular and unbalanced. For example, it pipelines a 96-bit dual subtractor into five stages of 48, 12, 12, 12, and 12 bits. Thus the pipelining philosophy adapted by *FloPoCo* for adders is completely different from that used for a dual subtractor, and because of adopting an unbalanced pipelining strategy (the motivation for which is difficult to understand), the user-specified target frequency is no longer met.

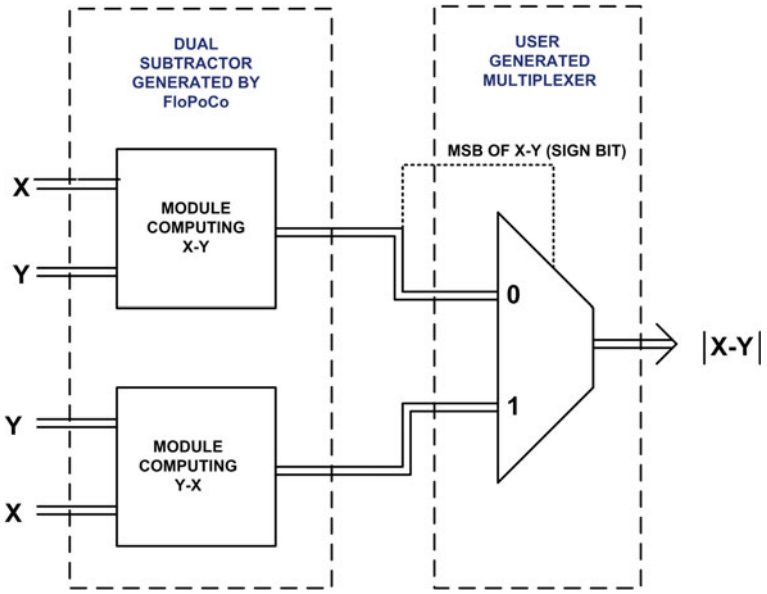

Fig. 4.9 Architecture for *FloPoCo*-based absolute difference circuit

4.3.4 Absolute Difference Circuit Implementation Results

The absolute difference circuit has been compared for three design styles—the proposed design, DSP slice-based implementation, and *FloPoCo*-based implementation. The proposed design of the absolute difference circuit outperforms the DSP mode of implementation in terms of speed. There is slight deterioration in performance with increase in operand width for the proposed design. This might be due to the complex routing of the final carry chain output from the module detecting whether $A < B$, to all the LUT inputs of the module computing the absolute difference. However, for the DSP mode of implementation, the speed degradation with increase of operand width is drastic. The DSP-based implementation requires additional fabric logic to realize the equivalent functionality, but gives a lower PDP value. *FloPoCo*-based implementation could not pipeline the 32-bit absolute difference circuit. The frequency of operation for the 32-bit circuit was determined by inserting registers at the inputs and outputs of the circuit. The higher bit-width realizations could not be effectively pipelined by *FloPoCo* to achieve the desired speed given as the input constraints (Table 4.2).

Table 4.2 Integer absolute difference circuit implementation results

Operand width	Design style	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
32	DSP slice implementation[a]	500.00	4	0.20	0	64	2	24
	FloPoCo-based implementation[b]	284.41	1	0.10	160	96	0	40
	Proposed design[a] [16]	**626.96**	**13**	**0.59**	**11**	**48**	**0**	**12**
48	DSP slice implementation[a]	500.00	4	0.25	0	96	2	42
	FloPoCo-based implementation[b]	542.59	2	0.35	108	239	0	110
	Proposed design[a] [16]	**624.22**	**17**	**1.64**	**17**	**72**	**0**	**18**
64	DSP slice implementation[a]	295.25	6	0.48	0	128	4	56
	FloPoCo-based implementation[b]	337.95	4	1.19	196	328	0	176
	Proposed design[a] [16]	**603.14**	**23**	**3.02**	**23**	**96**	**0**	**24**
96	DSP slice implementation[a]	278.16	6	0.70	0	192	4	87
	FloPoCo-based implementation[b]	407.83	4	1.60	320	486	0	230
	Proposed design[a] [16]	**558.97**	**35**	**4.98**	**35**	**144**	**0**	**36**
128	DSP slice implementation[a]	381.68	8	1.24	0	256	6	101
	FloPoCo-based implementation[b]	279.96	5	3.41	512	685	0	331
	Proposed design[a] [16]	**517.00**	**47**	**12.302**	**47**	**192**	**0**	**48**

[a] In the absolute difference circuit design with constrained placement, an n-bit circuit had $\lceil 3n/8 \rceil - 1$ pipeline stages. The authors in [16] had not pipelined the design and had presented results only for a 32-bit implementation. For the DSP slice-based design, no pipelining was done if $n < 48$, while if $n > 48$, the number of pipeline stages internal to the DSP slices were $\lceil n/48 \rceil - 1$, and no FF consumption was reported by the Xilinx synthesis tool

[b] *FloPoCo* can only generate a dual subtractor. User has to generate a multiplexer to decide upon the absolute difference value

4.4 Integer Multiplier Architecture

We now discuss the architectures for unsigned, two's complement and combined unsigned and two's complement multiplication schemes. The advantage in realizing a combined architecture lies in the fact that for an n-bit integer X, the same architecture can support the operation for the range of $-2^{n-1} \leq X \leq (2^{n-1} - 1)$ for two's complement integers, as well as the entire range of $0 \leq X \leq (2^n - 1)$ for unsigned integers. It is to be noted that the complete, detailed proofs for optimized unsigned and two's complement multiplication and squaring schemes are difficult to find in the existing literature, as the optimizations are based on Boolean algebraic properties, which although not always trivially provable, are often accepted by intuition. However, the detailed proofs provided by us give better insights into the effects of the optimization steps on the corresponding hardware architectures.

In [5], a proof for correctness of a basic binary adder was presented. Subsequently, an extension of this work was carried out by Feng et al. in [8] that presented formal proofs of sequential and parallel prefix adders. These works were primarily inspired by the fact that such complete formalism for adders are hard to find in any previously published literature on computer arithmetic. In the subsequent sections, we carry forward such analyses for integer multiplication and squaring operations.

4.4.1 Unsigned Integer Multiplier

Let two unsigned numbers of n-bit and m-bit, A (multiplicand) and B (multiplier) have face values $a_{n-1}a_{n-2}...a_1a_0$ and $b_{m-1}b_{m-2}...b_1b_0$, respectively, and give rise to a $(m+n)$-bit product $P = p_{m+n-1}p_{m+n-2}...p_1p_0$. The numerical values of A, B, and P are

$A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{j=0}^{m-1} b_j 2^j$ and $P = \sum_{k=0}^{m+n-1} p_k 2^k$.

Theorem 4.1 *The product of an n-bit unsigned integer multiplicand A with an m-bit unsigned integer multiplier B is given by*

$$P_{us} = \sum_{i=0}^{min(m,n)-1} a_i b_i 2^{2i} + \sum_{i=1}^{n-1} \sum_{j=0}^{min(i-1,m-1)} a_i b_j 2^{i+j} + \sum_{j=1}^{m-1} \sum_{i=0}^{min(j-1,n-1)} a_i b_j 2^{i+j}.$$

Proof: Consider any partial product (PP) $a_x b_y$. If $x = y$, we are considering the "diagonal elements" of the PP array, which contributes to the sum given by $P_{diag} = \sum_{i=0}^{min(m,n)-1} a_i b_i 2^{2i}$. Here $min(x, y)$ represents the minimum of two numbers x and y. If $x > y$, we are referring to the PPs situated above the diagonal elements, which contribute to the partial sum $P_{upper} = \sum_{i=1}^{n-1} \sum_{j=0}^{min(i-1,m-1)} a_i b_j 2^{i+j}$. Similarly,

for $x < y$, we are referring to the PPs situated below the diagonal elements, which contribute to the partial sum given by $P_{lower} = \sum_{j=1}^{m-1} \sum_{i=0}^{min(j-1,n-1)} a_i b_j 2^{i+j}$. Finally,

$$\begin{aligned} P_{us} &= P_{diag} + P_{upper} + P_{lower} \\ &= \sum_{i=0}^{min(m,n)-1} a_i b_i 2^{2i} + \sum_{i=1}^{n-1} \sum_{j=0}^{min(i-1,m-1)} a_i b_j 2^{i+j} + \sum_{j=1}^{m-1} \sum_{i=0}^{min(j-1,n-1)} a_i b_j 2^{i+j} \end{aligned} \quad (4.21)$$

For $m = n$, (4.21) becomes:

$$P_{us} = \sum_{i=0}^{n-1} a_i b_i 2^{2i} + \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} a_i b_j 2^{i+j} + \sum_{j=1}^{n-1} \sum_{i=0}^{j-1} a_i b_j 2^{i+j} \quad (4.22)$$

The unsigned multiplier in general has mn PPs, and n^2 PPs when $m = n$. Equation. (4.22) is represented by the PP array of Fig. 4.10 for 6×6 unsigned multiplier.

4.4.2 Two's Complement Multiplier

When two signed numbers represented in the two's complement system are to be multiplied, the complexity of the algorithm increases as the sign of the number is embedded in the number itself. An iterative array algorithm for multiplication of two's complement numbers was proposed in [3]. Consider two two's complement integers $A = a_{n-1}a_{n-2}\ldots a_1a_0$ and $B = b_{m-1}b_{m-2}\ldots b_1b_0$, where $A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$ and $B = -b_{m-1}2^{m-1} + \sum_{j=0}^{m-2} b_j 2^j$, and their product P_{tc} $(= -p_{m+n-1}2^{m+n-1} + \sum_{k=0}^{m+n-2} p_k 2^k)$ is represented as $p_{m+n-1}p_{m+n-2}\ldots p_1 p_0$.

We shall now prove the two's complement product P in the same light as suggested in [13] for the original *Baugh-Wooley Multiplication Algorithm* [3]. The main idea is

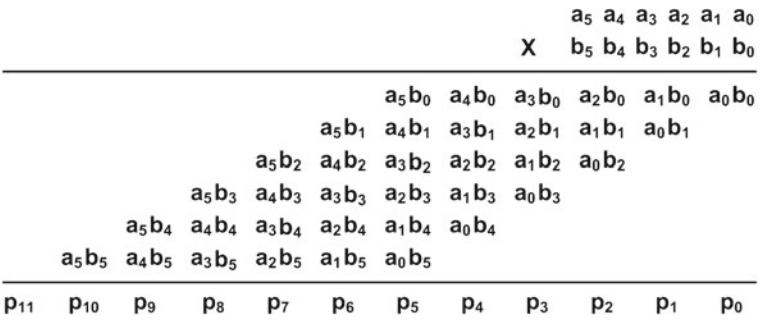

Fig. 4.10 A 6×6 unsigned array multiplication

to convert a two's complement multiplication to an equivalent parallel array addition problem in which all partial product bits are positive.

Theorem 4.2 *The product of an n-bit two's complement integer multiplicand A and an m-bit two's complement integer multiplier B is given by*

$$P_{tc} = -2^{m+n-1} + 2^{m-1} + 2^{n-1} + a_{n-1}b_{m-1}2^{m+n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} + 2^{n-1}\sum_{j=0}^{m-2}\overline{a_{n-1}b_j}2^j + 2^{m-1}\sum_{i=0}^{n-2}\overline{a_i b_{m-1}}2^i.$$

Proof:

$$\begin{aligned}
P_{tc} &= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right)\left(-b_{m-1}2^{m-1} + \sum_{j=0}^{m-2} b_j 2^j\right) \\
&= \left(a_{n-1}b_{m-1}2^{m+n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j}\right) - \left(2^{n-1}\sum_{j=0}^{m-2} a_{n-1}b_j 2^j + 2^{m-1}\sum_{i=0}^{n-2} a_i b_{m-1}2^i\right) \\
&= \left(a_{n-1}b_{m-1}2^{m+n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j}\right) - \left(2^{n-1}\sum_{j=0}^{m-2} 2^j - 2^{n-1}\sum_{j=0}^{m-2}\overline{a_{n-1}b_j}2^j\right) \\
&\quad -\left(2^{m-1}\sum_{i=0}^{n-2} 2^i - 2^{m-1}\sum_{i=0}^{n-2}\overline{a_i b_{m-1}}2^i\right) \\
&= -2^{m+n-1} + 2^{m-1} + 2^{n-1} + a_{n-1}b_{m-1}2^{m+n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} + 2^{n-1}\sum_{j=0}^{m-2}\overline{a_{n-1}b_j}2^j \\
&\quad +2^{m-1}\sum_{i=0}^{n-2}\overline{a_i b_{m-1}}2^i
\end{aligned} \tag{4.23}$$

Here, for any two bits p and q, the identities $pq = 1-\overline{pq}$ and $\sum_{j=0}^{m-2} 2^j = 2^{m-1} - 1$ have been used. For $m = n$, (4.23) becomes:

$$\begin{aligned}
P_{tc} &= -2^{2n-1} + 2^n + a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\
&\quad +2^{n-1}\left(\sum_{j=0}^{n-2}\overline{a_{n-1}b_j}2^j + \sum_{i=0}^{n-2}\overline{a_i b_{m-1}}2^i\right)
\end{aligned} \tag{4.24}$$

Equation (4.24) is represented by the PP array of Fig. 4.11 for 6 × 6 modified *Baugh–Wooley Multiplier* [12, 15]. The two's complement multiplier has $(mn + 2)$ PPs if $m \neq n$, and $(n^2 + 2)$ PPs when $m = n$. The PP count is inclusive of the '1's present at the 2^n and 2^{2n-1} positions.

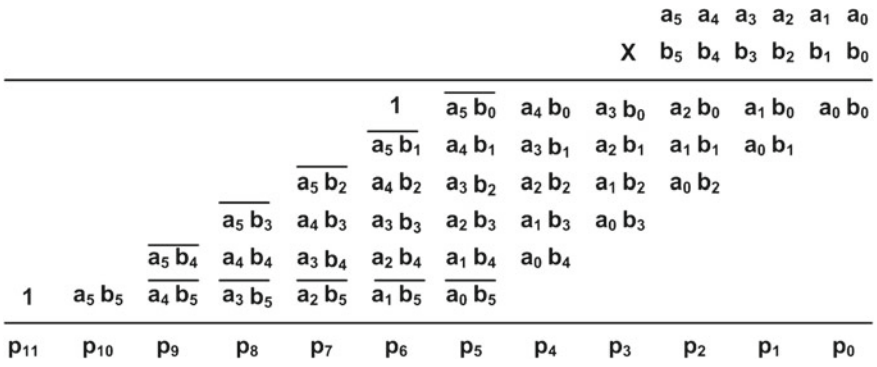

Fig. 4.11 A 6 × 6 two's complement array multiplication

4.4.3 Combined Unsigned and Two's Complement Multiplier

The advantage in realizing a combined architecture lies in the fact that for an n-bit integer X, the same architecture can support the operation for the range of $-2^{n-1} \leq X \leq (2^{n-1} - 1)$ for two's complement integers, as well as the entire range of $0 \leq X \leq (2^n - 1)$ for unsigned integers. From Figs. 4.10 and 4.11, it can be observed that the trend of partial product generation is same both for unsigned and two's complement multipliers. We introduce a control signal t which determines the mode of operation of the multiplier. If $t = 0$, the circuit functions as an unsigned multiplier, while for $t = 1$, the circuit behaves as a two's complement multiplier. This combined multiplier is shown in Fig. 4.12, where $\widehat{a_i b_j} = a_i b_j \oplus t$. On close observation of (4.21) and (4.23), the product P_c for a combined multiplier can be written as:

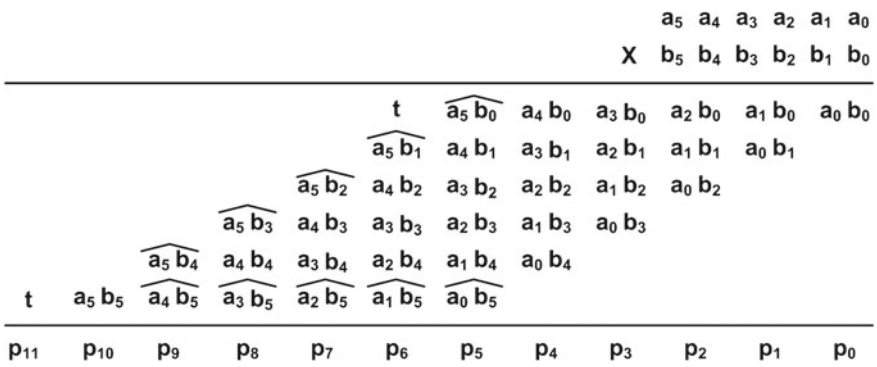

Fig. 4.12 A 6 × 6 combined array multiplication

$$P_c = -2^{m+n-1}t + 2^{m-1}t + 2^{n-1}t + a_{n-1}b_{m-1}2^{m+n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} + 2^{n-1}\sum_{j=0}^{m-2}\widehat{a_{n-1}b_j}2^j + 2^{m-1}\sum_{i=0}^{n-2}\widehat{a_i b_{m-1}}2^i \tag{4.25}$$

For $m = n$, (4.25) can be written as:

$$P_c = -2^{2n-1}t + 2^n t + a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + 2^{n-1}\left(\sum_{j=0}^{n-2}\widehat{a_{n-1}b_j}2^j + \sum_{i=0}^{n-2}\widehat{a_i b_{n-1}}2^i\right) \tag{4.26}$$

The combined multiplier thus has $(mn + 2)$ PPs when $m \neq n$, and $(n^2 + 2)$ PPs when $m = n$. The PP count is inclusive of the control signal t present at the 2^n and 2^{2n-1} positions.

Multiplication operation involves generation of partial products and computing the intermediate sum and carries. The entire multiplier can be envisaged to be composed of identical sub-circuits, each capable of generating new partial products and computing the partial sum and carries for every stage. Since each slice contains four FFs, these intermediate partial sum and carry bits can be latched without consuming extra slices, and given as input to the next stage, thereby achieving a *carry-save* pipelined architecture. In such a situation, every carry chain can be configured as a 3-bit ripple carry adder, producing a 3-bit sum and 1-bit carry as shown in Fig. 4.14. Such hybrid multipliers are termed as *carry-save, pipelined, iterative array multipliers*. It should be noted that although in more advanced Xilinx FPGA families such as Virtex-6/Virtex-7/Spartan-6 [18, 20, 23], each slice contains eight FFs, none of these extra four FFs can be utilized to latch the final carry signal propagating from the carry chain of the same slice, as no hardwired connection is present between the final carry signal and the four extra FFs. Hence, we cannot achieve a more compact architecture using Virtex-6, Virtex-7, or Spartan-6 FPGAs.

Addition of two PPs or a PP with a partial sum (denoted as X_i and Y_j) can be achieved by exploiting the dual output LUTs and the carry chain. The O_6 output of the LUT computes the *propagate* function $p_l = X_i \oplus Y_j$ as shown in Fig. 4.13. The O_5 output computes Y_j. Let $g_l = X_i \cdot Y_j$ be the *generate* function. The corresponding sum bit is then calculated by XOR-ing the LUT and MUXCY outputs as $s_r = p_l \oplus c_k = X_i \oplus Y_j \oplus c_k$, where c_k is the carry-in bit. Each MUXCY computes the carry-out as:

$$\begin{aligned} c_{oi} &= g_l + p_l \cdot c_k = X_i \cdot Y_j + (X_i \oplus Y_j) \cdot c_k \\ &= (\overline{X_i} \cdot \overline{Y_j} + X_i \cdot Y_j) \cdot X_i + (X_i \oplus Y_j) \cdot c_k = \overline{p_l} \cdot X_i + p_l \cdot c_k \end{aligned} \tag{4.27}$$

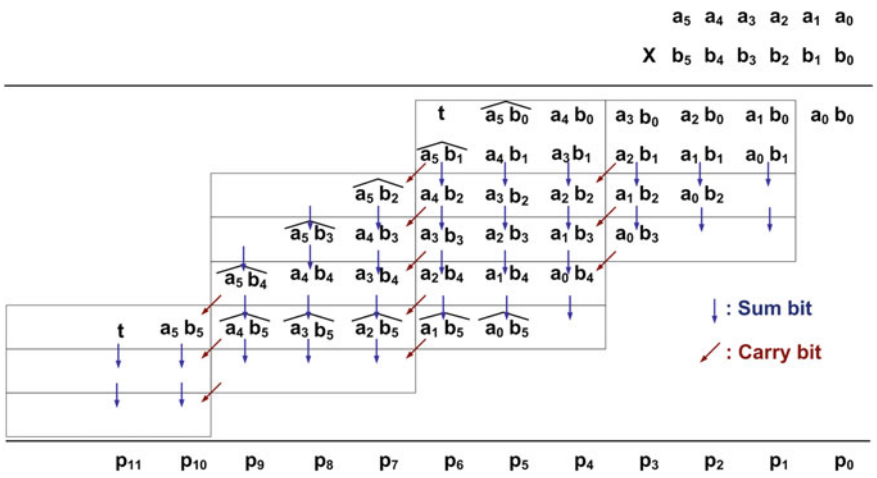

Fig. 4.13 Array multiplication depicting the grouping of partial products

An efficient implementation of carry-save addition in Spartan 3 FPGAs was reported in [10]. A similar, but however, modified design philosophy has also been extended for the carry save addition on Virtex-5 FPGAs. The additions at different bit positions in the same row are carried out in parallel as shown in Fig. 4.13. However, the last step, where the final partial sums and carries are to be added, can be tackled carefully by performing a *parallel bit-level carry-save pipelined addition*, so that the frequency bottleneck, which otherwise would have been imposed by a scheme such as a large ripple-carry *vector merging adder* [15], can be avoided. For an $A \times B$ multiplier, where each of A and B are n-bit wide, the number of pipeline stages can be calculated to be $(n-2)+\lfloor n/3 \rfloor$.

A pipelined implementation of the multiplier requires correct timing synchronization of the arrival of the partial products to the architecture. With the consideration that if A and B are the multiplicand and the multiplier, respectively, all the a_i bits must be available after every clock cycle, whereas the b_i bit is only required to arrive after $(i-1)$th clock delay. We perform certain hardware optimizations by delaying all the a_i bits using FFs, whereas the b_ith bit for $i \geq 2$ is delayed using shift registers [4] implemented using the SRLC32E primitive [22] which maps to a SLICEM LUT. Since the priority for our designs is speed, with a slight modification, we implement an n-bit delay for a multiplier bit by configuring the SLICEM LUTs to function as an $(n-1)$-bit shift register, and then using a FF available in the same slice to realize the last unit of delay, as depicted in Fig. 4.2. The advantage of this scheme is that the FF used simultaneously also serves to pipeline the architecture, without any overall increase in the latency of the circuit. A similar design concept was also spelt out for Xilinx Spartan-3 FPGAs in [19]. The output bits can be aligned following the same design philosophy, with the FF and SLICEM LUT positions interchanged for maximum speed. The addition of the partial products in the two topmost rows of the

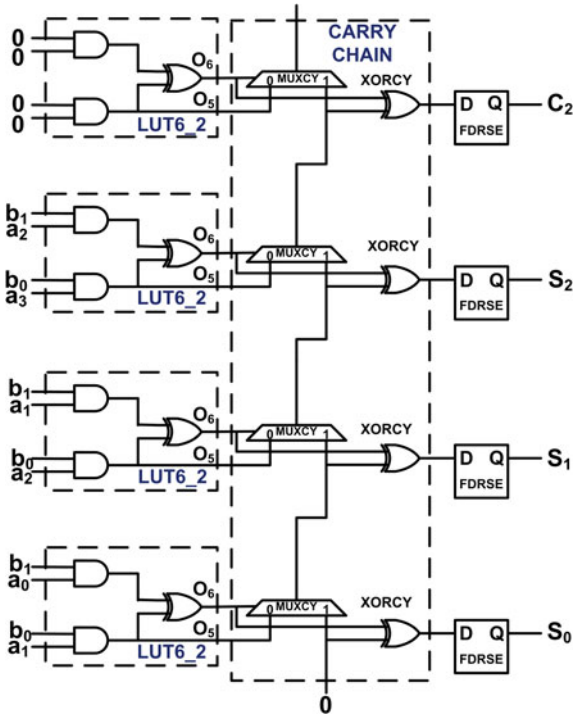

Fig. 4.14 Slice configuration for partial product addition in the first row of Hybrid Carry Save Multiplication

multiplication array (as shown in Fig. 4.13) can be achieved by using the dual output LUTs to compute the two partial products to be added and configuring the carry chain appropriately as depicted in Fig. 4.14 for addition of the partial products. Subsequent partial products can be added with the partial sum and partial carry generated in the previous row similarly.

From Fig. 4.13, it is evident that we have to realize underutilized adders (carry chains and LUTs), thereby resulting in increase of hardware cost. Since the newer FPGA families support huge fabric logic, this trade-off is acceptable to ensure higher throughput. However, it must be noted that the circuit complexity for multiplier has a strong dependence on operand width, as the number of partial products is a quadratic function of the operand width. In such cases, custom implementations demand higher hardware cost and greater PDP. The approach described so far is well suited to operands of smaller bit-width. Higher bit-width implementations can be realized using DSP slices, or a combination of DSP slices and fabric logic [1, 7] to save on hardware and to have superior PDP, with some compromise in speed (always lower than or equal to an operating frequency of 500 MHz for Virtex-5 FPGAs). However, since DSP slices are relatively very few in number in comparison to fabric logic slices, an alternative is to go for pure fabric logic-based implementation at highest speed, but at higher hardware overhead and higher PDP.

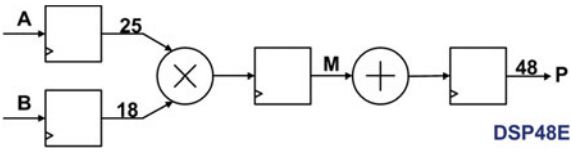

Fig. 4.15 Pipelined two's complement 25 × 18 DSP multiplier [21]

4.4.4 DSP Slice-Based Signed Multiplier

Each Xilinx Virtex-5 DSP slice can perform 25×18 two's complement multiplication as shown in Fig. 4.15. The DSP slices have been configured to operate at the fastest speed with a latency of three clock cycles. The necessary OPMODE and ALUMODE settings are "0000101" and "0000", respectively [21]. To perform the multiplication of two numbers M and N, it is important to treat M as multiplicand and N as multiplier if M has higher bit-width than N. Such a choice eventually leads to lesser hardware and pipeline stages.

4.4.5 FloPoCo-Based Signed Multiplier

FloPoCo can also generate integer multiplier descriptions. However, *FloPoCo* is unable to generate a pipelined multiplier in spite of the user specifying the target operating frequency, but generates an erroneous comment in the VHDL code that it has performed single-stage pipelining. This inability to pipeline has been reported by us to the authors of [6]. The authors have admitted through personal correspondence that a bug exists in the arithmetic core generation for integer multipliers, which they have filed and are trying to solve that issue before the next release.

4.4.6 Multiplier Implementation Results

The proposed custom implementation of multipliers result in excess hardware to achieve symmetric placement of logic blocks also leading to generation of partially redundant carry chains that are configured as adders. The DSP multipliers come at a lower hardware cost and lesser PDP. However, since DSP slices are much fewer in number compared to available fabric logic slices, the use of DSP slices must be done judiciously by ensuring as much complete utilization of each DSP slice configured. Thus, fabric logic configuration may be a feasible option that guarantees the maximum throughput for relatively smaller bit-width numbers. *FloPoCo* is however unable to generate pipelined integer multipliers and erroneously reports that it has achieved a single stage pipelining. In such a situation, registers were placed at appropriate locations to determine the operating frequency of the circuit (Table 4.3).

Table 4.3 Integer multiplier implementation results

Operand width	Design style	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
6 × 6	Fabric multiplier (IP core)	564.65	6	0.26	82	84	0	29
	DSP slice multiplier[a]	550.00	3	0.09	0	0	1	0
	FloPoCo multiplier[b]	230.47	1	0.04	24	60	0	16
	Proposed multiplier[a]	**714.29**	**6**	**0.25**	**77**	**93**	**0**	**27**
9 × 9	Fabric multiplier (IP Core)	565.93	10	0.60	119	124	0	39
	DSP slice multiplier[a]	550.00	3	0.09	0	0	1	0
	FloPoCo multiplier[b]	173.34	1	0.08	36	122	0	37
	Proposed multiplier[a]	**699.30**	**10**	**0.70**	**187**	**244**	**0**	**64**
12 × 12	Fabric multiplier (IP Core)	529.10	14	1.13	191	188	0	65
	DSP slice multiplier[a]	550.00	3	0.10	0	0	1	0
	FloPoCo multiplier[b]	161.06	1	0.07	48	191	0	55
	Proposed multiplier[a]	**678.43**	**14**	**1.53**	**343**	**453**	**0**	**124**
15 × 15	Fabric multiplier (IP Core)	527.70	18	2.21	295	288	0	92
	DSP slice multiplier[a]	550.00	3	0.10	0	0	1	0
	FloPoCo multiplier[b]	156.15	1	0.09	60	288	0	84
	Proposed multiplier[a]	**668.45**	**18**	**2.83**	**545**	**727**	**0**	**202**

[a] In our proposed multiplier design for an $A \times B$ multiplier where A and B are both n-bit wide, $(n-2) + \lfloor n/3 \rfloor$ pipeline stages are required. The DSP slice-based multiplier had three pipeline stages internal to the DSP slices, and no FF resource consumption was reported by the Xilinx synthesis tool

[b] *FloPoCo* is unable to pipeline any of its multipliers. Frequency of operation was determined by inserting registers at appropriate locations. This inability to pipeline has been reported by us to the authors of [6]. The authors have admitted through personal correspondence that a bug exists in the arithmetic core generation for integer multipliers which they have filed and are trying to solve that issue before the next release

4.5 Integer Squarer Architecture

Although mathematically the squaring operation is a special case of multiplication, integer squarer circuits can be built with lesser hardware overhead than integer multipliers, by intelligent exploitation of the fact that squaring is an one-operand operation [14]. An integer squarer circuit must be optimized such that the number of partial products are reduced and the depth of the partial-product array is shortened. Once all the optimizations are done, the same design philosophy is adhered to for mapping the equivalent architecture on the slices of the target FPGA platform, as in the case of integer multiplier. The identities used for optimization [11, 17] are as follows: $a_i a_j + a_j a_i = 2a_i a_j$, $a_i a_i = a_i$, and $a_i + a_i a_{i-1} = a_i \overline{a_{i-1}} + 2a_i a_{i-1}$ where a_i, a_j are given operand bits. The first identity aims at reducing the number of partial products and depth of the squaring array, whereas the second and third identities together can reduce the depth of the squaring array. We shall now prove certain useful identities to establish the proof of correctness of squaring schemes.

Lemma 4.1 $(a_i + a_i a_{i-1})2^n = a_i \overline{a_{i-1}} 2^n + a_i a_{i-1} 2^{n+1}$

Proof:

$$\begin{aligned} L.H.S. &= (a_i + a_i a_{i-1})2^n \\ &= (a_i(a_{i-1} + \overline{a_{i-1}}) + a_i a_{i-1})2^n \\ &= (a_i a_{i-1} + a_i \overline{a_{i-1}} + a_i a_{i-1})2^n \\ &= a_i \overline{a_{i-1}} 2^n + a_i a_{i-1} 2^{n+1} \end{aligned} \tag{4.28}$$

Lemma 4.2 $(1 + \overline{a_{n-1} a_0})2^n = a_{n-1} a_0 2^n + \overline{a_{n-1} a_0} 2^{n+1}$

Proof:

$$\begin{aligned} L.H.S. &= (1 + \overline{a_{n-1} a_0})2^n \\ &= (a_{n-1} a_0 + \overline{a_{n-1} a_0} + \overline{a_{n-1} a_0})2^n \\ &= (a_{n-1} a_0 + 2\overline{a_{n-1} a_0})2^n \\ &= a_{n-1} a_0 2^n + \overline{a_{n-1} a_0} 2^{n+1} \end{aligned} \tag{4.29}$$

Lemma 4.3 $(a_i + \overline{a_i a_{i-1}})2^n = a_i \overline{a_{i-1}} 2^{n+1} + \overline{a_i \overline{a_{i-1}}} 2^n$

Proof:

$$\begin{aligned} L.H.S. &= (a_i + \overline{a_i a_{i-1}})2^n \\ &= (a_i(a_{i-1} + \overline{a_{i-1}}) + \overline{a_i a_{i-1}})2^n \\ &= (a_i a_{i-1} + a_i\overline{a_{i-1}} + \overline{a_i a_{i-1}})2^n \\ &= (1 + a_i\overline{a_{i-1}})2^n \\ &= (a_i\overline{a_{i-1}} + \overline{a_i\overline{a_{i-1}}} + a_i\overline{a_{i-1}})2^n \\ &= a_i\overline{a_{i-1}}2^{n+1} + \overline{a_i\overline{a_{i-1}}}2^n \end{aligned} \tag{4.30}$$

Lemma 4.4 $-2^{2n-1} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_{n-1}2^{2n-2} = -a_{n-1}\overline{a_{n-2}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2}$

Proof:

$$\begin{aligned} L.H.S. &= -2^{2n-1} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_{n-1}2^{2n-2} \\ &= -2^{2n-1} + a_{n-1}\overline{a_{n-2}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} \ (using\ Lemma\ 4.3) \\ &= (-1 + a_{n-1}\overline{a_{n-2}})2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} \\ &= -\overline{a_{n-1}\overline{a_{n-2}}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} \end{aligned} \tag{4.31}$$

Lemma 4.5 $(1 + a_{\frac{n}{2}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n = \overline{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}2^n + a_{\frac{n}{2}}2^{n+1}$

Proof:

$$\begin{aligned} L.H.S. &= (1 + a_{\frac{n}{2}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= (a_{\frac{n}{2}} + \overline{a_{\frac{n}{2}}} + a_{\frac{n}{2}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= (2a_{\frac{n}{2}} + \overline{a_{\frac{n}{2}}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= a_{\frac{n}{2}}2^{n+1} + (\overline{a_{\frac{n}{2}}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= a_{\frac{n}{2}}2^{n+1} + (\overline{a_{\frac{n}{2}}}(a_{\frac{n}{2}-1} + \overline{a_{\frac{n}{2}-1}}) + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= a_{\frac{n}{2}}2^{n+1} + (\overline{a_{\frac{n}{2}}}a_{\frac{n}{2}-1} + \overline{a_{\frac{n}{2}}}.\overline{a_{\frac{n}{2}-1}} + a_{\frac{n}{2}}a_{\frac{n}{2}-1})2^n \\ &= a_{\frac{n}{2}}2^{n+1} + a_{\frac{n}{2}-1}(\overline{a_{\frac{n}{2}}} + a_{\frac{n}{2}})2^n + \overline{a_{\frac{n}{2}}}.\overline{a_{\frac{n}{2}-1}}2^n \\ &= a_{\frac{n}{2}}2^{n+1} + a_{\frac{n}{2}-1}2^n + \overline{a_{\frac{n}{2}}}.\overline{a_{\frac{n}{2}-1}}2^n \\ &= (a_{\frac{n}{2}-1} + \overline{a_{\frac{n}{2}}})(a_{\frac{n}{2}-1} + \overline{a_{\frac{n}{2}-1}})2^n + a_{\frac{n}{2}}2^{n+1} \\ &= (a_{\frac{n}{2}-1} + \overline{a_{\frac{n}{2}}})2^n + a_{\frac{n}{2}}2^{n+1} \\ &= \overline{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}2^n + a_{\frac{n}{2}}2^{n+1} \end{aligned} \tag{4.32}$$

4.5.1 Unsigned Squarers

An n-bit unsigned integer $A = a_{n-1}a_{n-2}...a_1a_0$ whose value is $A = \sum_{i=0}^{n-1} a_i 2^i$, has its squared value as $S = A^2$, where S is obtained as follows [9, 11, 17], after applying the logic identities for optimization to reduce the number of partial products and the height of the multiplication array. The PP arrays for odd- and even-bit operand unsigned squaring are shown in Figs. 4.16 and 4.17, respectively.

Theorem 4.3 *The square of an n-bit unsigned integer A is given by*
$S_{us} = a_0 + \sum_{i=1}^{n-1} \left(a_i \overline{a_{i-1}} 2^{2i} + a_i a_{i-1} 2^{2i+1}\right) + \sum_{i=2}^{n-1} \sum_{j=0}^{i-2} a_i a_j 2^{i+j+1}$.

Proof:

$$
\begin{aligned}
S_{us} &= \left(\sum_{i=0}^{n-1} a_i 2^i\right)^2 = \sum_{i=0}^{n-1} a_i 2^{2i} + \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} a_i a_j 2^{i+j} + \sum_{j=1}^{n-1}\sum_{i=0}^{j-1} a_j a_i 2^{i+j} \\
&= \sum_{i=0}^{n-1} a_i 2^{2i} + 2\left(\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} a_i a_j 2^{i+j}\right) = \sum_{i=0}^{n-1} a_i 2^{2i} + \sum_{i=1}^{n-1} a_i a_{i-1} 2^{2i} + \sum_{i=2}^{n-1}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \\
&= a_0 + \sum_{i=1}^{n-1} \left(a_i + a_i a_{i-1}\right) 2^{2i} + \sum_{i=2}^{n-1}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \\
&= a_0 + \sum_{i=1}^{n-1} \left(a_i \overline{a_{i-1}} 2^{2i} + a_i a_{i-1} 2^{2i+1}\right) + \sum_{i=2}^{n-1}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \qquad (4.33)
\end{aligned}
$$

Post-optimization, the number of rows of PPs obtained on squaring an unsigned integer with odd number of bits is $\frac{n+1}{2}$, where the topmost row contains $2n$ PPs

Fig. 4.16 7-bit unsigned squaring matrices [17]. **a** Original matrix. **b** Modified matrix

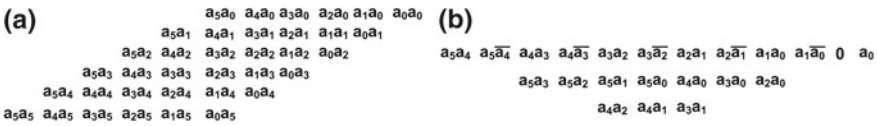

Fig. 4.17 6-bit unsigned squaring matrices [17]. **a** Original matrix. **b** Modified matrix

and the number of PPs decreases by $5 + 4(i-1)$ with i increasing as we keep approaching the bottom rows. Thus, the number of PPs is:

$$N_{us_{odd}} = 2n + \sum_{i=1}^{\frac{n-1}{2}} [2n - (5 + 4(i-1))] = \frac{n(n+1)}{2} + 1 \tag{4.34}$$

Similarly, for even-bit integer squaring, the number of rows of PPs post-optimization is $\frac{n}{2}$, where the topmost row contains $2n$ PPs and the number of PPs decreases by $5 + 4(i-1)$ with i increasing as we keep approaching the bottom rows. Thus, the number of PPs is:

$$N_{us_{even}} = 2n + \sum_{i=1}^{\frac{n}{2}-1} [2n - (5 + 4(i-1))] = \frac{n(n+1)}{2} + 1 \tag{4.35}$$

Clearly, $N_{us_{odd}}$ and $N_{us_{even}}$ are smaller than n^2 (the number of PPs for an unsigned $n \times n$ multiplier). Figure 4.16 shows 7 (odd)-bit unsigned squarer matrices, while Fig. 4.17 shows 6 (even)-bit unsigned squarer matrices in accordance with (4.33).

4.5.2 Two's Complement Squarers

A two's complement integer $A = a_{n-1}a_{n-2}...a_1a_0$ has the value $A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$. Next, we derive the expression for its square value in the same light as was suggested in [13] for the original Baugh–Wooley's two's complement multiplication [3], and applying the previously mentioned identities for the overall hardware optimization [9, 11, 17].

Theorem 4.4 *The square of an n-bit two's complement integer A for odd n is given by*

$$S_{to} = -\overline{a_{n-1}\overline{a_{n-2}}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} + a_{n-1}a_0 2^n + \sum_{i=1}^{n-2}(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}) + \sum_{i=1}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_0}2^{n+1} + a_0 + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1}.$$

For even n, the square value is given by

$$S_{te} = -\overline{a_{n-1}\overline{a_{n-2}}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} + \overline{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}2^n + a_{\frac{n}{2}}2^{n+1} + \sum_{i=1}^{\frac{n}{2}-1}\left(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}\right) + \sum_{i=0}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \sum_{i=\frac{n}{2}+1}^{n-2}\left(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}\right) + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_0.$$

Proof:

$$
\begin{aligned}
A^2 &= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right)^2 \\
&= a_{n-1}a_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i a_j 2^{i+j} - \sum_{i=0}^{n-2} a_i a_{n-1} 2^{n+i-1} - \sum_{i=0}^{n-2} a_{n-1} a_i 2^{n+i-1} \\
&= a_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i a_j 2^{i+j} + \sum_{i=0}^{n-2} \overline{a_i a_{n-1}} 2^{n+i-1} - \sum_{i=0}^{n-2} 2^{n+i-1} + \sum_{i=0}^{n-2} \overline{a_{n-1} a_i} 2^{n+i-1} \\
&\quad - \sum_{i=0}^{n-2} 2^{n+i-1} \; (as\; x + \overline{x} = 1, -x = \overline{x} - 1) \\
&= a_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i a_j 2^{i+j} + \sum_{i=0}^{n-2} \overline{a_i a_{n-1}} 2^{n+i} - \sum_{i=0}^{n-2} 2^{n+i} \\
&= -2^{2n-1} + 2^n + \sum_{i=0}^{n-2} \overline{a_{n-1} a_i} 2^{n+i} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i a_j 2^{i+j} + a_{n-1}2^{2n-2} \\
&\quad (as \sum_{i=0}^{n-2} 2^{n+i} = 2^{2n-1} - 2^n) \\
&= -2^{2n-1} + 2^n + \sum_{i=0}^{n-2} \overline{a_{n-1} a_i} 2^{n+i} + \sum_{i=0}^{n-2}\sum_{\substack{j=0 \\ i\neq j}}^{n-2} a_i a_j 2^{i+j} + \sum_{i=0}^{n-2} a_i a_i 2^{2i} + a_{n-1}2^{2n-2} \\
&= -2^{2n-1} + 2^n + \sum_{i=0}^{n-2} \overline{a_{n-1} a_i} 2^{n+i} + \sum_{i=1}^{n-2}\sum_{j=0}^{i-1} a_i a_j 2^{i+j+1} + \sum_{i=0}^{n-2} a_i a_i 2^{2i} + a_{n-1}2^{2n-2} \\
&= -2^{2n-1} + 2^n + \sum_{i=0}^{n-2} \overline{a_{n-1} a_i} 2^{n+i} + \sum_{i=1}^{n-2}\sum_{j=0}^{i-1} a_i a_j 2^{i+j+1} + \sum_{i=0}^{n-1} a_i 2^{2i}
\end{aligned}
\tag{4.36}
$$

Equation (4.36) is the starting point for deriving the final squared expressions for odd and even-bit integers. For odd-bit integers, the final squared expression S_{to} can be proved as follows:

$$
\begin{aligned}
S_{to} &= -2^{2n-1} + 2^n + \overline{a_{n-1} a_0} 2^n + \sum_{i=1}^{n-3} \overline{a_{n-1} a_i} 2^{n+i} + \overline{a_{n-1} a_{n-2}} 2^{2n-2} + a_0 + \sum_{i=1}^{n-2} a_i 2^{2i} \\
&\quad + a_{n-1}2^{2n-2} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_1 a_0 2^2 + \sum_{i=2}^{n-2} a_i a_{i-1} 2^{2i} \\
&\quad (splitting\ the\ summation\ ranges\ of\ (4.36))
\end{aligned}
$$

$$
\begin{aligned}
&= -2^{2n-1} + a_{n-1}a_0 2^n + \overline{a_{n-1}a_0}2^{n+1} + \sum_{i=1}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_0 \\
&\quad + \sum_{i=1}^{n-2} a_i 2^{2i} + a_{n-1}2^{2n-2} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + \sum_{i=1}^{n-2} a_i a_{i-1} 2^{2i} \ (using\ Lemma\ 4.2) \\
&= -2^{2n-1} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_{n-1}2^{2n-2} + a_{n-1}a_0 2^n + \sum_{i=1}^{n-2}(a_i + a_i a_{i-1})2^{2i} \\
&\quad + \sum_{i=1}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_0}2^{n+1} + a_0 + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \\
&\quad (rearranging\ and\ grouping\ the\ terms) \\
&= -2^{2n-1} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_{n-1}2^{2n-2} + a_{n-1}a_0 2^n + \sum_{i=1}^{n-2}(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}) \\
&\quad + \sum_{i=1}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_0}2^{n+1} + a_0 + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \ (using\ Lemma\ 4.1) \\
&= -\overline{a_{n-1}\overline{a_{n-2}}}2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}2^{2n-2} + a_{n-1}a_0 2^n + \sum_{i=1}^{n-2}(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}) \\
&\quad + \sum_{i=1}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_0}2^{n+1} + a_0 + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} \qquad (4.37) \\
&\quad (using\ Lemma\ 4.4)
\end{aligned}
$$

Figure 4.18 shows 7 (odd)-bit two's complement squarer matrices in accordance with (4.37).

For even values of n, the final squared expression S_{te} starting from (4.36) can be proved as follows:

$$
\begin{aligned}
S_{te} &= -2^{2n-1} + 2^n + \sum_{i=0}^{n-3}\overline{a_{n-1}a_i}2^{n+i} + \overline{a_{n-1}a_{n-2}}2^{2n-2} + a_0 + \sum_{i=1}^{\frac{n}{2}-1} a_i 2^{2i} + a_{\frac{n}{2}}2^n \\
&\quad + \sum_{i=\frac{n}{2}+1}^{n-2} a_i 2^{2i} + a_{n-1}2^{2n-2} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + \sum_{i=1}^{n-2} a_i a_{i-1} 2^{2i}
\end{aligned}
$$

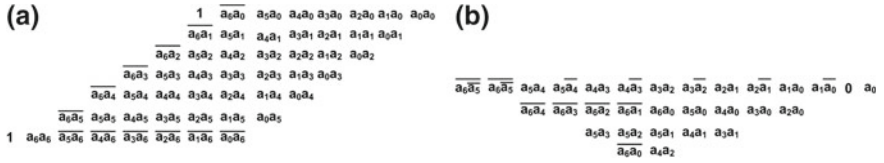

Fig. 4.18 7-bit two's complement squaring matrices [17]. **a** Original matrix. **b** Modified matrix

$$
\begin{aligned}
&(\textit{splitting the summation ranges of } (4.36))\\
&= -2^{2n-1} + 2^n + \sum_{i=0}^{n-3} \overline{a_{n-1}a_i}\,2^{n+i} + \overline{a_{n-1}a_{n-2}}\,2^{2n-2} + a_0 + \sum_{i=1}^{\frac{n}{2}-1} a_i 2^{2i} + a_{\frac{n}{2}} 2^n\\
&\quad + \sum_{i=\frac{n}{2}+1}^{n-2} a_i 2^{2i} + a_{n-1}2^{2n-2} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + \sum_{i=1}^{\frac{n}{2}-1} a_i a_{i-1} 2^{2i} + a_{\frac{n}{2}} a_{\frac{n}{2}-1} 2^n\\
&\quad + \sum_{i=\frac{n}{2}+1}^{n-2} a_i a_{i-1} 2^{2i} \ (\textit{splitting the summation ranges further})\\
&= -2^{2n-1} + \overline{a_{n-1}a_{n-2}}\,2^{2n-2} + a_{n-1}2^{2n-2} + 2^n(1 + a_{\frac{n}{2}} + a_{\frac{n}{2}} a_{\frac{n}{2}-1}) + \sum_{i=1}^{\frac{n}{2}-1} (a_i + a_i a_{i-1})\, 2^{2i}\\
&\quad + \sum_{i=0}^{n-3} \overline{a_{n-1}a_i}\,2^{n+i} + \sum_{i=\frac{n}{2}+1}^{n-2} (a_i + a_i a_{i-1})\, 2^{2i} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_0\\
&\quad (\textit{rearranging and grouping the terms})\\
&= -\overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-2} + 2^n(1 + a_{\frac{n}{2}} + a_{\frac{n}{2}} a_{\frac{n}{2}-1}) + \sum_{i=1}^{\frac{n}{2}-1} (a_i + a_i a_{i-1})\, 2^{2i}\\
&\quad + \sum_{i=0}^{n-3} \overline{a_{n-1}a_i}\,2^{n+i} + \sum_{i=\frac{n}{2}+1}^{n-2} (a_i + a_i a_{i-1})\, 2^{2i} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_0\\
&\quad (\textit{using Lemma } 4.4)\\
&= -\overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-2} + \overline{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}\,2^n + a_{\frac{n}{2}} 2^{n+1} + \sum_{i=1}^{\frac{n}{2}-1} (a_i + a_i a_{i-1})\, 2^{2i}\\
&\quad + \sum_{i=0}^{n-3} \overline{a_{n-1}a_i}\,2^{n+i} + \sum_{i=\frac{n}{2}+1}^{n-2} (a_i + a_i a_{i-1})\, 2^{2i} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_0\\
&\quad (\textit{using Lemma } 4.5)\\
&= -\overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-1} + \overline{a_{n-1}\overline{a_{n-2}}}\,2^{2n-2} + \overline{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}\,2^n + a_{\frac{n}{2}} 2^{n+1}\\
&\quad + \sum_{i=1}^{\frac{n}{2}-1} \left(a_i \overline{a_{i-1}}\,2^{2i} + a_i a_{i-1} 2^{2i+1}\right) + \sum_{i=0}^{n-3} \overline{a_{n-1}a_i}\,2^{n+i} + \sum_{i=\frac{n}{2}+1}^{n-2} \left(a_i \overline{a_{i-1}}\,2^{2i} + a_i a_{i-1} 2^{2i+1}\right)\\
&\quad + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2} a_i a_j 2^{i+j+1} + a_0 (\textit{using Lemma } 4.1) \qquad (4.38)
\end{aligned}
$$

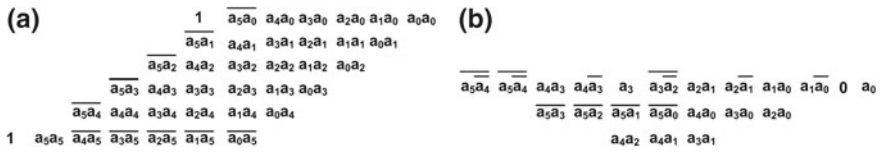

Fig. 4.19 6-bit two's complement squaring matrices [17]. **a** Original matrix. **b** Modified matrix

Fig. 4.19 shows 6 (even)-bit two's complement squarer matrices in accordance with (4.38).

The number of PPs obtained on squaring an n-bit two's complement integer is $N_{us_{odd}} + 1$ for odd n, and $N_{us_{even}}$ for even n, both of which are lower than $(n^2 + 2)$ (for an $n \times n$ two's complement multiplier).

4.5.3 Combined Unsigned and Two's Complement Squarer

A control signal t is defined where $t = 1$ for two's complement squaring and $t = 0$ for unsigned squaring. $\widehat{p}$ is used to denote $p \oplus t$ and "|" denotes the Boolean OR operation. Thus, for an odd-bit integer, its combined squared expression S_{co} can be written as follows, after closely observing (4.33) and (4.37):

$$S_{co} = -(a_{n-1}a_{n-2}|\overline{a_{n-1}}t)\,2^{2n-1} + \widehat{a_{n-1}\overline{a_{n-2}}}2^{2n-2} + a_{n-1}a_0 2^n + \sum_{i=1}^{n-2}\left(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}\right) + \overline{a_{n-1}a_0}t\,2^{n+1} + \sum_{i=1}^{n-3}\widehat{a_{n-1}a_i}\,2^{n+i} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2}a_i a_j 2^{i+j+1} + a_0 \tag{4.39}$$

Figure 4.20 shows 7 (odd)-bit combined squarer matrices in accordance with (4.39).

Similarly, for even-bit integer, its combined squared expression S_{ce} can be written as follows, after closely observing (4.33) and (4.38), where "|" denotes the Boolean OR operation, and $\widehat{a_i b_j} = a_i b_j \oplus t$:

$$S_{ce} = -(a_{n-1}a_{n-2}|\overline{a_{n-1}}t)\,2^{2n-1} + \widehat{a_{n-1}\overline{a_{n-2}}}2^{2n-2} + \widehat{a_{\frac{n}{2}}\overline{a_{\frac{n}{2}-1}}}2^n + a_{\frac{n}{2}}\left(t|a_{\frac{n}{2}-1}\right)2^{n+1} + \sum_{i=1}^{\frac{n}{2}-1}\left(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}\right) + \sum_{i=\frac{n}{2}+1}^{n-2}\left(a_i\overline{a_{i-1}}2^{2i} + a_i a_{i-1}2^{2i+1}\right) + \sum_{i=0}^{n-3}\widehat{a_{n-1}a_i}\,2^{n+i} + \sum_{i=2}^{n-2}\sum_{j=0}^{i-2}a_i a_j 2^{i+j+1} + a_0 \tag{4.40}$$

They combined unsigned and two's complement squaring arrays have the same number of PPs as the two's complement squaring arrays. Fig. 4.21 shows 6 (even)-bit combined squarer matrices in accordance with (4.40).

The implementation of optimized squarers following the mathematical analyses presented above has been done using the convention of carry-save, pipelined

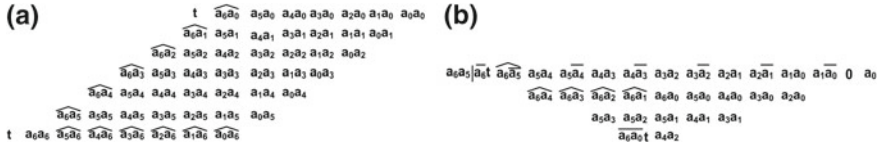

Fig. 4.20 7-bit combined squaring matrices [17]. **a** Original matrix. **b** Modified matrix

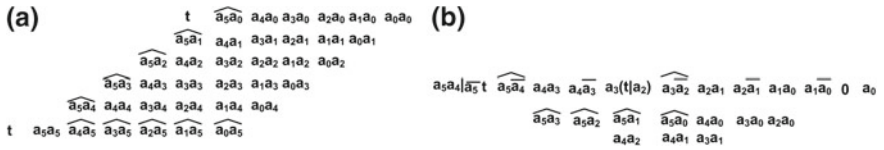

Fig. 4.21 6-bit combined squaring matrices [17]. **a** Original matrix. **b** Modified matrix

approach. For an $A \times A$ combined unsigned and two's complement squarer where A is an n-bit number, $\lfloor n/2 \rfloor + \lfloor n/3.6 \rfloor$ pipeline stages are required when n is odd, and $\lfloor n/2 \rfloor - 1 + \lfloor n/3.6 \rfloor$ pipeline stages when n is even. Thus, the number of pipeline stages for a squarer is lesser than the number of pipeline stages for an $n \times n$ combined unsigned and two's complement multiplier $(n - 2 + \lfloor n/3 \rfloor)$. It is to be noted that (4.37)–(4.40) have been wrongly stated in [17] where the PP at the sign bit position should have had the opposite polarity and the upper limit of the summation range of the terms $(\overline{a_{n-1}a_i}2^{n+i})$ of (4.37) and (4.38), and $(\widehat{a_{n-1}a_i}2^{n+i})$ of (4.39) and (4.40) should have been $(n - 3)$ instead of $(n - 2)$.

We adopt a similar approach of carry-save pipelined addition for adding the partial products for squaring as was done for multiplication. The grouping of the partial products for the squarer implementation has been shown in Fig. 4.22, where each carry chain of a slice can be configured as a 3-bit adder generating a 3-bit sum and 1-bit carry. Since squaring is essentially a one-operand operation, and mostly all the operand bits are required at every stage of the pipelined addition, we realize the delays using FFs. Hence, the SRLC32E primitives are no longer required, in contrast to multiplier circuits.

4.5.4 DSP Slice-Based Squarers

DSP multipliers can also be configured to function as squarers. With the help of a simple trick, we can optimize usage of DSP multipliers to realize multipliers or squarers. For example, a 19×19 squarer can be realized using a single DSP 25×18

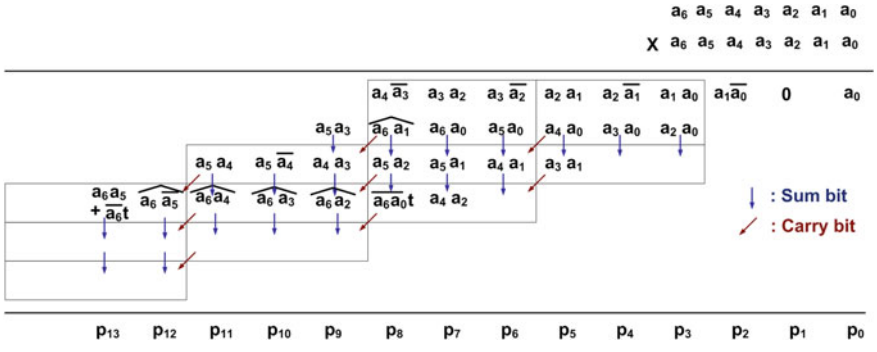

Fig. 4.22 Array squaring depicting the grouping of partial products

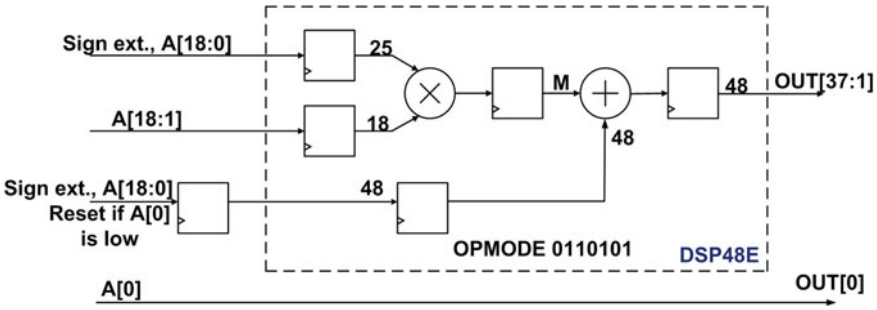

Fig. 4.23 Squaring extension by one 1 bit in DSP slice for 19 × 19 bit squaring [21]

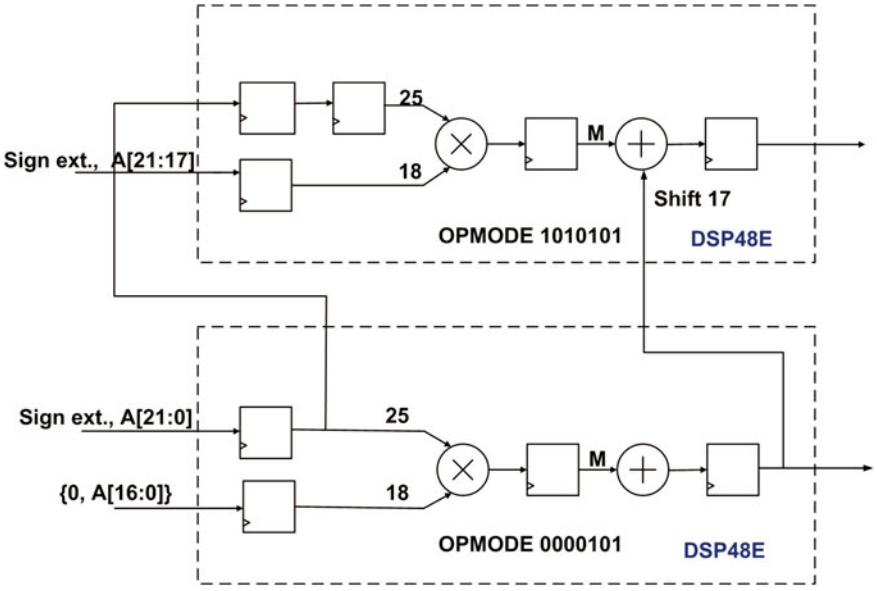

Fig. 4.24 DSP-based squarer for 22 × 22 operand width

multiplier, where the final output is realized by concatenating the LSB of the input operand at the LSB position of the DSP slice output. To realize such an operation, the OPMODE and ALUMODE inputs [21] can be set as "0110101" and "0000", respectively. This holds as the square of an odd number is odd, and the square of an even number is even, and is illustrated in Fig. 4.23. The adder present in the DSP slice multiplier performs addition of the first row of partial products with the remaining partial product array. DSP slices can be cascaded for realizing higher order multiplications and squaring. Figure 4.24 shows the DSP slice implementation for a 22-bit squarer. Here, the OPMODE input [21] of the lower and upper DSP slices

can be set as "0000101" and "1010101", respectively. The ALUMODE [21] input, however is "0000" for both the DSP slices.

Just as in the case for multipliers, custom architectures like the ones developed here demand higher hardware cost and greater PDP even for squarers. The approach described so far can be well suited to numbers of smaller bit-width. Higher bit-width implementations must be realized using DSP slices or a combination of DSP slices and fabric logic [1, 7] to save on hardware and PDP with some compromise on speed (Table 4.4).

4.5.5 FloPoCo-Based Squarers

FloPoCo can also generate integer squarers, but is unable to pipeline squarers with input operand width that is less than 13. Again, the pipelined design descriptions generated for higher operand widths are unable to match the user specified operating frequency.

4.5.6 Squarer Implementation Results

The proposed custom implementation of squarers guarantees higher speed and lower hardware cost in comparison to the IP Core-based fabric squarer and *FloPoCo*-based circuit. Actually, IP Core-based fabric implementation does not support squarers explicitly—the multipliers with both its inputs tied together realize the squarers. Again, DSP slice-based squarers come at a lower hardware cost and lesser PDP. *FloPoCo* is unable to pipeline 7×7 and 13×13 squarers, registers have been added at appropriate locations to realize the operating frequency of those two implementations.

4.6 Universal Shift Register Architecture

4.6.1 Universal Shift Register

A *Universal Shift Register* has bidirectional shifting as well as parallel load capabilities. The block diagram showing the basic building blocks of the architecture is shown in Fig. 4.25 and its functionality has been tabulated in Table 4.5. The inputs S_1 and S_0 control the mode of operation of the registers. The shift-left and shift-right functionality can be used to realize multiplication and division operation by two, respectively. Other functionality involves loading of external data to all the FFs of the register in parallel, and also retaining the previous output data in the subsequent

Table 4.4 Integer squarer implementation results

Operand width	Design style	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
7×7	Fabric squarer (IP core)	623.44	4	0.24	80	63	0	27
	DSP slice squarer[a]	550.00	3	0.09	0	0	1	0
	FloPoCo squarer[b]	242.48	1	0.04	20	60	0	20
	Proposed squarer[a]	**706.71**	**4**	**0.13**	**49**	**39**	**0**	**17**
13×13	Fabric squarer (IP core)	521.38	9	0.99	228	235	0	76
	DSP slice squarer[a]	550.00	3	0.09	0	0	1	0
	FloPoCo squarer[b]	154.08	1	0.09	207	38	0	61
	Proposed squarer[a]	**673.85**	**9**	**0.64**	**201**	**149**	**0**	**65**
19×19	Fabric squarer (IP core)	507.87	14	2.76	471	464	0	150
	DSP slice squarer[a]	500.00	3	0.12	48	1	1	12
	FloPoCo squarer[b]	324.99	4	0.58	114	574	0	177
	Proposed squarer[a]	**652.32**	**14**	**1.79**	**449**	**331**	**0**	**139**
22×22	Fabric squarer (IP core)	464.47	16	3.56	585	602	0	165
	DSP slice squarer[a]	500.00	3	0.24	0	0	2	0
	FloPoCo squarer[b]	129.25	4	0.94	142	713	0	211
	Proposed squarer[a]	**643.50**	**16**	**2.49**	**583**	**433**	**0**	**185**

[a] In our proposed squarer design for an $A \times A$ squarer where A is an n-bit number, there are $\lfloor n/2 \rfloor + \lfloor n/3.6 \rfloor$ pipeline stages when n is odd, and $\lfloor n/2 \rfloor - 1 + \lfloor n/3.6 \rfloor$ pipeline stages when n is even. The DSP slice-based multiplier had three pipeline stages internal to the DSP slices, and no FF resource consumption was reported by the Xilinx synthesis tool

[b] *FloPoCo* is unable to pipeline the 7×7 and 13×13 squarers. Frequency of operation was determined by inserting registers at appropriate locations

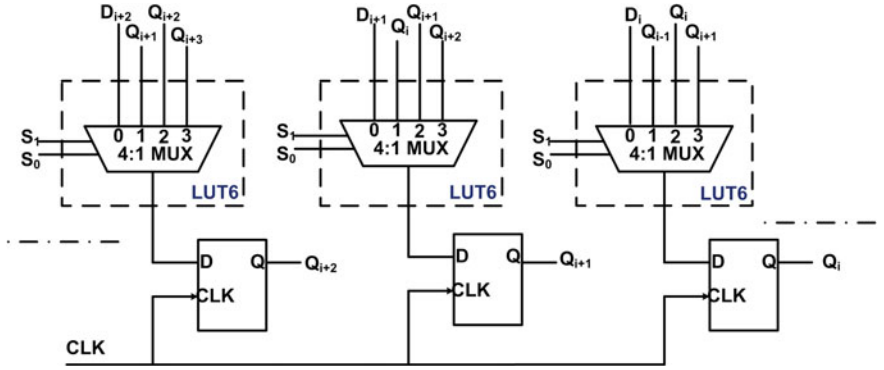

Fig. 4.25 Universal shift register

Table 4.5 Function table of universal shift register

Mode control		Register operation
S_1	S_0	
0	0	Parallel load
0	1	Shift left
1	0	Freeze data
1	1	Shift right

clock cycle(s). The combinational logic block deciding the functionality of the register can be implemented using the 6-input LUT where all the six inputs of the LUT are fully utilized.

Currently, *FloPoCo* (v 2.5.0) does not generate HDL for universal shift registers (Table 4.6).

4.6.2 Universal Shift Register Implementation Results

The Universal Shift Register implementation has been compared for the design implemented following the proposed constrained placement methodology, and the design implemented using behavioral modeling. It was observed from the physical implementation that design inferred from the behavioral model had the programmed slices placed in a very irregular pattern on the FPGA fabric. The superior PDP results obtained for constrained placement design can be completely attributed to the regular and compact placement of the slices on FPGA. This significantly reduces routing complexity and consequently has lower PDP. The FPGA routing is based on programmable switch routing and the MOSFET switches which actually control the routing are responsible for maximum power dissipation. *FloPoCo* does not support universal shift register implementations till its latest release.

Table 4.6 Universal shift register implementation results

Power operand width	Design style	Freq (MHz)	Delay product (pJ)	#FF	#LUT	#Slice
32	Behavioral	921.66	38.24	32	32	8
	Proposed	**1039.50**	**27.58**	**32**	**32**	**8**
48	Behavioral	886.52	48.21	48	48	12
	Proposed	**1042.75**	**44.55**	**48**	**48**	12
64	Behavioral	815.00	60.54	64	64	16
	Proposed	**1042.75**	**41.47**	**64**	**64**	**16**
96	Behavioral	516.26	107.95	96	96	24
	Proposed	**1039.50**	**57.48**	**96**	**96**	**24**
128	Behavioral	654.02	132.41	128	128	32
	Proposed	**1041.67**	**128.81**	**128**	**128**	**32**

4.7 Summary

All our implementation results clearly reveal that the proposed design methodology outperforms all other modes of implementation. This is achieved primarily due to two reasons: (a) correct instantiation of logic elements, (b) constrained placement of logically related fabric elements in adjacent locations with closest proximity, where the proximity of bit indices imply physical proximity of the logic blocks on the FPGA fabric. A partial floorplan view for a 128-bit adder is shown in Fig. 4.26 and is shown

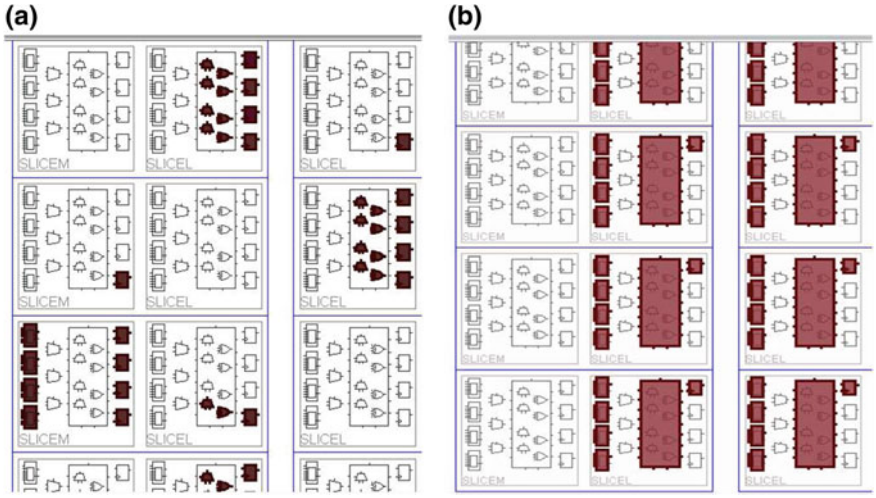

Fig. 4.26 Partial floorplan views for 128-bit adder for fabric IP Core-based adder and hybrid RCA with constrained placement. **a** Partial Floorplan of IP Core-based Fabric Adder with unconstrained placement. **b** Partial Floorplan of Hybrid RCA with constrained placement

for two different implementation modes: Fabric IP Core-based 128-bit adder with unconstrained placement, and a 128-bit Hybrid RCA with primitive instantiation and constrained placement. From this figure, it is clearly evident that the CAD tool performs a random unoptimized placement coupled with unoptimized inference of logic elements.

The next chapter describes implementations of two controlpath circuits—a comparator and a loadable bidirectional counter, using the technique of direct primitive instantiation coupled with constrained placement.

References

1. Athow, J.L., Al-Khalili, A.J.: Implementation of large–integer hardware multiplier in Xilinx FPGA. In: 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 1300–1303 (2008)
2. Brent, R.P., Kung, H.T.: A regular layout for parallel adders. IEEE Trans. Comput. **C–31**(3), 260–264 (1982)
3. C.R, Baugh, Wooley, B.A.: A two's complement parallel array multiplication algorithm. IEEE Trans. Comput. **C–22**(12), 1045–1047 (1973)
4. Chapman, K.: Xilinx Inc., Saving costs with the SRL16E, white paper: Xilinx FPGAs WP271 (v1.0). http://www.xilinx.com/support/documentation/white_papers/wp271.pdf Cited 22 May 2008
5. Chen, G., Liu, F.: Proofs of correctness and properties of integer adder circuits. IEEE Trans. Comput. **59**(1), 134–136 (2010)
6. de Dinechin, F., Pasca, B.: Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. **28**(3), 18–27 (2011)
7. Dinechin, F. de, Pasca, B.: Large multipliers with fewer DSP blocks. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 250–255 (2009)
8. Feng, L., QingPing, T., Ait, M.O.: Formal proof of integer adders using all-prefix-sums operation. Sci. China Inf. Sci. **55**(9), 1949–1960 (2012)
9. Griesbach, W.R., Kolagotla, R.K.: Squarer With Diagonal Row Merged Into Folded Partial Product Array. U.S. Patent 6,018,758 Available: http://www.google.com.ar/patents/US6018758 Cited 25 Jan. 2000
10. Hormigo, J., Jaime, F.J., Villalba, J., Zapata, E.L.: Efficient implementation of carry–save adders in FPGAs. In: 20th IEEE International Conference on Application–specific Systems, Architectures and Processors (ASAP), pp. 207–210 (2009)
11. Kolagotla, R.K., Griesbach, W.R., Srinivas, H.R.: VLSI Implementation of 350 MHz 0.35μm 8 bit merged squarer. Electron. Lett. J. **34**(1), 47–48 (1998)
12. Koren, I.: Computer Arithmetic Algorithms, 2nd edn. A. K. Peters, Ltd., Natick (2002)
13. Kroft, D.: Comments on a two's complement parallel array multiplication. IEEE Trans. Comput. **C–23**(12), 1327–1327 (1974)
14. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Design, 2nd edn. Oxford University Press, New York (2000)
15. Parhi, K.K.: VLSI Digital Signal Processing Systems: Design and Implementation. Wiley, New Delhi (2007)
16. Perri, S., Zicari, P., Corsonello, P.: Efficient Absolute Difference Circuits in Virtex–5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 309–313 (2010)
17. Wires, K.E., Schulte, M.J., Marquette, L.P., Balzola, P.I.: Combined Unsigned and Two's Complement Squarers. In: Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers **2**, 1215–1219 (1999)

18. Xilinx Inc.: 7 Series FPGAs configurable logic block, UG474 (v1.5). http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf Cited 6 Aug 2013
19. Xilinx Inc.: Application note: spartan–3 FPGA series, using look–up tables as shift registers (SRL16) in spartan–3 generation FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp465.pdf Cited 20 May 2005
20. Xilinx Inc.: Spartan-6 FPGA configurable logic block, UG384 (v1.1).http://www.xilinx.com/support/documentation/user_guides/ug384.pdf Cited 23 Feb 2010
21. Xilinx Inc.: Virtex–5 FPGA XtremeDSP design considerations user guide, UG193 (v3.5). http://www.xilinx.com/support/documentation/user_guides/ug193.pdf Cited 26 Jan 2012
22. Xilinx Inc.: Virtex–5 libraries guide for HDL designs, UG621 (v11.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex5_hdl.pdf Cited 6 Sep 2009
23. Xilinx Inc.: Virtex-6 FPGA configurable logic block, UG364 (v1.2). http://www.xilinx.com/support/documentation/user_guides/ug364.pdf Cited 3 Feb 2012
24. Zicari, P., Perri, S.: A Fast Carry Chain Adder for Virtex–5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 304–308 (2010)

Chapter 5
Architecture of Controlpath Circuits

Abstract This chapter explores the mathematical analyses and bit-sliced architectures of the pipelined implementations of two controlpath circuits: *integer comparator* and *loadable bidirectional counter* supported by the *FlexiCore* CAD tool currently.

5.1 Introduction

Integer comparators and loadable bidirectional counters (Finite State Machines (FSMs)) have regular structures. The architecture of the circuits, as well as the Boolean algebraic mathematical analyses to justify all the proposed architectures have been explained in detail. The design automation of the circuits using the *FlexiCore* tool has also been discussed.

The rest of the chapter is organized as follows. In Sect. 5.2, we present the architectural details, Boolean logic analyses, and implementation results for an integer comparator. Section 5.3 presents the details of a loadable, bidirectional counter that detects terminal count along with its Boolean logic analyses and implementation results. We conclude in Sect. 5.4.

5.2 Integer Comparator Architecture

Comparator is a widely used circuit for control path implementations. The comparator design proposed by us accepts a pair of two n-bit unsigned numbers A and B, and generates active-high outputs depending on whether A is greater than, equal to, or less than B.

5.2.1 Proposed Comparator Architecture

The LUT and gate-level architectures of the proposed comparator design are shown in Figs. 5.1 and 5.2, respectively. For the module detecting the condition $A < B$, each

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_5

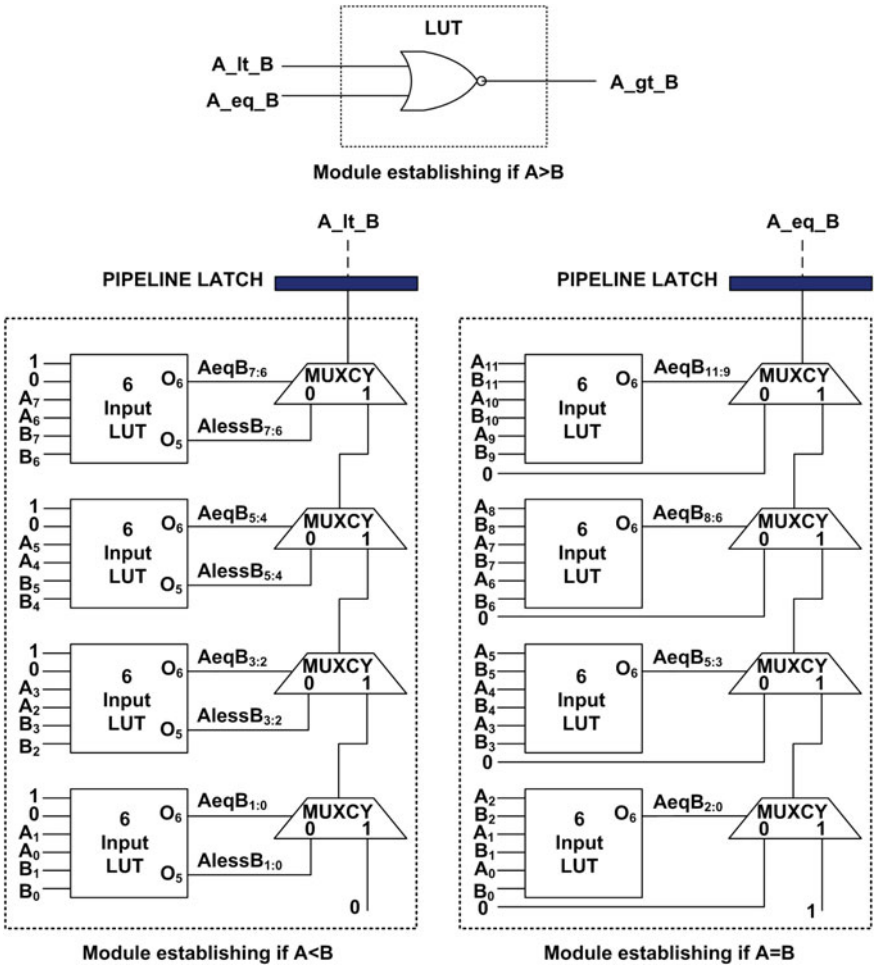

Fig. 5.1 LUT-level architecture of the comparator

LUT in a slice accepts 2-bit sub-words $A_{i:i-1}$ and $B_{i:i-1}$ with i ranging from 0 to $n-1$, and generates the outputs $Aeq\,B_{i:i-1}$ and $Aless\,B_{i:i-1}$. The output $Aeq\,B_{i:i-1}$ is 1 if $A_{i:i-1} = B_{i:i-1}$, and is used as the select signal of the multiplexer MUXCY, whereas $Aless\,B_{i:i-1} = 1$ if $A_{i:i-1} < B_{i:i-1}$, which is also an input to the multiplexer that is selected if $Aeq\,B_{i:i-1} = 0$ [1]. For an n-bit less-than comparator, its output $A\_l\_B_n$ is obtained using the following recurrence relation:

$$A\_l\_B_n = \overline{Aeq\,B_{n:n-1}}\,Aless\,B_{n:n-1} + Aeq\,B_{n:n-1}\,A\_l\_B_{n-2} \tag{5.1}$$

where the *base* condition is $A\_l\_B_0 = 0$. This recurrence relation bears exact resemblance to (3.11) making it an ideal candidate for carry chain implementation.

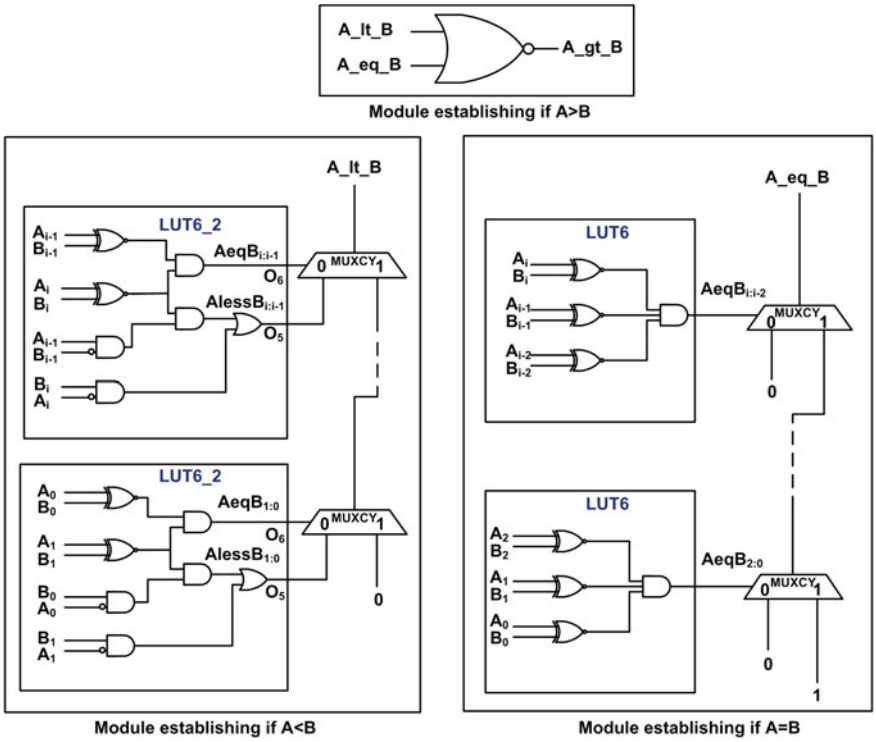

Fig. 5.2 Gate-level architecture of the comparator

For the module detecting the equality condition $A = B$, each LUT accepts two 3-bit binary words $A_{i:i-2}$ and $B_{i:i-2}$ and outputs $Aeq\,B_{i:i-1}$ which goes high when $A_{i:i-2} = B_{i:i-2}$. This output is connected to the select line input of the multiplexer of the carry chain. The carry chain has been configured in such a way so that it can logically function as a wide input AND gate. For an n-bit equality comparator, its output $A\_eq\_B_n$ is obtained using the following recurrence relation:

$$\begin{aligned} A\_eq\_B_n &= A\_eq\_B_{n:n-2} A\_eq\_B_{n-2} \\ &= \overline{A\_eq\_B_{n:n-2}} \cdot 0 + A\_eq\_B_{n:n-2} A\_eq\_B_{n-2} \end{aligned} \tag{5.2}$$

where the *base* condition is $A\_eq\_B_0 = 1$. This recurrence relation also bears exact resemblance to (3.11), making it an ideal candidate for carry chain implementation. The output of the two modules A_lt_B and A_eq_B is fed to a NOR gate (realized using LUT) to obtain A_gt_B, which goes high when $A > B$. For an n-bit comparator, $\lceil n/8 \rceil$ pipeline stages are required.

The unsigned comparator architecture can be extended to handle signed numbers, without disturbing the original architecture. The equality comparator for signed and unsigned numbers is identical. However, the less-than comparator is made to accept

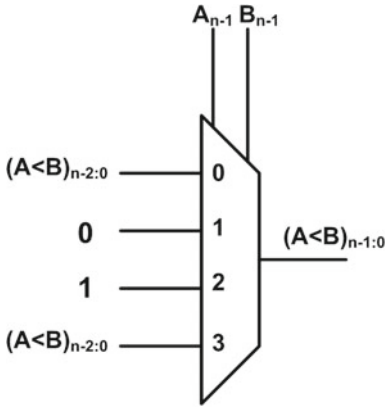

Fig. 5.3 Additional circuitry for unsigned comparator to handle two's complement integers

the lower $(n-1)$ bits and its output $(A < B)_{n-2:0}$ is fed to the inputs of a $4:1$ multiplexer as shown in Fig. 5.3. The select lines to the multiplexer are the sign bits of the input integers. If the select lines are of opposite polarity, the integer with $MSB = 1$ is automatically the smaller integer and the less-than comparator takes the required decision. If the select lines are of same polarity, the final output is the output of the $(n-1)$-bit unsigned comparator, $(A < B)_{n-2:0}$, as shown in Table 5.1. This $4:1$ multiplexer shown in Fig. 5.3 has no more than *three* distinct inputs and can be realized using a single LUT.

5.2.2 DSP Slice-Based Comparator

Comparators can also be designed using DSP48E slices and n-bit comparators with $n > 48$ can be realized by cascading $\lceil n/48 \rceil$ DSP48E slices along a column. To achieve the desired functionality, the slices have been configured as a signed magnitude subtractor and pattern detector each by setting the attributes "OPMODE" as "0110011", and "ALUMODE" as "0011" [4]. However, it is to be noted that comparators realized using DSP48E slices handle signed numbers unlike that of the

Table 5.1 Functionality of two's complement comparator

Sign bits		Output
A_{n-1}	B_{n-1}	$(A < B)_{n-1:0}$
0	0	$(A < B)_{n-2:0}$
0	1	0
1	0	1
1	1	$(A < B)_{n-2:0}$

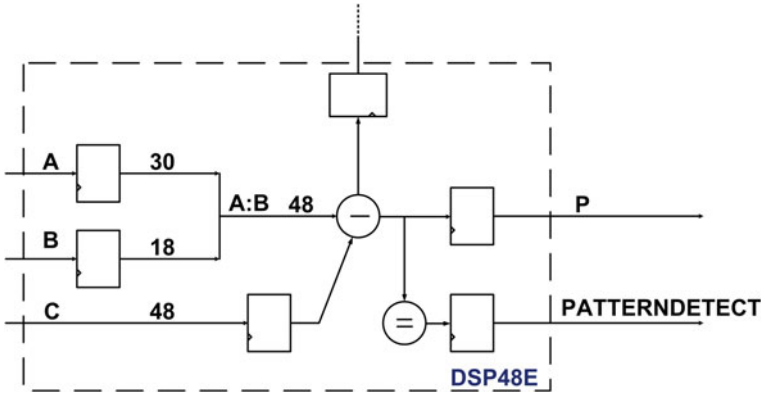

Fig. 5.4 Xilinx Virtex-5 DSP slice-based comparator [4]

proposed comparator design which handles unsigned numbers. Figure 5.4 illustrates the DSP slice-based comparator architecture [4]. If the DSP48E slice is configured to perform a subtraction and the pattern detector is used, then $C > (A : B)$ and $C = (A : B)$ can be simultaneously detected. The sign bit of the output P indicates whether $A : B$ is greater than or less than C. The *PATTERNDETECT* output indicates whether $(A : B) - C == 0$. The outputs detecting $A = B$ and $A < B$ are subsequently passed through a NOR gate to determine if $A > B$. Comparators of higher word length ($n > 48$) are realized by cascading DSP slices along a column. In such cases, the equality check is achieved by AND-ing the corresponding *PATTERNDETECT* outputs.

It is to be noted that till the latest release (v 2.5.0), *FloPoCo* does not support comparators.

5.2.3 Comparator Implementation Results

The proposed comparator design with constrained placement exhibits superior operand width scalability in terms of speed. However, Xilinx IP Core-based comparator is not supported for Xilinx Virtex-5 FPGA, device family XC5VLX330T, package FF1738, and speed grade-2 using the Xilinx ISE 12.4 design environment, and hence its performance could not be reported. The DSP slice-based comparators, on the other hand, give a lower speed performance in comparison to the proposed fabric logic, but give the best PDP. *FloPoCo* does not support comparator implementation till its latest release (Table 5.2).

Table 5.2 Comparator implementation results

Operand width	Design style	Freq (MHz)	Latency (#clk cycles)	Power delay product (nJ)	#FF	#LUT	#DSP	#Slice
32	DSP slice comparator [a]	550.00	2	0.08	0	1	1	1
	Proposed comparator[a]	**790.51**	**3**	**0.13**	**8**	**31**	**0**	**8**
48	DSP slice comparator [a]	550.00	2	0.09	0	1	1	1
	Proposed comparator[a]	**789.89**	**5**	**0.34**	**12**	**44**	**0**	**12**
64	DSP slice comparator [a]	465.12	2	0.09	0	2	2	1
	Proposed comparator[a]	**789.27**	**8**	**0.44**	**16**	**62**	**0**	**16**
96	DSP slice comparator [a]	465.12	2	0.12	0	2	2	1
	Proposed comparator[a]	**789.27**	**12**	**1.18**	**24**	**88**	**0**	**22**
128	DSP slice comparator[a]	465.12	3	0.22	0	2	2	1
	Proposed comparator [a]	**788.02**	**16**	**1.46**	**32**	**119**	**0**	**30**

[a] In our proposed comparator design with constrained placement, an n-bit comparator had $\lceil n/8 \rceil - 1$ pipeline stages. For the DSP slice-based comparator, no pipelining was done if $n < 48$, while if $n > 48$, the number of pipeline stages internal to the DSP slices were $\lceil n/48 \rceil - 1$, and no FF consumption was reported by the Xilinx synthesis tool

5.3 Loadable Bidirectional Binary Counter Architecture

Binary counter is a fundamental component of many controlpath implementations. The most desirable features of a counter are: it should be resettable, loadable, reversible (up/down counter), count-enabled, can be read on-the-fly, and be able to detect terminal count [2].

5.3.1 Proposed Counter Architecture

An up/down counter can be realized as a combination of a D-FF-based Parallel-In Parallel-Out (PIPO) register and an incrementer/decrementer, which accepts the output of the register as its input, and feedbacks its outputs to the input of the register. If the counter outputs as indicated by the FF outputs are $Q_{n-1}Q_{n-2}\ldots Q_1Q_0$, then the logic equations for the inputs of the FFs for an up counter are:

$$D_0 = \overline{Q_0} \tag{5.3}$$

$$D_i = Q_i \oplus (Q_{i-1}Q_{i-2}\ldots Q_1Q_0) \quad \text{if } i \geq 1 \tag{5.4}$$

Similarly, the logic equations for the FF inputs for a down counter are defined by:

$$D_0 = \overline{Q_0} \tag{5.5}$$

$$D_i = \overline{Q_i} \oplus (Q_{i-1} + Q_{i-2} + \cdots + Q_1 + Q_0) \quad \text{if } i \geq 1 \tag{5.6}$$

Equations (5.4) and (5.6) suggest that a wide AND and OR logic has to be realized, which can be configured using the carry chain as shown in Fig. 5.5. In this figure, larger counters can be realized by successive cascading of the "Stage 1" block. The PIPO register has been realized using the "FDRSE" Xilinx primitive which is a D-FF with synchronous reset and set and clock enable. Pipeline latency affects the correct functionality of the counters and cannot be tolerated in a counter design, as the inputs to the PIPO register come at a specific instant of time, but output values are expected to be obtained in the following clock cycle. Hence, the pipelined latches are realized using the "FDCPE" Xilinx primitive [3] which is a D-FF with clock enable and asynchronous preset and clear. These FFs are presetted if the output from the previous carry chain of the adjacent slice is high and cleared if low. For an n-bit counter, $\lceil n/4 \rceil - 1$ asynchronous pipeline stages are required.

The basic building block of this architecture is a 4-bit counter realized within a single slice. The logic functionality of accepting a new data $DATA_i$, when the "load control" signal *LD* to load external data to the FFs is high, and accepting the output from the FFs when *LD* is low, along with the XOR operation, is taken care of by the 6-input LUT configured as:

$$O_6 = (\overline{LD} \cdot Q_i + LD \cdot DATA_i) \oplus \overline{UP}/DOWN \tag{5.7}$$

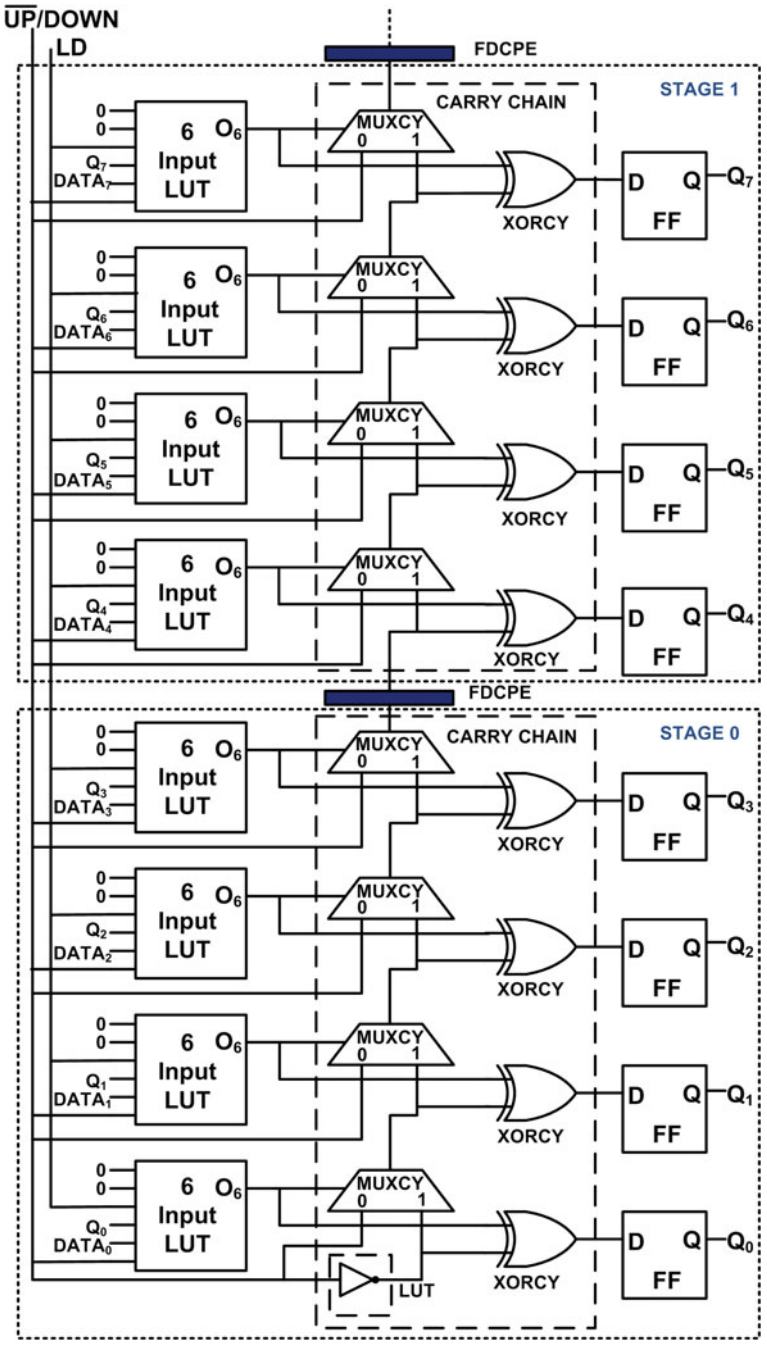

Fig. 5.5 Architecture for loadable, *up/down* counter targeted toward Xilinx Virtex-5 FPGA

where the counter counts up if $\overline{UP}/DOWN = 0$, and down if $\overline{UP}/DOWN = 1$. In other words, when the control signal $\overline{UP}/DOWN = 0$, the carry chain is configured as a wide AND gate, and when the control signal $\overline{UP}/DOWN = 1$, the carry chain is configured as a wide OR gate.

The terminal count is detected by the carry output of the most significant carry chain. As the external data to be loaded into the register is not supplied directly to the input of the FFs, but comes from the output of the incrementer/decrementer logic, the user must send the value $(x - 1)$ in case she wants to load an up counter with the value x, or send $(y + 1)$ in case she wants to load a down counter with the value y.

5.3.2 DSP Slice-Based Counter

Just like adders and multipliers, counters can also be designed using DSP48E slices. n-bit counters with $n > 48$ can be realized by cascading $\lceil n/48 \rceil$ DSP48E slices along a column, as shown in Fig. 5.6. To achieve the desired functionality, the slices have been configured as a 48-bit accumulator each by setting the attributes "OPMODE" as 0101100, and "ALUMODE" as 0000 for addition and 0011 for subtraction [4]. The additional usage information to be taken note of here is that while the counter is operating as a down counter and the user wants to load the registers with the value x, she must send the two's complement of x as the input.

Currently, *FloPoCo* (v 2.5.0) does not generate HDL description for counters.

5.3.3 Counter Implementation Results

The routing complexity for the proposed counter architecture is comparatively involved in comparison to adders, as there exists a feedback path from the output of the FFs to the input of the LUTs. Operating frequencies for both the counter

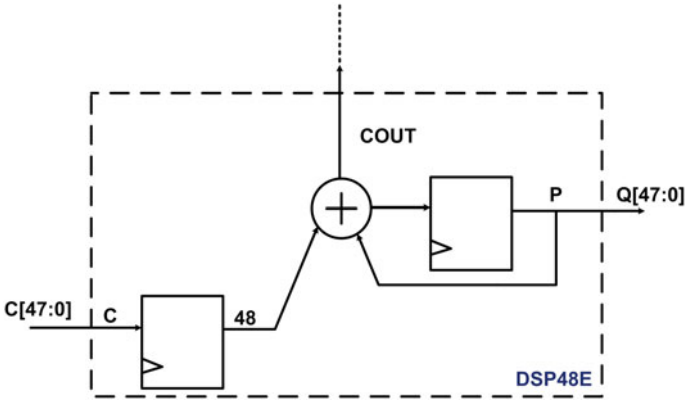

Fig. 5.6 Xilinx Virtex-5 DSP slice-based counter [4]

synthesized from behavioral description and that generated through the GUI utility deteriorate steadily with increase in the number of output bits, which is undesirable. In contrast, the proposed design shows only a relatively minor degradation in performance with increase in the number of counter output bits. The proposed design also outperforms the non-pipelined behavioral counter for most cases in terms of PDP and the fabric counter gave the worst PDP performance. *FloPoCo* however does not support counter implementations till its latest release (Table 5.3).

5.4 Summary

All our implementation results for the integer comparator and loadable bidirectional counter indicate superior operand width scalability with respect to frequency. It can thus again be concluded that constraining the placement of the circuit building blocks

Table 5.3 Counter implementation results

Operand width	Design style	Freq (MHz)	Power delay product (pJ)	#FF	#LUT	#DSP	#Slice
32	Behavioral counter	504.80	48.67	32	49	0	15
	Fabric counter (IP Core)	536.19	50.52	32	47	0	14
	DSP slice counter[a]	550.00	35.65	0	0	1	0
	Proposed counter[a]	**587.89**	**37.30**	**39**	**41**	**0**	**17**
48	Behavioral Counter	400.32	43.39	48	70	0	29
	Fabric counter (IP core)	427.35	59.09	48	69	0	30
	DSP slice counter[a]	550.00	40.80	0	0	1	0
	Proposed counter[a]	**567.21**	**57.10**	**59**	**61**	**0**	**25**
64	Behavioral counter	367.51	53.33	64	94	0	32
	Fabric counter (IP core)	387.59	67.47	64	93	0	39
	DSP slice counter[a]	500.00	50.46	0	0	2	0
	Proposed counter[a]	**565.61**	**59.85**	**79**	**81**	**0**	**33**
96	Behavioral counter	292.65	89.08	96	139	0	46
	Fabric counter (IP core)	298.95	84.39	96	137	0	43
	DSP slice counter[a]	500.00	61.64	0	0	2	0
	Proposed counter[a]	**562.75**	**72.59**	**119**	**121**	**0**	**48**
128	Behavioral counter	231.70	121.02	128	186	0	64
	Fabric counter (IP core)	246.49	119.03	128	184	0	67
	DSP slice counter[a]	500.00	79.30	0	0	3	0
	Proposed counter[a]	**563.70**	**111.12**	**159**	**161**	**0**	**64**

[a] In our proposed counter design with constrained placement, an n-bit counter had $\lceil n/4 \rceil - 1$ pipeline stages. For the DSP slice-based counter, no pipelining was done if $n < 48$, while if $n > 48$, the number of pipeline stages internal to the DSP slices were $\lceil n/48 \rceil - 1$, and no FF resource consumption was reported by the Xilinx synthesis tool

is crucial in improving the circuit performance, and the regularity of the proposed designs is sufficient to automate their design. In the next chapter, we shall explore the implementation of Cellular Automata-based Pseudorandom Sequence Generator on FPGAs, using the methodology proposed so far.

References

1. Perri, S., Zicari, P., Corsonello, P.: Efficient absolute difference circuits in Virtex-5 FPGAs. In: Proceedings of the15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 309–313 (2010)
2. Stan, M.R., Tenca, A.F., Ercegovac, M.D.: Long and fast up/down counters. IEEE Trans. Comput. **47**(7), 722–735 (1998)
3. Xilinx Inc.: Virtex-5 libraries guide for HDL designs, UG621 (v11.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex5_hdl.pdf Cited 6 Sep 2009
4. Xilinx Inc.: Virtex-5 FPGA xtremeDSP design considerations user guide, UG193 (v3.5). http://www.xilinx.com/support/documentation/user_guides/ug193.pdf Cited 26 Jan 2012

Chapter 6
Compact FPGA Implementation of Linear Cellular Automata

Abstract Cellular Automata (CA) have been proposed as popular VLSI primitives owing to their regular, cascadable structure, and supposedly local interconnects. However, rather surprisingly, the published literature does not stress that the regularity and locality of interconnects is often more *logical* rather than being of *physical* nature, and requires proper design methodologies to harness the advantage of CA in practical circuits. We address this issue with a case study of a one-dimensional (1-D) CA, and develops a methodology for the physical realization of such circuits. The main idea is to make optimal use of the underlying architecture, especially the hardware logic resources available in the FPGA *slices*, coupled with direct primitive instantiation and constrained placement of the logic elements.

6.1 Introduction

The regular, modular and cascadable structure of cellular automata (CA) with only local neighborhood dependence of the cells are the attractive features that makes it suitable for VLSI implementation [7, 8, 12]. Although theoretically CAs are attractive enough, in practice, their implementation on FPGAs often turn out to be inefficient, because usually the user has limited control on the inference of logic elements, along with their placement and routing. Moreover, the CAD algorithms that perform these steps in a FPGA typically have some probabilistic metaheuristic components (e.g., *simulated annealing*) [2], which make the performance of a given design when implemented on a given FPGA platform somewhat unpredictable. Hence, it cannot be guaranteed that connected CA cells in logical proximity would actually have a very short length of interconnect between them, when implemented physically.

In the past, CA circuits have been proposed for test pattern generation and construction of Built-In Self-Test (BIST) structures within VLSI chips [14]. Since these structures are based on simple rules and amenable to fast designs, high-performance hardware implementation of CA algorithms has always been an important research topic over the years. Previously, when the current advanced FPGA families were not available, a new FPGA architecture supporting 5-input 3-output AND-XOR-based logic blocks along with an efficient multilevel AND-XOR logic minimization scheme

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_6

was proposed for efficient implementation of Cellular Automata Array (CAA) [6]. In [10], the authors demonstrated faster implementation of CA on FPGA hardware, compared to optimized software implementation by achieving a speedup in the range of 14–19. A methodology for VLSI implementation of CA algorithms, including an automatic translation scheme from CA algorithms to the corresponding VHDL was proposed in [15]. FPGA-based CA implementation was also reported in [16]. However, to the best of our knowledge, there has been no reported work regarding the principles and design philosophy for efficient low-level implementation of CA on modern families of actual FPGAs, aiming to map the CA structures optimally to the native architecture of the FPGA.

The rest of the chapter is organized as follows. In Sect. 6.2, we discuss the general structure of CA, and introduce the relevant CA terminology. In Sect. 6.3, we discuss the principles of adapting the CA structure to the native FPGA architecture. The CA implementation results and related observations have been discussed in Sect. 6.4. We conclude in Sect. 6.5.

6.2 Preliminaries on Cellular Automata

Structurally, CA consists of a collection of identical building blocks termed as "cells," where each "cell" is a combination of its characteristic combinational logic (CL), and a sequential element (e.g., a D-FF) to hold its state, as shown in Fig. 6.1. The collection of all the states of the cells of a CA is defined as its *state*. Usually, the next state of a particular cell is a function of its own current state, and the current state of its two immediate neighbors—such a dependency is called a *three-neighborhood dependency*. The particular function describing this dependency is called the *rule* of the CA [9]. We have considered a common CA variant called "Null Boundary CA," where the terminal cells on either side are connected to logic-0 permanently.

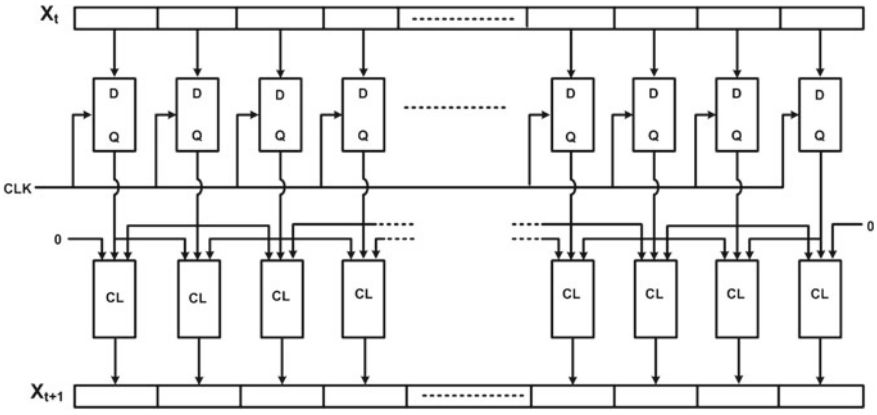

Fig. 6.1 The Cellular Automata structure

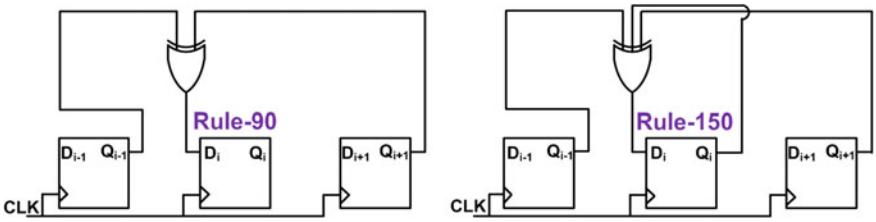

Fig. 6.2 Combinational logic for cells corresponding to *rule-90* and *rule-150*

Let $X_t = \{q_0(t), q_1(t), \ldots, q_{n-1}(t)\}$ denote the state of an n-cell CA at instance t, where i denotes the position of an individual cell in the one-dimensional array of cells, t denotes the time step, $q_i(t)$ denotes the output state of the ith cell at the tth instant of time. Then, for a three neighborhood dependency, the next state of the i-th cell is given by $q_i(t+1) = f(q_{i-1}(t), q_i(t), q_{i+1}(t))$ where $f()$ denotes the *rule of the CA* [5, 9], which is inherently a Boolean function that can be expressed in the form of a truth table. The decimal equivalent of the output bitstring as written in the truth table is conventionally called the *rule number* for the cell. For example, the next state logic equations for *rule-90* and *rule-150* [5] CAs are given in (6.1) and (6.2) respectively, with their circuit representations depicted in Fig. 6.2.

$$\text{Rule-90:} \quad q_i(t+1) = q_{i-1}(t) \oplus q_{i+1}(t) \tag{6.1}$$

$$\text{Rule-150:} \quad q_i(t+1) = q_{i-1}(t) \oplus q_i(t) \oplus q_{i+1}(t) \tag{6.2}$$

where $q_i = 0$ if $i < 0$ or $i \geq n$. Table 6.1 shows the next states computed according to rules 90 and 150. The top row shows all the possible configurations of the left, self, and right cells at instant t. The states at the instant of time $(t+1)$ are computed according to the rules.

If the CA is *linear*, the combinational logic functions $f()$ involves only XOR logic. A CA having a combination of XOR and XNOR logic is called an *additive* CA, whereas for *nonlinear* or *non-additive* CA, $f()$ involves AND/OR logic [11]. If all CA cells obey the same rule, then it is termed as *uniform* CA, else it is a *hybrid* CA. *Linear* CAs can also be described by their *characteristic polynomials*. From a given polynomial, we can efficiently determine the structure of the corresponding CA [4]. The corresponding CA by convention is usually described by a string of 0's and 1's, where, for example, '1' refers to rule-150 and '0' refers to rule-90. Our proposed methodology can efficiently implement two-rule *linear*, *additive*, *uniform*, and *hybrid* CAs.

Table 6.1 Rules 90 and 150

Neighborhood state:	111	110	101	100	011	010	001	000	
Next state:	0	1	0	1	1	0	1	0	(rule 90)
Next state:	1	0	0	1	0	1	1	0	(rule 150)

Table 6.2 Additive CA rules [5]

Complement		Dependency	Noncomplement	
Rule	Logic function		Rule	Logic function
195	$\overline{q_{i-1}(t) \oplus q_i(t)}$	Left and self	60	$q_{i-1}(t) \oplus q_i(t)$
165	$\overline{q_{i-1}(t) \oplus q_{i+1}(t)}$	Left and right	90	$q_{i-1}(t) \oplus q_{i+1}(t)$
153	$\overline{q_i(t) \oplus q_{i+1}(t)}$	Self and right	102	$q_i(t) \oplus q_{i+1}(t)$
105	$\overline{q_{i-1}(t) \oplus q_i(t) \oplus q_{i+1}(t)}$	Left and self and right	150	$q_{i-1}(t) \oplus q_i(t) \oplus q_{i+1}(t)$
85	$\overline{q_{i+1}(t)}$	Right	170	$q_{i+1}(t)$
51	$\overline{q_i(t)}$	Self	204	$q_i(t)$
15	$\overline{q_{i-1}(t)}$	Left	240	$q_{i-1}(t)$

On minimization, the truth tables for the rules 15, 51, 60, 85, 90, 102, 105, 150, 165, 170, 195, 204, and 240 result in the logic functions noted in Table 6.2, where $q_i(t)$ denotes the state of the ith CA cell at the tth time instant, q_{i-1} and q_{i+1} refers to the state of its left and right neighbors.

6.3 Adapting CA to the Native FPGA Architecture

Before the availability of the modern Xilinx FPGA families, the technology mapping for CA-based circuits was an issue for which researchers proposed special FPGA architectures [6] to efficiently realize XOR/XNOR dominated functions especially using an AND-OR programmable logic fabric. However, with the advent of modern FPGA families such as Virtex-6 that have 6-input LUTs which can map any arbitrary six (or less) input functions, the problem has been simplified to a large extent. We take advantage of this availability of large LUTs in the proposed design methodology.

Packing is a key step in the FPGA tool flow that is tightly integrated with the boundaries between *logic synthesis*, *technology mapping*, and *placement* [1]. For Virtex-5 FPGAs, the packing technique targets the dual-output LUTs to achieve area efficiency by exploring the feasibility of packing two logic functions into a single LUT. This is possible whenever the two logic functions have no more than five distinct variables. In such cases, a more efficient mapping of the design is expected, culminating into shorter interconnect wirelength, which in turn results in lesser critical path delay. However, our implementation results for Virtex-6 family of FPGAs, which is an advanced and modified version of Virtex-5, clearly show that in spite of the methodology adopted by the common FPGA CAD tools, the packing step remains challenging, which often results in inefficient circuit implementations.

Consider an 1-D CA where the next state of a particular cell depends only on itself, or on one or both of its two immediate neighbors. It is easy to deduce that in such a case, any two adjacent cells can have a maximum of four distinct inputs.

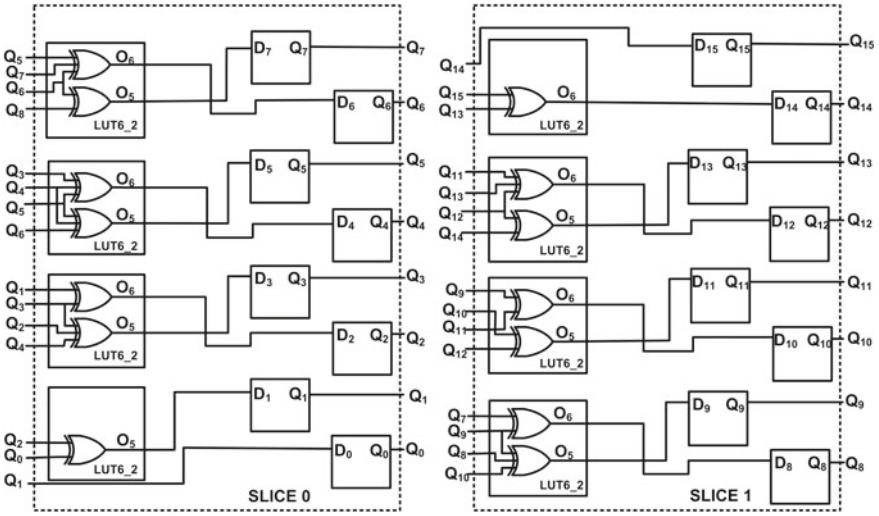

Fig. 6.3 Structure of a 16-bit 1-D linear CA for the (primitive) polynomial $x^{16}+x^5+x^3+x^2+1$ (or the equivalent *hybrid* rule $<$ 0001111001001000 $>$) mapped on Xilinx Virtex-6 FPGAs, following the proposed design philosophy [13]

In such a situation, the next state logic for any two cells of a CA can be packed into a single LUT. Since Virtex-6 architectures facilitate registering of both the LUT outputs to two FFs present in the same slice as that of the LUT, we can achieve a compact FPGA realization of the architecture [13]. The architecture for a 16-cell 1-D linear maximal length CA for the (primitive) polynomial $x^{16}+x^5+x^3+x^2+1$ (or the equivalent *hybrid* rule $<$ 0001111001001000 $>$) [3] is shown in Fig. 6.3. Thus, in this process, for an n-cell maximal length CA architecture, $\lceil n/8 \rceil$ Virtex-6 FPGA slices are required.

6.4 CA Implementation Results

The circuits described in Sect. 6.3 were implemented on Xilinx Virtex-6 FPGA, device family XC6VLX550T, package FF1760 and speed grade -2 using the *Xilinx ISE* (v 12.4) design environment. Cellular Automata structures with polynomials of the order 32, 48, 64, 80 and 96 (whose corresponding CA is given in Table 6.3) were implemented by two different techniques—the proposed design methodology (using *FlexiCore*), and by RTL coding followed by unconstrained automatic logic synthesis by ISE. The implementation results were compared with respect to their frequency of operation, PDP, and hardware resource requirement (FFs, LUTs and slices), and are tabulated in Table 6.4. The polynomials are from [17] and, for example, the entry in the polynomial field of Table 6.4, 32 28 27 1 0, represents the polynomial $x^{32}+x^{28}+x^{27}+x+1$.

Table 6.3 Polynomial and their corresponding CA [3]

Polynomial	CA
32, 28, 27, 1 ,0	00001100010001110000110000000110
48, 28, 27, 1, 0	010100000001111101000100101101111001111000001010
64, 4, 3, 1, 0	1001110101001101111011011001100100111001101101111011001010111001
80, 38, 37, 1, 0	01010110010000100000101000110011101111011110101011011101111000000100001001101010
96, 49, 47, 2, 0	11110111101100000010001001101101110101100100011011100011010100110000000000001101011000100101001 0

Table 6.4 CA implementation results [13]

Polynomial	Mode of implementation	Freq (MHz)	Power-delay product (pJ)	#FF	#LUT	#Slice
32, 28, 27, 1, 0	RTL design	1014.20	31.61	32	30	8
	Proposed design	**1103.75**	**31.15**	**32**	**16**	**4**
48, 28, 27, 1, 0	RTL design	320.41	37.36	48	46	12
	Proposed design	**1089.32**	**40.26**	**48**	**24**	**6**
64, 4, 3, 1, 0	RTL design	361.40	43.80	64	64	1
	Proposed design	**1083.42**	**52.92**	**64**	**32**	**8**
80, 38, 37, 1, 0	RTL design	414.08	64.05	80	78	20
	Proposed design	**976.56**	**59.08**	**80**	**40**	**10**
96, 49, 47, 2, 0	RTL design	361.79	70.92	96	96	24
	Proposed design	**908.27**	**62.59**	**96**	**48**	**12**

It was observed that for an RTL description of the CA circuit, the Xilinx Post Place and Route results indicate that double the FPGA area is getting consumed than what a compact realization should have taken. The speed of operation for the CA circuits also drastically reduces as the order of the polynomial is steadily increased.

6.5 Summary

From the partial floorplan (physical) view of the mapped CA circuits in Fig. 6.4, it can be observed, that for the RTL-based design, the inferred logic elements (shown using dark shades) are not compactly packed within each slice and the logically related slices are placed distant apart in a random, haphazard fashion throughout a large FPGA area. On the contrary, for the proposed design, the floorplan shows a very compact realization of the circuit by taking advantage of the dual-output nature of the LUTs and by placing the logic elements having proximity of bit-indices, in the closest physical proximity on the target FPGA fabric. Thus, we can conclude that FPGA CAD tools, for designs derived from RTL design descriptions, cannot exploit the local

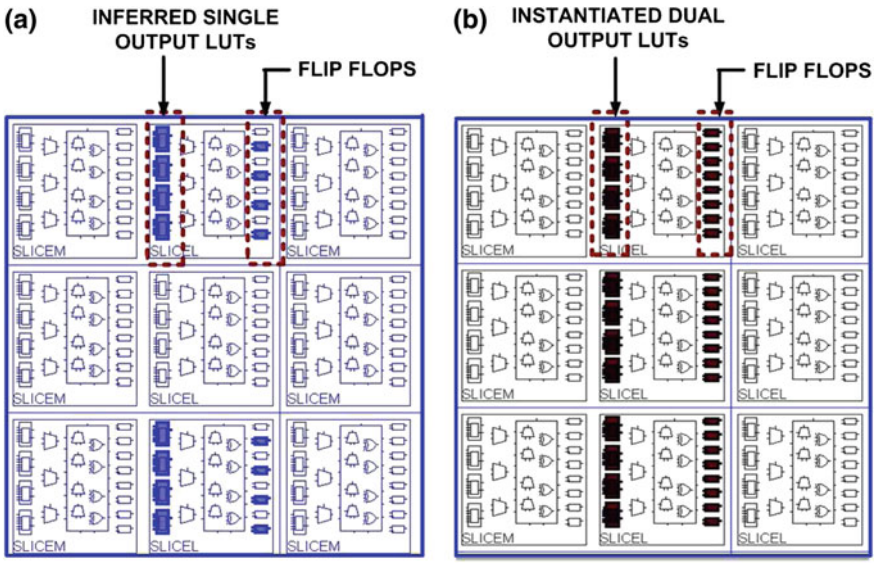

Fig. 6.4 Partial floorplan views for the CA circuits mapped onto the Virtex-6 FPGA fabric. **a** Partial floorplan view for RTL design of CA circuit. **b** Partial floorplan view for proposed desgin of CA circuit

neighborhood property of CAs, and ultimately result in long interconnects leading to serious performance degradation. This is undesirable as speed is critical from the hardware accelerator point of view in building high-performance cryptographic cores [14]. The higher resource requirement also shows that the mapping of the circuit on the FPGA fabric is not optimal for the RTL-derived implementations.

In the next chapter, we shall study the features of the *FlexiCore* design automation tool and its CAD flow that was developed by us, along with two case studies of system design examples where a larger system is built from smaller subsystems or circuits which are supported by the *FlexiCore* platform and study the associated performance advantages obtained.

References

1. Ahmed, T., Kundarewich, P.D., Anderson, J.H.: Packing techniques for Virtex-5 FPGAs. ACM Trans. Reconfigurable Technol. Syst. (TRETS) **2**(18), 18:24 (2009)
2. Areibi, S., Grewal, G., Banerji, D., Du, P.: Hierarchical FPGA placement. Can. J. Electr. Comput. Eng. **32**(1), 53–64 (2007)
3. Cattell, K. Muzio, J.: Technical Report: Tables of linear cellular automata for minimal weight primitive polynomials of degrees upto 300. Issue: 163. University of Victoria (B.C.) Department of Computer Science (1991)
4. Cattell, K., Muzio, J.C.: Synthesis of one-dimensional linear hybrid Cellular Automata. IEEE Trans. Comput.-Aided Des. Integ. Circuits and Syst. **15**(3), 325–335 (1996)

5. Chaudhuri, P.P., Chowdhury, D.R., Nandi, S., Chattopadhyay, S.: Additive Cellular Automata theory and its application. IEEE Comput. Soc. Press, **1** (1997)
6. Chattopadhyay, S., Roy, S., Chaudhuri, P.P.: Technology mapping on a multi-output logic module built around Cellular Automata array for a new FPGA architecture. In: Proceedings of the 8th International Conference on VLSI Design, pp. 57–62 (1995)
7. Chowdhury, D.R., Chaudhuri, P.P.: Architecture for VLSI design of CA based byte error correcting code decoders. In: Proceedings of the 7th International Conference on VLSI Design, pp. 283–286 (1994)
8. Chowdhury, D.R., Gupta, I.S., Chaudhuri, P.P.: CA-based byte error-correcting code. IEEE Trans. Comput. **44**(3), 371–382 (1995)
9. Das, A.K., Ganguly, A., Dasgupta, A., Bhawmik, S., Chaudhuri, P.P.: Efficient characterization of Cellular Automata. IEE Proc. E Compu. Digital Techn. **137**(1), 81–87 (1990)
10. Halbach, M., Hoffmann, R.: Implementing Cellular Automata in FPGA logic. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, pp. 258–262 (2004)
11. Mukhopadhyay, D.: Group properties of non-linear Cellular Automata. J. Cellular Autom. **5**(1–2), 139–155 (2010)
12. Nandi, S., Rambabu, C., Chaudhuri, P.P.: A VLSI Architecture for Cellular Automata based reed-solomon decoder. In: Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN), pp. 158–165 (1999)
13. Palchaudhuri, A., Chakraborty, R.S., Salman, M., Kardas, S., Mukhopadhyay, D.: Highly Compact Automated Implementation of Linear CA on FPGAs. In: Cellular Automata - 11th International Conference on Cellular Automata for Research and Industry (ACRI), Series Lecture Notes in Computer Science, vol. 8751, pp. 388–397 (2014)
14. Sarkar, P.: A brief history of Cellular Automata. ACM Comput. Surv. (CSUR) **32**(1), 80–107 (2000)
15. Sirakoulis, G.C., Karafyllidis, I., Thanailakis, A., Mardiris, V.: A methodology for VLSI implementation of Cellular Automata algorithms using VHDL. Adv. Eng. Softw. **32**(3), 189–202 (2000)
16. Torres-Huitzil, C., Delgadillo-Escobar, M., Nuno-Maganda, M.: Comparison between 2D Cellular Automata based pseudorandom number generators. IEICE Electron. Express. **9**(17), 1391–1396 (2012)
17. Bardell, P.H., McAnney, W.H., Savir, J.: Built-In Test for VLSI: Pseudorandom Techniques. Wiley, New York (1987)

Chapter 7
Design Automation and Case Studies

Abstract All the architectures proposed in the previous chapters have been realized using the bit-sliced design paradigm. The architectures are very regular in their structures, thereby serving as a motivation to automate the generation of the arithmetic circuit descriptions for the target FPGA platform. In this chapter, we will introduce the proposed CAD tool for design automation named *FlexiCore*. We also present two relevant case studies comprising of multiple modules, whose HDL and placement constraints can be generated using *FlexiCore*.

7.1 Introduction

The design of all the circuits described in the previous chapters has been automated using a CAD tool developed by us. We call the CAD tool *FlexiCore*, in short for "**Flexi**ble Arithmetic Soft **Core** Generator". Presently, *FlexiCore* supports Hybrid RCA, Absolute Difference Circuit, Integer Multiplier, Integer Squarer, Universal Shift Register, Comparator, Loadable Bidirectional Binary Counter, and Cellular Automata Circuits as discussed in Chaps. 4–6. It is *flexible* in the sense that the operand widths for the mapped circuits can be varied, and the CAD tool allows partial control to the user over the placement of the circuits on the FPGA fabric. The tool is developed in JAVA, and includes a simple GUI. The CAD software executable is invoked from the TCL command prompt in-built in Xilinx ISE using a top-level TCL script.

The rest of the chapter is organized as follows. In Sect. 7.2, we give details of the *FlexiCore* design automation tool developed by us that generates the HDL and placement constraints for arithmetic and CA circuit descriptions. In Sect. 7.3, we present the case studies for two multimodule designs—a *GCD calculator* and a *matrix multiplication circuit*, and demonstrate the effectiveness of the proposed design methodology. We conclude in Sect. 7.4.

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_7

7.2 The FlexiCore CAD Tool

The *FlexiCore* design flow is depicted in Fig. 7.1. Here, the top-level script invokes a GUI which displays the list of circuits currently supported by *FlexiCore*, and prompts the user to enter (in the GUI entry fields) the circuits (along with their operand widths and whether the user wants pipelined/non-pipelined version), for which the user wants constrained placement-based high performance design. The GUI for designing a pipelined 64-bit adder is shown in Fig. 7.2. The user can also optionally enter the starting coordinate for the entire constrained placement exercise. If this is not provided, *FlexiCore* determines the feasible starting coordinate from the existing Xilinx proprietary project constraints file called the "User Constraints File" (.ucf).

After the user enters her options, *FlexiCore* examines the feasibility of placement of the selected building blocks on the FPGA fabric, in a regular fashion as described in the previous chapters, with the starting coordinate entered by the user as origin,

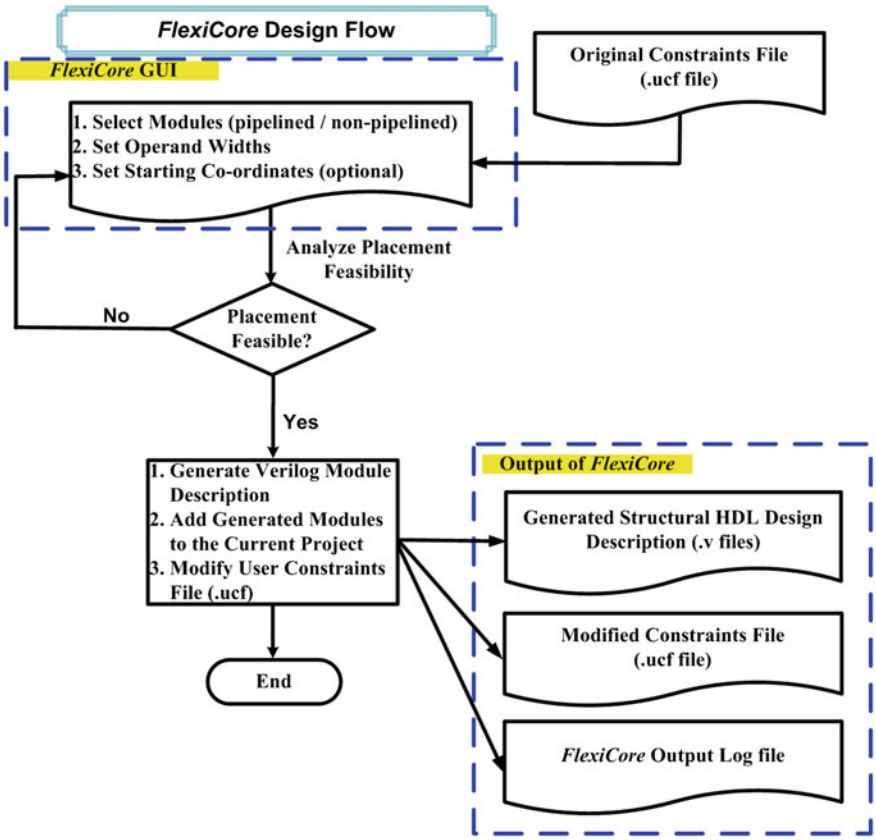

Fig. 7.1 The *FlexiCore* design flow for arithmetic circuits

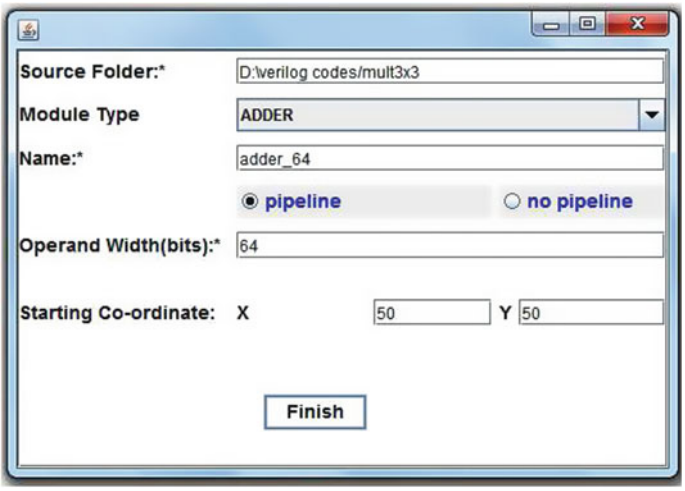

Fig. 7.2 *FlexiCore* GUI for arithmetic circuits

or the starting coordinate inferred. It takes into consideration the existing placement constraints, if any, in the project constraints file. If the placement is deemed feasible, *FlexiCore* performs the following:

- Generates the Verilog module descriptions for the selected circuit building blocks, and adds the files to the current project. Care is taken to ensure that no hardware primitive instance on the FPGA is used more than once in building the high-performance building blocks. Pipeline registers as required, are automatically inserted. At present, *FlexiCore* supports two options—either a maximally pipelined implementation (optimized for Virtex-5 platform), or a purely combinational circuit. We expect to support variable latency circuits in future.
- Modifies the project .ucf, by adding the placement constraints for the generated high-performance circuit building blocks.
- Creates a log file to provide the user with all the necessary information about the generated modules.

If *FlexiCore* fails to find a feasible placement configuration, it reports it to the user and again prompts her to enter a (reduced) number of building blocks, or a different starting coordinate. Note that the situation where *FlexiCore* fails to find a feasible placement rarely arises, given the large availability of resources on a Virtex family FPGA. We did not find any such scenario with our real-life design test cases.

To accommodate the Cellular Automata circuits into the CAD tool for their automatic generation, a provision has been kept for the user to invoke the GUI, which displays all the list of rules (see Table 6.2) corresponding to which equivalent CA circuits can be generated, and prompts the user to enter the following fields: the two CA rule numbers, their corresponding encoding of 0 and 1, and the hybrid CA rule comprising of a string of 0's and 1's. The CAD tool interprets the string by reading

two bits at a time, calculates the *truth table* of the dual-output LUTs appropriately for realizing the next state logic for the CA cells, and instantiates the required FPGA logic elements in the HDL code. The remaining design flow remains to be the same as for arithmetic circuits. The design flow, particularly for CA circuits, is shown in Fig. 7.3. An example snapshot of the *FlexiCore* generated log file for the CA rule "00001100010001110000110000000110" (as entered in the GUI of *FlexiCore* shown in Fig. 7.4) corresponding to rule-90 (encoded as '0') and rule-150 (encoded as '1') is shown in Fig. 7.5. The computational complexity of *FlexiCore* can be analyzed as a function of the input size of the operands. For adders, absolute difference circuits, universal shift registers, comparators, and counters, the computational complexity is $O(n)$, where n is the operand width of the real-time data inputs. However for multipliers and squarers, the computational complexity is a quadratic function of the operand width n, i.e., of $O(n^2)$. For CA circuits, the complexity is $O(n)$, where n is the order of the primitive polynomial corresponding to the CA rule which is entered in the GUI.

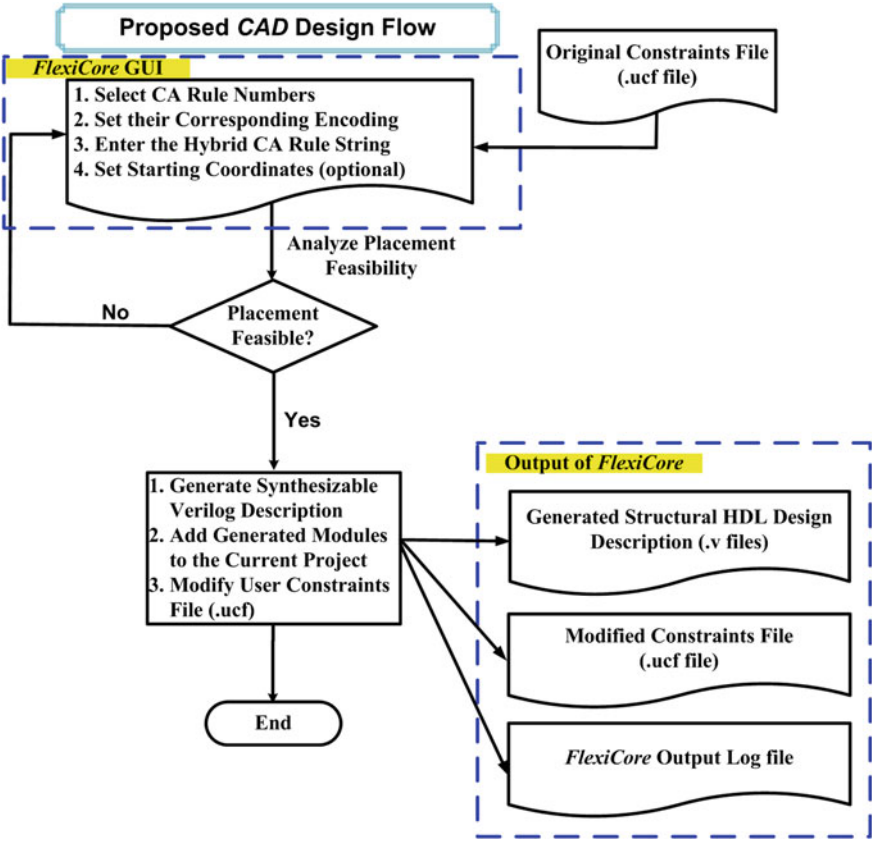

Fig. 7.3 The *FlexiCore* design flow for CA circuits

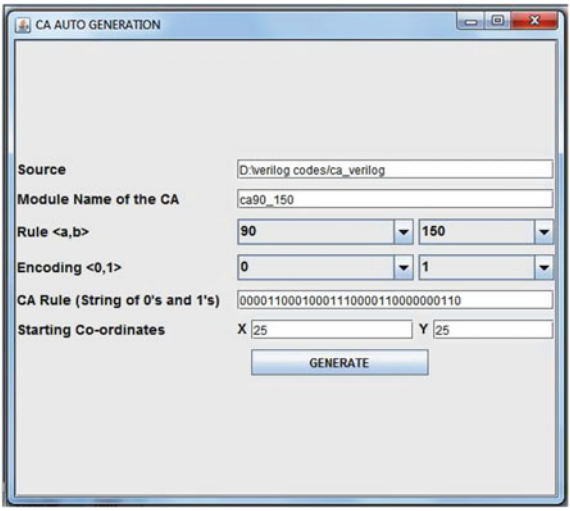

Fig. 7.4 *FlexiCore* GUI for CA circuits

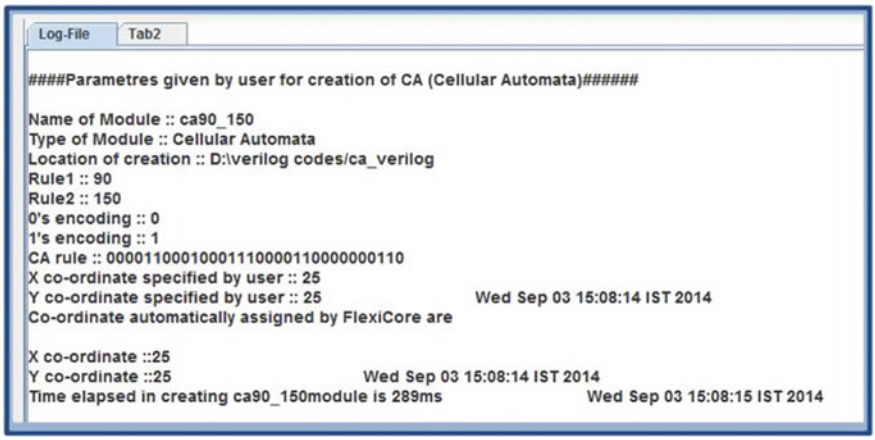

Fig. 7.5 *FlexiCore* log file for CA circuits

7.3 Case Studies

We shall present case studies of two arithmetic circuits—a Greatest Common Divisor (GCD) calculator circuit and a Distributed Arithmetic (DA)-based Matrix Multiplication circuit. Both the architectures use several of the arithmetic building blocks supported by the *FlexiCore* platform.

Algorithm 1: GCD Calculation Algorithm

```
Input: 2 unsigned integers: P and Q.
Output: S : GCD of P and Q
Rem(P, Q): Remainder when P is divided by Q
abs(P − Q): Absolute difference of P and Q
min(P, Q): Minimum of P and Q
Computation_Over_Flag ← 0, R ← 0
begin
    while P ≠ Q do
        if (Rem(P,2)==0) then
            P ← P/2;
            if (Rem(Q,2)==0) then
                Q ← Q/2;
                R ← R + 1;
        else
            if (Rem(Q,2)==0) then
                Q ← Q/2;
            else
                P ← abs(P − Q);
                Q ← min(P, Q);
S ← P ∗ (2^R);
Computation_Over_Flag ← 1;
```

7.3.1 GCD Calculator Circuit

We now present the complete architecture and implementation results for a *Greatest Common Divisor* (GCD) computation circuit. This particular architecture was chosen because it utilizes several of the building blocks currently provided by the *FlexiCore* platform such as the absolute difference circuit, counter and a barrel shifter. The architecture has been derived from the Binary GCD algorithm [10] which has been explained in Algorithm 1. This algorithm uses simpler arithmetic operations than the conventional Euclidean GCD algorithm as it replaces complex operations such as division and multiplication with division and multiplications by powers of two. Assuming a binary representation of integers, such operations can be very efficiently implemented using "right shift" and "left shift" operations, comparisons, and subtraction [3], thereby making it suitable for hardware implementation .

7.3.1.1 Proposed Architecture of Binary GCD Circuit

The architecture for the algorithm at the block diagram level has been shown in Fig. 7.6. We present two *multifunction* registers P and Q which are loaded in accordance with the control signals: active low load control signal $\overline{INIT}$ which accepts two

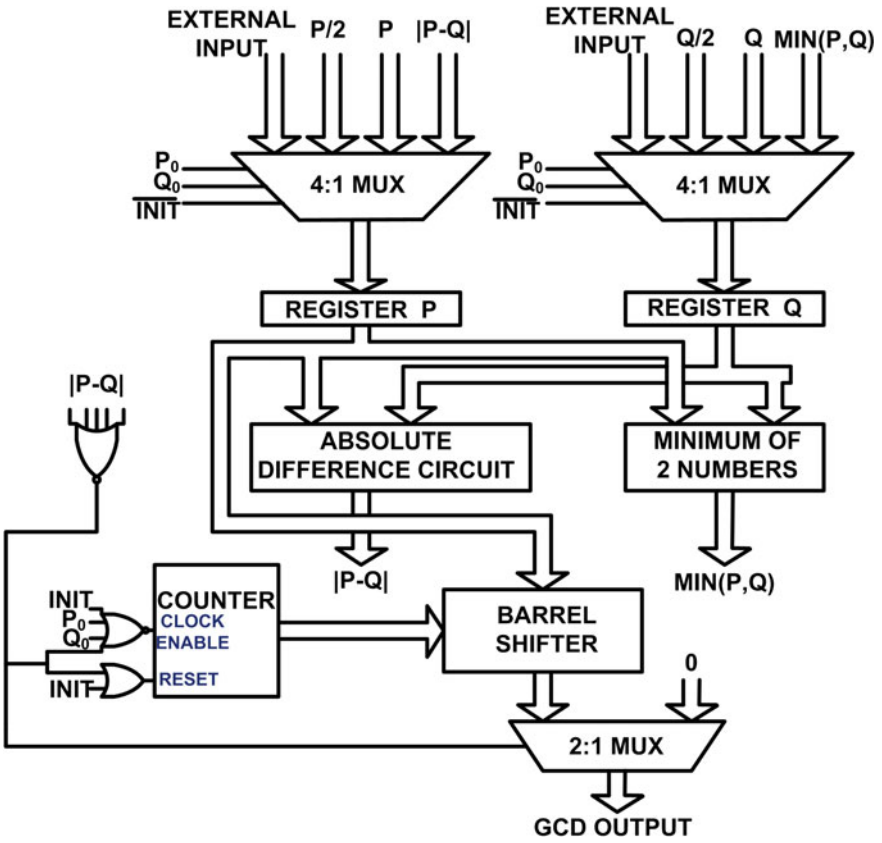

Fig. 7.6 Overall architecture of the GCD computation circuit

unsigned integers as inputs whose GCD has to be computed, and LSBs of registers P and Q as depicted in Table 7.1. The multifunction registers and their associated combinational function logic which is a nonstandard representation of a 4:1 multiplexer,

Table 7.1 Function table for the multifunction registers and counter

Control/select signals			Registers		Counter
$\overline{INIT}$	P_0	Q_0	P	Q	R
0	X	X	Load	Load	0
1	0	0	$P/2$	$Q/2$	$R+1$
1	0	1	$P/2$	Q	R
1	1	0	P	$Q/2$	R
1	1	1	$\|P-Q\|$	$min(P, Q)$	R

have been mapped intelligently to the 6-input LUTs and wide-function multiplexers *MUXF*7 available in each slice of the FPGA as shown in Fig. 7.7 to ensure compact implementation.

The absolute difference circuit has been pipelined using the "FDCPE" Xilinx primitive [13] which is a D-FF with clock enable and asynchronous preset and clear. These FFs are presetted if the output from the previous carry chain of the adjacent slice is high and cleared if low.

An intermediate output *A_l_B* (which decided whether to compute $A - B$ or $B - A$) of the absolute difference circuit serves as a select line to the multiplexer which outputs the minimum of two numbers. This architecture to compute the minimum of two numbers has been realized using dual-output LUTs as shown in Fig. 7.8. The counter keeps track of the number of left shifts to be applied to the final contents of the *P* register after the last iteration. We realize such a shifter in hardware using a barrel shifter. The schematic of the implementation structural details of the barrel shifter using dual-output LUTs is shown in Fig. 7.9. Each dual-output LUT realizes the functionality of two adjacent multiplexers present in the same row or stage. The barrel shifter is composed of a number of stages as decided by the shift amount or the width of the unsigned integers given as inputs, where stage i ($i \geq 0$) can implement a $2^i/0$ bit shift. Thus, the data to be shifted is given to the data inputs

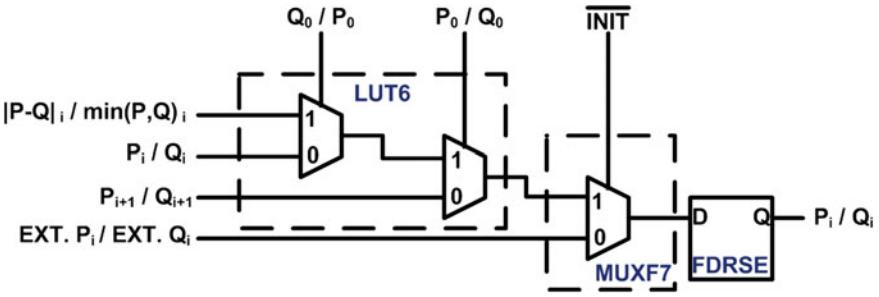

Fig. 7.7 Multifunction register

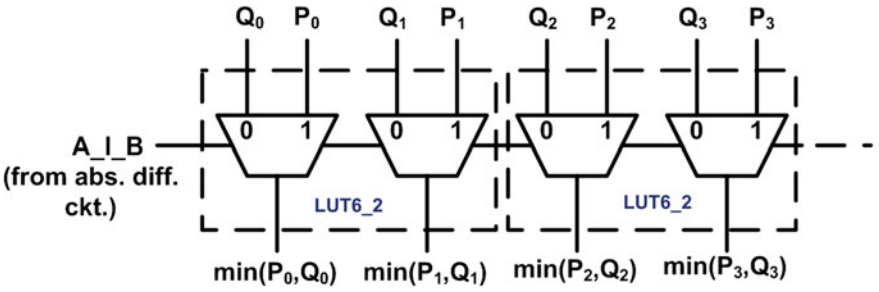

Fig. 7.8 Circuit to compute minimum of two numbers

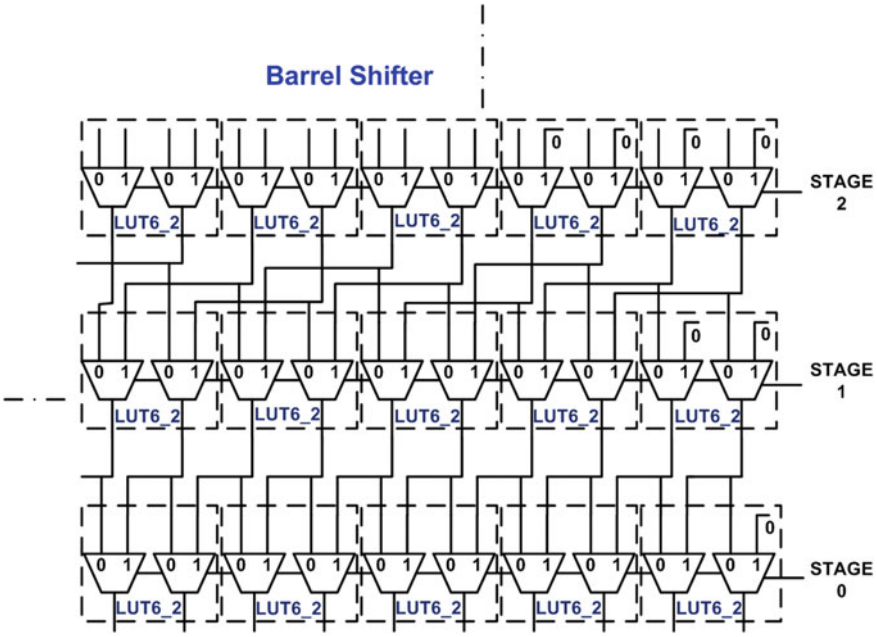

Fig. 7.9 LUT-level implementation of the barrel shifter

of the multiplexers, whereas, the amount of left shift is given as input to the select lines of the multiplexers. The final output of the barrel shifter gives the GCD of two numbers.

7.3.1.2 GCD Implementation Results

The GCD computation circuit for 32-bit operands was implemented on the Xilinx Virtex-5 FPGA using two approaches: behavioral Verilog modeling, and second using constrained arithmetic circuit descriptions generated by FlexiCore. Results are tabulated in Table 7.2, where the two input operands are 70 and 100. The results clearly indicate that using the second approach, the designer can achieve a higher frequency and lower PDP value with considerable lesser amount of hardware resources. This was achievable due to an optimized realization of the absolute difference circuit, minimum detector circuit, and barrel shifter by exploiting the carry chains and LUTs efficiently. The intelligent combination of usage of the LUT coupled with the wide-function multiplexer for realizing the multifunction register leads to significant hardware overhead reduction. Also the entire realization of the GCD circuit has been done following the constrained placement exercise leading to the overall superior performance.

Table 7.2 Implementation results for a 32-bit GCD circuit (operands: 100 and 70)

Design style	Freq (MHz)	Power-delay product (pJ)	#FF	#LUT	#Slice
Behavioral modeling	160.49	745.85	69	356	208
Primitive instantiation	**214.73**	**508.87**	**87**	**298**	**93**

7.3.2 Distributed Arithmetic-Based Matrix Multiplication Circuit

Distributed Arithmetic (DA) is an important FPGA technology as it replaces explicit multiplications that limit the speed of operation, by ROM lookups which is an efficient technique to implement on FPGAs. The most often encountered form of computation in DSP is a *sum of products* which is executed most efficiently by DA [11]. It is used to design bit-level architectures for vector–vector multiplications. Each word in the vector is represented as a binary number and the multiplications are reordered and mixed in such a way that the arithmetic becomes "distributed" through the structure [9]. In this section, we shall discuss the architecture for matrix multiplication using DA.

Consider two matrices A and B of dimensions $p \times m$ and $m \times n$ which are to be multiplied giving matrix C of dimension $p \times n$:

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{p1} & c_{p2} & \cdots & c_{pn} \end{pmatrix}_{p\times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \cdots & a_{pm} \end{pmatrix}_{p\times m} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}_{m\times n}$$

where,

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1m}b_{m1}$$
$$c_{pn} = a_{p1}b_{1n} + a_{p2}b_{2n} + \cdots + a_{pm}b_{mn}$$

In [2], authors had proposed bit-level systolic array architecture for implementation of matrix product using a serial–parallel *Baugh–Wooley multiplier*. An FPGA-based parameterisable system for matrix product implementation using both systolic array and bit serial DA methodologies have been presented in [1]. High performance systolic arrays for band matrix multiplication were proposed in [14]. However, none of these works provide an insight into the methodology to map their processing elements into the target FPGA architecture optimally.

We now introduce the mathematical logic supporting DA to realize the elements of the matrix C. We assume that each element of the matrix B, b_{ij}, is a two's complement B_x-bit wide number. If the coefficients a_{ik} are known a priori, the partial product term

$a_{ik}b_{kj}$ becomes a multiplication with a constant. In this way, several optimizations to the constant coefficient multiplier architectures can be carried out [12].

$$
\begin{aligned}
c_{ij} &= \sum_{k=1}^{m} a_{ik}b_{kj} \\
&= \sum_{k=1}^{m} a_{ik}\left(\sum_{l=0}^{B_x-2} 2^l b_{kj,l} - 2^{B_x-1}b_{kj,B_x-1}\right) \\
&= -\sum_{k=1}^{m} 2^{B_x-1}b_{kj,B-1}a_{ik} + \sum_{k=1}^{m}[(a_{ik}b_{kj,0})\,2^0 + \cdots + (a_{ik}b_{kj,B_x-2})\,2^{B_x-2}] \\
&= -(a_{i1}b_{1j,B_x-1} + \cdots + a_{im}b_{mj,B_x-1})\,2^{B_x-1} + (a_{i1}b_{1j,0} + \cdots + a_{im}b_{mj,0})\,2^0 \\
&\quad + (a_{i1}b_{1j,1} + \cdots + a_{im}b_{mj,1})\,2^1 + \cdots + (a_{i1}b_{1j,B_x-2} + \cdots + a_{im}b_{mj,B_x-2})\,2^{B_x-2} \\
&= \sum_{l=0}^{B_x-1} 2^l \times \sum_{k=1}^{m} \underbrace{a_{ik} \times b_{kj,l}}_{f(a_{ik},b_{kj,l})} = \sum_{l=0}^{B_x-1} 2^l \times \sum_{k=1}^{m} f(a_{ik}, b_{kj,l})
\end{aligned}
\tag{7.1}
$$

The preferred implementation method is to realize the mapping $f(a_{ik}, b_{kj,l})$ using a ROM. Thus the ROM can be preprogrammed to accept an B_x-bit input vector b_{kj} and output $f(a_{ik}, b_{kj,l})$ which are precomputed and stored in the ROM. The individual mappings $f(a_{ik}, b_{kj,l})$ are weighted by the appropriate power-of-two factor (realized by simple left-shift operations) and added, thereby leading to a multiplier-free realization [8].

Some recent works [5–7] have proposed dynamically run-time reconfigurable finite impulse response (FIR) filter where the filter coefficients can be reconfigured during run-time using Xilinx run-time, reconfigurable 5-input LUT, CFGLUT5, as a primitive [13]. It uses the 6-input LUTs present in SLICEM logic of Virtex-5, Virtex-6, Virtex-7, and Spartan-6 FPGAs and provides configuration interface pins comprising of configuration data in (CDI), configuration data out (CDO), a configuration clock (CLK), and a clock enable (CE) as shown in Fig. 7.10. The entire contents of the LUT can be changed by loading a new 32-bit configuration vector at CDI in

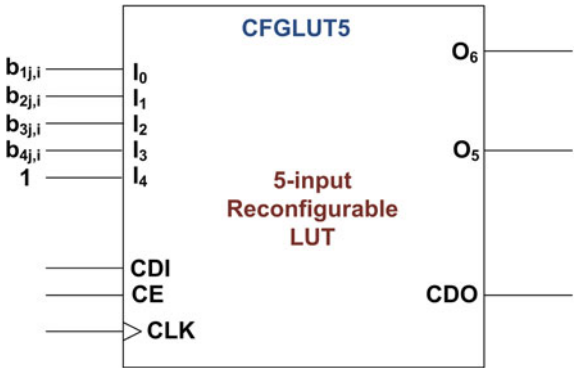

Fig. 7.10 5-input dynamically reconfigurable Lookup Table (LUT) [13]

32 clock cycles. Hence, for our implementation, the matrix A which was assumed to have constant coefficients can be completely modified once in every 32 clock cycles using a simple state machine and staging logic to reload the memory contents of the dynamically reconfigurable LUTs [12]. Such run-time self-reconfigurable architecture has been proposed in [4] to design constant multipliers (both signed and unsigned) on FPGA, where the constants can be reloaded in run-time.

7.3.2.1 Proposed Distributed Arithmetic-Based Architecture for Matrix Multiplication

Going by the consideration that performance has been our primary design target, and noting the abundance of logic resources that modern family of FPGAs promise, we go for a real-time signal processing application where maximum speed can be achieved through implementation of a *fully pipelined word-parallel architecture* as shown in Fig. 7.11. We have exploited the dual-output LUT nature of the CFGLUT5 primitives for retrieving the contents of the precomputed values stored in it by passing the bit vectors of the elements from the matrix B as address lines. The output of the LUTs are passed via pipeline registers as shown in the figure forming a *pipelined adder tree* as we keep moving from one stage of the adder to the next. Also, we

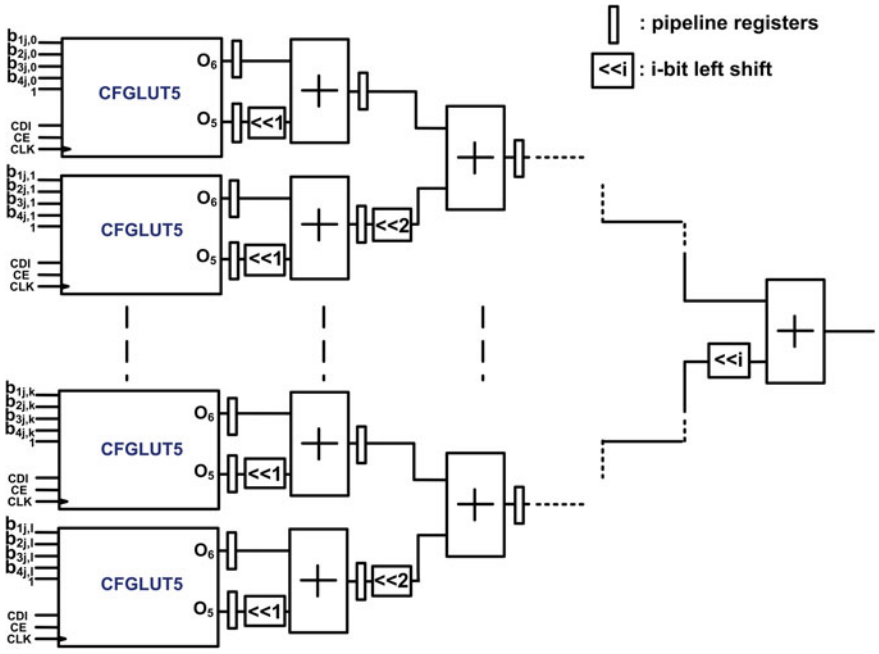

Fig. 7.11 Proposed architecture for DA-Based Matrix Multiplication using reconfigurable LUTs

try to push most of the left-shift operations toward the final output of the adder tree as much as possible for reduction of word size and consequently, the size of the intermediate ripple carry adders that decide the critical path of the architecture. Each of the HDL descriptions of the ripple carry adders and their subsequent placement constraint files have been generated using the *FlexiCore* platform.

7.3.2.2 Xilinx IP Core-Based or DSP-Based Architecture for Matrix Multiplication Using Constant Coefficient Multipliers

It is to be noted that neither IP Core-based fabric logic or logic realized using DSP slices offer dynamic reconfigurability of the constant cofficients that serves as one of the input operands of a multiplier logic. Hence, it is difficult to obtain an exact logic equivalent of the CFGLUT5 primitives using existing design philosophies. For comparison of our proposed architecture with circuits realized through IP Core-based fabric logic and DSP logic, we have considered constant coefficient multipliers as shown in Fig. 7.12. Design elements such as constant coefficient signed multipliers and signed adders have been designed using the Xilinx IP Core-based CAD tool. On the other hand, DSP slices can be configured to support a multiply adder/subtractor logic with the help of which matrix multiplication was achieved.

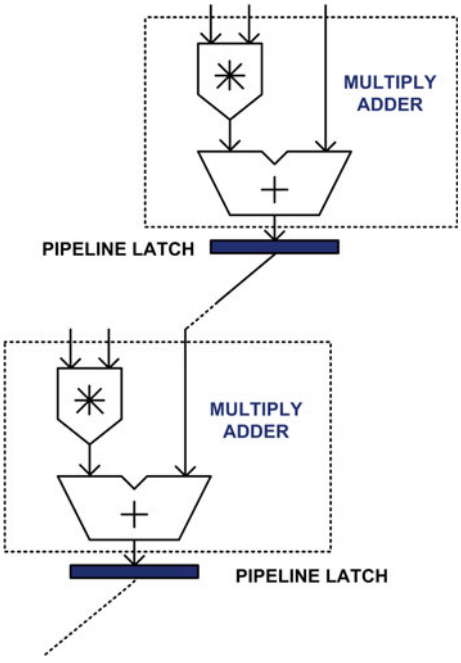

Fig. 7.12 Xilinx IP Core-based or DSP-based architecture for matrix multiplication using constant coefficient multipliers

Table 7.3 Implementation results for multiplication of a 4 × 4 with a 4 × 2 matrix where all the 16-bit elements of the 4 × 4 matrix have been chosen as a unique (two's complement) constant, 32767, and all elements of the 4 × 2 matrix are user-input 16-bit two's complement numbers

Design style	Freq (MHz)	Latency	Power-delay product (pJ)	#FF	#LUT	#DSP	#Slice
IP core based	125.15	3	616.27	294	780	0	257
DSP slice based	171.32	3	418.86	294	95	8	51
Proposed design	**602.41**	**4**	**973.42**	**1194**	**928**	**0**	**258**

7.3.2.3 Implementation Results for Matrix Multiplication Circuit

Implementation results for multiplication of a 4 × 4 with a 4 × 2 matrix where all the 16-bit elements of the 4 × 4 matrix have been chosen as a unique 16-bit (two's complement) constant, 32767. This was done to compare the worst-case performance, as use of the constant 32767 results in maximum hardware and thus slower speed in the IP Core-based designs. All elements of the 4 × 2 matrix are real-time data inputs, each of which are 16-bit two's complement numbers. The architecture has been compared for three design styles—IP Core-based fabric logic, DSP slice-based logic, and the proposed architecture with constrained placement. Results clearly reveal that the proposed architecture clearly outperforms other design styles with respect to speed. This is because the critical path is dictated only by the delay of the longest fast carry chain adder, realized by careful, constrained placement. The PDP is however higher as we realize a *fully pipelined word-parallel architecture* for which the hardware consumption is higher. The Xilinx IP Core-based design has a multiplier and an adder in its critical path and coupled with random, unconstrained placement, thus adversely affecting the operating frequency of the resultant implementation. For the DSP-based design, which has configured DSP slices and registers, the speed is much slower compared to the proposed architecture (Table 7.3).

7.4 Summary

The *FlexiCore* CAD tool is equipped with the facility of automatically generating synthesizable HDL and constraint placement directives for arithmetic circuits. The utility of the CAD tool has been done for design of multimodule architectures to prove its effectiveness and ease of design. In the next section, we conclude the book and list some of the future research directions on the same principles of this work.

References

1. Amira, A., Bensaali, F.: An FPGA based parameterisable system for matrix product implementation. In: IEEE Workshop on Signal Processing Systems (SiPS), pp. 75–79 (2002)
2. Amira, A., Bouridane, A., Milligan, P., Sage, P.: A high throughput FPGA implementation of a bit-level matrix product. In: IEEE Workshop on Signal Processing Systems (SiPS), pp. 356–364 (2000)
3. Brent, R.P., Kung, H.T.: A systolic algorithm for integer GCD computation. In: IEEE 7th Symposium on Computer Arithmetic (ARITH), pp. 118–125 (1985)
4. Hormigo, J., Caffarena, G., Oliver, J.P., Boemo, E.: Self-reconfigurable constant multiplier for FPGA. ACM Trans. Reconfigurable Technol. Syst. **6**(3), 14.1–14.17 (2013)
5. Kumm, M., Moller, K., Zipf, P.: Dynamically reconfigurable FIR filter architectures with fast reconfiguration. In: 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), pp.1–8 (2013)
6. Kumm, M., Moller, K., Zipf, P.: Partial LUT size analysis in distributed arithmetic FIR filter on FPGAs. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2054–2057 (2013)
7. Kumm, M., Moller, K., Zipf, P.: Reconfigurable FIR filter using distributed arithmetic on FPGAs. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2058–2061 (2013)
8. Meyer-Baese, U.: Digital Signal Processing with Field Programmable Gate Arrays. Signals and Communication Technology, 3rd edn. Springer, New York (2007)
9. Parhi, K.K.: VLSI Digital Signal Processing Systems: Design and Implementation. Wiley, New York (2007)
10. Stehlé, D., Zimmermann, P.: A binary recursive Gcd algorithm. In: Proceedings of ANTS'04. Lecture Notes in Computer Science, vol. 3076, pp. 411–425, Springer (2004)
11. White, S.A.: Applications of distributed arithmetic to digital signal processing: a tutorial review. IEEE ASSP Mag. **6**(3), 4–19 (1989)
12. Wirthlin, M.J.: Constant coefficient multiplication using look-up tables. J. VLSI Signal Process. Syst. **36**(1), 7–15 (2004)
13. Xilinx Inc.: Virtex-5 libraries guide for HDL designs. UG621 (v11.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex5_hdl.pdf. Accessed 6 Sept 2009
14. Yang, Y., Zhao, W., Inoue, Y.: High-performance systolic arrays for band matrix multiplication. In: IEEE International Symposium on Circuits and Systems (ISCAS), vol. 2, pp. 1130–1133 (2005)

Chapter 8
Conclusions and Future Work

Abstract This chapter summarizes the contributions of this book. It also provides certain important pointers to potential future research direction in the field of FPGA-based arithmetic circuit design.

8.1 Introduction

In the world of high-speed digital circuits, FPGAs have steadily grown in stature as a platform of choice in the last two decades. The most recent technological advancements ensure huge logic support and a host of other features, including on-chip embedded processors and DSP blocks (hard macros). Careful design considerations such as custom design methodology with manual instantiation of hardware primitives and macros, and their careful, constrained placement on the FPGA fabric leads to very high performances in terms of speed. We have considered various arithmetic circuits which are very regular in their architectures, and have shown how they can be implemented following the bit-sliced design paradigm. Designs that are pipelined and have a very regular dataflow, like those considered in our work, usually lend themselves to regular floorplanning [3]. Due to the coarse nature of the routing matrix in FPGAs, placement is vastly more important in the context of ensuring high performance, in comparison to routing [3]. Since each slice of an FPGA is register-rich, pipelined implementations can be done with ease without consuming additional number of slices.

8.2 Contributions of the Book

The contributions of this book can be summarized as follows:

- In Chaps. 2 and 3, we have presented the advanced FPGA architectures of Xilinx FPGAs, and have stressed upon the fact that the knowledge of the structural implementation is an important prerequisite for designing optimized arithmetic circuits

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1_8

on an arbitrary FPGA fabric. Only the knowledge of the structural implementation can help the designer to appropriately configure and optimally utilize the LUTs, carry chains and wide function multiplexers, aided through appropriate Boolean logic manipulations, and place the logically related primitives in closest proximity to one another using placement constraints.

- We have taken up different examples of pipelined implementations of datapath and controlpath arithmetic circuits in Chaps. 4 and 5, and have exhibited superior operand width scalability of the designs with respect to the performance of the circuits. In Chap. 6, we have discussed the improved FPGA implementation of pseudorandom binary sequence generator circuits based on cellular automata, following the proposed methodology.
- In Chap. 7, we have described the operations of a CAD software tool *FlexiCore* developed by us, that leverages the regularity of the arithmetic and CA circuits, and automatically generates the hardware descriptions and related placement constraints.
- The design automation tool was utilized to implement multiple modules for two test circuits: (a) GCD calculator and (b) Distributed Arithmetic-based matrix multiplier. Implementation results reveal that our implementations comfortably outperforms all other modes of implementations.

It must also be noted that no amount of changes in the option settings of the Xilinx ISE tool for optimization goal and effort, placer extra effort, global optimization, and other additional synthesis and timing constraints to realize circuits using existing CAD tools or design styles can infer or match up the high speed of our proposed implementations.

8.3 Future Research Directions

While high-performance architecture development for arithmetic circuits on FPGAs has been thoroughly researched in the exiting literature, only a handful of works have discussed a low-level custom implementation-based design philosophy on FPGA fabric. This was the primary motivation for us to pursue research in this direction, and can pave the way for many other future research directions on this topic. All the architectures and circuits discussed in this book are very useful in many signal processing and image processing applications. Certain research directions can be pursued from here.

- It was pointed out in Chap. 4 as to how iterative array structures can be mapped to the Xilinx fabric logic by implementing multipliers and squarers using *hybrid, carry save* approach. In [6], an iterative array for nonrestoring binary division was proposed, whereas in [7], a cellular array for nonrestoring extraction of square roots was proposed. Both the designs use controlled add/subtract cells to perform the required subtraction/addition as prescribed by the algorithm. Mapping of controlled add/subtract logic to LUT and carry chain fabric has been discussed in detail in Sect. 4.2 of Chap. 4. Since optimized implementation of dividers and

square root extractors on FPGA has relatively been an unexplored topic, a significant contribution can be made in this area.

- Chap. 6 discusses FPGA implementations of two-rule, linear, one-dimensional cellular automata structures. A similar philosophy can be extended toward optimized implementation of two-dimensional cellular automata having higher neighborhood dependencies [2, 8]. Two classes of circuits having structural and functional similarities to CAs and having wide application domains are *Linear Feedback Shift Registers* (LFSRs) and *Multiple Input Signature Register* (MISR). While LFSRs are widely used as source of pseudorandom bitstreams, MISRs are used for test response compaction in IC testing. For LFSRs, longer sequences of random numbers require a longer LFSR with higher hardware overhead. In [5], an area-efficient LFSR implementation was proposed where the number of FF stages between any two taps can be realized using the SRL16 primitives. Advanced Virtex and Spartan families support special LUTs that can realize upto a 32-bit stage delay as described in Sect. 3.3 of Chap. 3. A MISR has a similar structure; the only difference being the input to every FF is fed through an XOR/XNOR gate [4]. Under such a situation, MISR can be realized in a similar fashion as has been done for the CA circuits described in Chap. 6, subject to the circuit complexity.
- Research can be extended further for designing other signal processing applications using a similar design philosophy like Finite Impulse Response (FIR) filters, Discrete Fourier Transform (DFT), and Fast Fourier Transform (FFT) processors that involve computation of trigonometric functions using CORDIC (COordinate Rotation DIgital Computer) algorithm [9]. The proposed design methodology is also applicable in the implementation of high-performance floating point arithmetic circuits.
- Further research work can be focused upon improving the proposed CAD tool for achieving automatic pipelining inside general arithmetic or DSP circuits, where user only needs to input the desired latency. Another approach for system-level pipelining can be extended to develop a tool which has the capability to decide on the appropriate number of pipeline stages to get maximum performance by studying delay models for operators as well as the target fabric.
- All our proposed architectures have been targeted toward Xilinx FPGA platform. The primary reasons behind this choice were the ease of availability of the Xilinx ISE software, flexibility of low-level custom design techniques on the software version available, ease of integration of the CAD tool with Xilinx ISE, and widely available user manuals providing detailed literature on platform-specific FPGA architecture. Significant research work can be carried out on the applicability of the same scheme for other FPGA platforms. Altera is a major FPGA vendor along with Xilinx. However, the difficulty for deploying the same scheme on Altera platforms can be attributed toward the unavailability of the user guides and manuals for Altera platforms describing the native low-level primitives. Though a few related documents do exist [1], they contain very little information about how to do such low-level design optimally and effectively. We would target FPGA platforms other than Xilinx in future, with the possibility of modifying the implementations to suit the target FPGA platform.

References

1. Altera Inc.: Designing with low–level primitives user guide, Altera Corporation. http://www.altera.com/literature/ug/ug_low_level.pdf. Cited 30 April 2007
2. Chowdhury, D.R., Sengupta, I., Chaudhuri, P.P.: A class of two dimensional cellular automata and their applications in random pattern testing. J. Electron. Test. Theory Appl. **5**(1), 67–82 (1994)
3. Kilts, S.: Advanced FPGA Design Architecture, Implementation and Optimization. Wiley-IEEE Press, New Jersey (2007)
4. Lala, P.K.: An Introduction to Logic Circuit Testing. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, San Mateo (2008)
5. Lim, S., Miller, A.: Xilinx Inc., Application Note: Virtex Series, Virtex–II Series and Spartan–II Family, LFSRs as Functional Blocks in Wireless Applications, XAPP220 (v1.1). http://areeweb.polito.it/didattica/corsiddc/01NVD/Matappnote/Xapp220LinSFR.pdf. Cited 11 Jan 2001
6. Majithia, J.C.: Nonrestoring binary division using a cellular array. Electron. Lett. **6**(10), 303–304 (1970)
7. Majithia, J.C., Kitai, R.: A cellular array for the nonrestoring extraction of square roots. In: IEEE Trans. Comput. **C-20**(12), 1617–1618 (1971)
8. Tomassini, M., Sipper, M., Perrenoud, M.: On the generation of high-quality random numbers by two-dimensional cellular automata. IEEE Trans. Comput. **49**(10), 1146–1151 (2000)
9. Volder, J.E.: The CORDIC Trigonometric computing technique. In: IRE Trans. Electron. Comput. **EC–8**(3), 330–334 (1959)

Index

A. Palchaudhuri and R.S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA*, Springer Series in Advanced Microelectronics 51,
DOI 10.1007/978-81-322-2520-1