Amir Sabbagh Molahosseini
Leonel Seabra de Sousa
Chip-Hong Chang *Editors*

Embedded Systems Design with Special Arithmetic and Number Systems

Springer

Embedded Systems Design with Special Arithmetic and Number Systems

Amir Sabbagh Molahosseini
Leonel Seabra de Sousa • Chip-Hong Chang
Editors

Embedded Systems Design with Special Arithmetic and Number Systems

Springer

Editors
Amir Sabbagh Molahosseini
Department of Computer Engineering
Islamic Azad University, Kerman Branch
Kerman, Iran

Leonel Seabra de Sousa
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal

Chip-Hong Chang
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore, Singapore

ISBN 978-3-319-49741-9 ISBN 978-3-319-49742-6 (eBook)
DOI 10.1007/978-3-319-49742-6

Library of Congress Control Number: 2017934074

© Springer International Publishing AG 2017
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

For decades, innovation in the chip industry has been predominantly driven by the demand of personal computers (PCs). The central processing unit (CPU), which is the brain of a PC, is built from fundamental devices that follow a miniaturization path in close agreement with Moore's Law. Fifty years after the integrated circuit (IC) invention, the remarkable success in device geometry extrapolation and the seductive growth of chip content have changed the landscape of the platform of computing. New nano-/biocircuits and systems have emerged to take advantage of the increased integration density and connectivity to enhance living and lifestyles. The PC's influence on chip design is gradually eroded by the proliferation of connected mobile and portable devices due to the growth of consumer web and services of all types being made online and on demand. Tablets and mobile phones have overtaken the consumer market of desktops and laptops, and embedded systems permeate every aspect of work and life.

The new age of ubiquitous computing brings multifaceted challenges of versatility, portability, environmental friendliness, computational heterogeneity, and methodological pluralism into the design of embedded systems. It is time for a radical change in the algorithm and architecture of embedded system design to effect innovative solutions to the nascent multivariate optimization problems with new design constraints. Ironically, after more than 40 years of enormous R&D investment into renewing almost every relevant technology for IC design and manufacturing, the fundamental arithmetic operations and algebraic structures used in the prevalent embedded systems are still based on the same conventional weighted binary number representation inherited from the earliest microprocessor design. It is well understood that the way numbers are represented in a digital system has an impact on all levels of design abstraction from algorithm and architecture to circuit topology and layout. The choice of number systems influences the workload of an application by determining the number and complexity of operations required to accomplish a specific task. Since data activities depend on circuit topologies and stochastic properties of the inputs, the representation of data has a direct effect on the operator strength and the performance predictability.

Embedded systems implemented by algorithms based on the weighted binary number system suffer from the curse of dimensionality due to the inevitable long chain of carry propagation. It has limited "parallelism" and "modularity" to fully utilize the emerging VLSI technology on optimization of essential hardware attributes. Higher radix weighted number systems have reduced the word length of operations, and non-weighted number systems can limit the inter-digit carry propagation. Some alternative number systems inherently possess greater parallelism and sparser inter-digit communication amenable to speed enhancement and power reduction in a multi-ALU system and remain advantageous to layout and routing for the current and emerging 3D stacked IC technology. The design space of arithmetic functions based on different unconventional number systems and their potential have not been fully tapped. This book aims at overcoming issues and problems confronted by the use of different unconventional number systems to find a new avenue to sustain the continual design and development growth of the new generation of application-specific embedded systems.

Based on the chapter's content, we have divided the book into two parts. The first part is aimed to introduce the readers to the concepts, foundations, and design methods of circuits and systems for unconventional number systems including *residue number system*, *redundant number system*, *decimal floating-point number system*, and *continuous-valued number system*. Part two is dedicated to the applications of unconventional number systems in different areas of embedded system design from digital signal and image processing to emerging cryptographic algorithms. Chapter 1 introduces residue number systems and proposes a method to teach residue number systems in the context of embedded systems. Chapter 2 proposes a compact and scalable architecture for designing residue number system (RNS)-based programmable processors for general moduli sets. The design of an RNS processor has shown that performance and efficiency are improved by exploring the parallelism and carry-free characteristic of RNS.

The latest mathematical achievements in performing non-modular RNS operations using diagonal function are described in Chap. 3. Efficient implementation of non-modular RNS operations such as scaling, sign detection, magnitude comparison, and overflow detection could open the doors of general-purpose processors to RNS. Chapter 4 provides an overview of the principle of detecting, locating, and correcting single and multiple residue digit errors in arithmetic processing by redundant residue number system (RRNS) and highlights some applications that utilize RRNS codes. The location of erroneous residue digits is identified as the bottleneck operation for RRNS implementation. The complexity and latency of syndrome-based and CRT-based approaches are analyzed and compared. Inspired by the need for decimal floating-point numbers on computers, Chap. 5 presents efficient hardware implementations of a number of key arithmetic operations. It also discusses the intricacies of binary integer decimal and densely packed decimal encodings in migrating the designs from binary into decimal floating point and the functional verification approach to envisage future unified binary and decimal arithmetic units. Chapter 6 explores the design space of using redundant binary representation for high-performance Booth-encoded multiplier implementation.

It analyses the performance trade-offs of 21 different configurations of redundant binary multiplier architecture constructed from different binary to redundant binary encoding formats, different radices of binary, pseudo redundant binary and binary Booth encodings, and redundant binary to binary converters. Chapter 7 presents fundamentals of continuous-valued number system. The arithmetic operations in continuous-valued number system are performed using simple analog circuitry to provide arbitrary implementation precision. Potential applications of this number system are in the area of low noise and low cross-talk circuitry for arithmetic circuits used in mixed-signal systems and digitally assisted analog circuits. Chapter 8 reviews basic principles and hardware implementations of RNS-based digital signal processing (DSP) units by a team of researchers with 20 years of research experience in using RNS for DSP applications. They believe that RNS is promising with appropriate trade-offs between circuit parameters. The parallelism of RNS is advantageous in image processing applications where addition and multiplication are the dominant operations. This issue is comprehensively investigated in Chap. 9, where RNS-based realization of many important image processing applications such as edge detection, sharpening, smoothing, and wavelet processing is reviewed.

Chapter 10 introduces another unconventional number system called *logarithmic number system (LNS)*. This number system offers not only efficient multiplication but also division which is a difficult operation in both binary and residue number systems. It is shown that using LNS for FIR filter implementation can lead to significant saving in power consumption. Chapter 11 introduces canonical and extended double base number systems and the search algorithms for finding their minimum or quasi-minimum forms. Their applicability to efficient programmable FIR filter design is demonstrated by the direct mapping of these sparse double-based number representations into efficient time-multiplexed multiple-constant-multiplication architecture consisting of only adders, multiplexers, programmable shifters, and a lookup table. The results show that FIR filters designed with extended double base number system can reduce the logic complexity by up to 47.81 % and critical path delay by up to 14.32 % compared with the designs based on conventional binary number system. Chapter 12 addresses how RNS can be effectively used to design public-key cryptography systems. The design concepts, methodologies, and challenges of implementing RSA and elliptic curve public-key algorithms with residue arithmetic and security of RNS-based cryptosystems are presented and discussed in this chapter. Chapter 13 investigates the advantages of RNS arithmetic in *lattice-based cryptography*, an emerging cryptographic algorithm with post-quantum security. It is shown that the high parallelism of RNS is well suited to address the challenging problem of the efficient realization of this next-generation cryptographic algorithm. Finally, the last chapter introduces attractive applications of RNS in computer networks. In contrast to other applications, here RNS representation is used for routing, packet forwarding, and multicasting. Instead of using single regular large packet, distributive residues are used to achieve lower energy consumption of the sensor nodes and longer network lifetime.

To sum up, at the rate of growth of embedded processors and their increasing share and dominance in the consumer electronic markets, the university students

and professors, researchers, and industrial designers should be prepared for the bold and radical evolution of how future embedded systems will be designed. This book solicits alternative approaches to the design of efficient embedded systems effected by the change in the fundamental number representation for which digital arithmetic operations are performed. It is hoped that the comprehensive review, analysis, efficient implementation methods, and new applications covered in this book will stimulate and inspire more interesting applications, new developments, and exploration of these or other unconventional number systems.

Kerman, Iran — Amir Sabbagh Molahosseini
Lisboa, Portugal — Leonel Seabra de Sousa
Singapore, Singapore — Chip-Hong Chang

Contents

Part I Unconventional Number Representations: Arithmetic Units and Processor Design

1 Introduction to Residue Number System: Structure and Teaching Methodology 3
Amir Sabbagh Molahosseini and Leonel Sousa

2 RNS-Based Embedded Processor Design 19
Pedro Miguens Matutino, Ricardo Chaves, and Leonel Sousa

3 Non-Modular Operations of the Residue Number System: Functions for Computing 49
Giuseppe Pirlo

4 Fault-Tolerant Computing in Redundant Residue Number System .. 65
Thian Fatt Tay and Chip-Hong Chang

5 Decimal Floating Point Number System 89
Hossam A.H. Fahmy

6 Design and Evaluation of Booth-Encoded Multipliers in Redundant Binary Representation 113
Yajuan He, Jiaxing Yang, and Chip-Hong Chang

7 Robust Analog Arithmetic Based on the Continuous Valued Number System 149
Babak Zamanlooy and Mitra Mirhassani

Part II Applications of Unconventional Number Representations

8 RNS Applications in Digital Signal Processing 181
Gian Carlo Cardarilli, Alberto Nannarelli, and Marco Re

9 RNS-Based Image Processing 217
Nikolay Chervyakov and Pavel Lyakhov

10 Logarithmic Number System and Its Application in FIR Filter Design 247
Vassilis Paliouras

11 Double-Base Number System and Its Application in FIR Filter Design 277
Jiajia Chen and Chip-Hong Chang

12 RNS-Based Public-Key Cryptography (RSA and ECC) 311
Dimitris Schinianakis and Thanos Stouraitis

13 RNS Approach in Lattice-Based Cryptography 345
Jean-Claude Bajard and Julien Eynard

14 RNS Applications in Computer Networks 369
Azadeh Alsadat Emrani Zarandi

Index 381

Part I
Unconventional Number Representations: Arithmetic Units and Processor Design

Chapter 1
Introduction to Residue Number System: Structure and Teaching Methodology

Amir Sabbagh Molahosseini and Leonel Sousa

1.1 Introduction

The current embedded systems and wireless world applications require energy efficiency and fast computation [1, 2]. A lot of progress has been made on different aspects of electronic systems in this direction, from transistor to architecture level. However, arithmetic level operations still suffer from long carry-propagation chains. Although many computer arithmetic methods, such as parallel-prefix computations, have been introduced to reduce delay, the transition from traditional weighted two's complement number system to an efficient alternative number system can have a significant impact in computation systems performance. The residue number system (RNS) has been the most interesting and challengeable alternative number system in computer arithmetic for more than half a century [3–5]. Its main interesting feature is the ability of performing addition, subtraction, and multiplication without carry-propagation between residues [6, 7]. This exceptional ability makes the computer arithmetic researcher's target, i.e., carry-free arithmetic, achievable. However, there are some difficulties for efficiently implementing division, residue to binary conversion, sign detection, magnitude comparison, and overflow detection. Therefore, RNS has been mainly used in applications where dominant operations are addition, subtraction, and multiplication, such as digital signal processing [8] and

A.S. Molahosseini (✉)
Department of Computer Engineering, Kerman Branch, Islamic Azad University, Kerman, 7635131167 Iran
e-mail: amir@iauk.ac.ir

L. Sousa
Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento (INESC-ID), Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon 1000-029 Portugal
e-mail: las@inesc-id.pt

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_1

cryptography [9, 10]. Moreover, RNS has many other interesting properties, such as error detection/correction and fault-tolerant abilities [11], which makes it suitable for other type of applications, namely computer networks [12], DNA arithmetic [13], and cloud storage [14]. In recent years, many advances have occurred in both designing and applications targeting RNS [5].

In order to design an RNS-based digital system, first of all the dynamic range to cover has to be defined, i.e., the range of numbers that need to be handled. Then, an efficient moduli set should be selected according to that dynamic range and the type of targeted applications. A forward converter is required to interface RNS with the weighted representation of numbers. Then, independent modulo arithmetic channels perform computations on the residues in parallel. Finally, a reverse converter translates the resulting RNS number into a weighted representation, used by conventional digital systems. In addition to those essential RNS components, some optional components such as a sign detector, a magnitude comparator, and a scaler can be used in the RNS datapath according to the application's requirements [5]. In the rest of this chapter, different methods for designing all RNS parts and components will be introduced and recent work and advances discussed. Moreover, for the first time, a step-by-step RNS teaching methodology is introduced, which integrates hardware structures, implementations, and applications together with mathematical background for RNS. This teaching methodology can facilitate electrical and computer engineering researchers and students to more easily understand RNS concepts and designs, and motivate them to research new RNS architectures and apply it in further practical applications.

1.2 RNS Structure

The first step for designing an RNS system is to select an appropriate moduli set, given the dynamic range (the product of all moduli) and the degree of required parallelism (corresponding to the number of moduli in the set). The moduli set includes pair-wise relatively prime numbers. In general, RNS moduli sets can be organized into two main groups: *arithmetic-friendly* and *conversion-friendly* moduli sets. Conversion-friendly moduli sets result in simpler reverse converter structures, in contrast to arithmetic-friendly sets. If, for the target application, the rate of internal arithmetic operations to conversions is high, arithmetic-friendly should be selected and vice versa. Some hints and useful points about the method of selection and sample moduli sets can be found in [15–19].

The 2^n and $2^n \pm 1$ moduli are the most well known and efficient since they allow the design of simple hardware arithmetic units. Based on these popular numbers, lots of special moduli sets have been introduced for RNS. The most known is the traditional moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. Although the majority of research on RNS has been devoted to this three moduli set, the limited parallelism provided prevents its usage in nowadays high-performance computation systems.

Thus, extended RNS moduli sets such as $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$ and $\{2^n - 1, 2^n, 2^n + 1, 2^{n-1} - 1, 2^{n+1} - 1\}$ have been proposed. Note that in addition to special RNS moduli sets, general RNS moduli sets exist where each modulo can be favorably selected by the designer; for example, for prime moduli sets, where all the moduli are prime, modulo multiplications can be transformed to additions based on an isomorphism [20]. Note that moduli sets with a larger amount of moduli are usually considering for cryptography applications, which operate on very large numbers [21].

A general RNS structure is depicted in Fig. 1.1. The forward converter receives weighted integer operands and produces residues. For example, let us consider the moduli set $\{2^n - 1,\ 2^n,\ 2^n + 1\}$ and $n = 4$. The dynamic range is 4080, and the

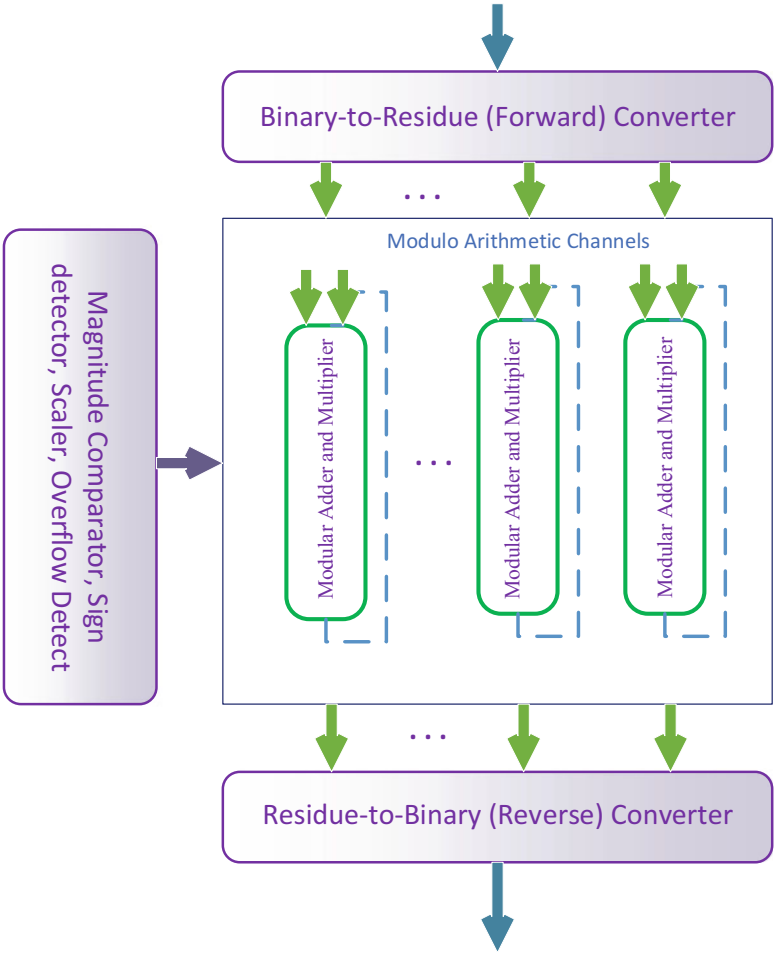

Fig. 1.1 The RNS structure

residue representation of the initially weighted input $X = 20$ can be achieved by calculating the remainder of the division of X by the different moduli, resulting in $X = (5, 4, 3)$. In practice, one does not have to compute the entire process of division to obtain the residues, but only modular additions are required to obtain them for particular moduli sets [22]. Therefore, forward converters are parallel architectures, for the different residues, composed of independent units, each one computing one residue based on adders. The forward converter design has been addressed in many works, such as [23–31]. Moreover, recently, the design of forward converters for general RNS moduli sets has been considered in [32].

The design of a reverse converter design includes five phases. In the first phase, a conversion algorithm is selected according to the moduli set and its properties. In the subsequent phase, the values of the moduli together with the values of multiplicative inverses are considered in the conversion formulas. Then, residue arithmetic properties should be used to work on the conversion equations in such a way that makes the computation simpler. Next, the simplified equations are implemented using arithmetic circuits such as adders and logical gates. The most well-known conversion algorithms are: the Chinese remainder theorem (CRT), the mixed-radix conversion (MRC), and the New CRTs-I and CRT–II, derived from the original CRT. Details and examples about each of these algorithms can be found in [22]. There are many reverse converter designs for RNS moduli sets. Since the structure of the reverse converter is dependent on the moduli sets, special moduli sets with a limited number of moduli are usually used to design efficient reverse converters to optimize the design for different cost functions, such as performance, cost, and power consumption.

Modulo adder and multiplier designs are also an important research field in RNS because they are not only used in the parallel modulo arithmetic channels, but also in the forward and reverse converters. There are different architectures for designing adders and multipliers, for general and specific moduli [7], most of them for the moduli set $2^n - 1$, $2^n + 1$, and 2^n (for this last modulo regular binary adders and multipliers can be used [22]).

There are many other operations in RNS which may be required by the applications. One of these is sign detection. In contrast to the weighted number system, where the sign can be detected by the most significant bit (MSB), in RNS sign detection is hard, requiring the comparison of numbers with half of the dynamic range: integers in the first half of dynamic range are considered positive and all the other negative [7]. Furthermore, magnitude comparison is another hard operation in RNS. Usually, a semi-transformation of the RNS number to a weighted equivalent is applied to design RNS sign detectors and magnitude comparators. Other difficult operations are scaling and division. These are important operations, namely to avoid overflow in series of RNS multiplications and additions, which are typical operations of digital filtering. The scaling factor is usually considered as the power of two modulo to reduce the complexity of the operation. The overflow, that occurs when the result of an operation cannot be represented within the dynamic range, is also quite hard to detect in RNS. Table 1.1 introduces the most efficient and latest references that report hardware design for each RNS component.

Table 1.1 State-of-the-art RNS hardware components

Component	Moduli	Reference
Forward converter	$\{2^n - 1, 2^n, 2^n + 1\}$	[23–27]
	$2^n \pm 2^k + 1$	[28]
	$2^n \pm k$	[29]
	Generic	[30–32]
Adders and multipliers	$2^n \pm 1$	[33–41]
	Multi-modulus	[42–46]
	$2^n \pm k$	[47, 48]
	$2^n - 2^{\mathrm{k}} - 1$	[49]
	Generic	[50–52]
Reverse converter	$\{2^n - 1, 2^n, 2^n + 1\}$	[53]
	$\{2^n - 1, 2^n, 2^{n+1} - 1\}$	[54]
	$\{2^\alpha, 2^\beta - 1, 2^\beta + 1\}$	[55–57]
	$\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$	[58]
	$\{2^n - 1, 2^n + 1, 2^{2n}, 2^{2n} + 1\}$	[59, 60]
	$\{2^n - 1, 2^n, 2^n + 1, 2^{2n+1} - 1\}$	[60, 61]
	$\{2^n - 1, 2^n, 2^n + 1, 2^n - 2^{(n+1)/2} + 1, 2^n + 2^{(n+1)/2} + 1, 2^{n\pm1} + 1\}$	[62]
	$\{2^n - 1, 2^n, 2^n + 1, 2^{n-1} - 1, 2^{n+1} - 1\}$	[63]
	$\{2^n - 1, 2^n + 1, 2^n - 3, 2^n + 3\}$	[64]
	Signed-output	[65]
Overflow detection	$\{2^n - 1, 2^n, 2^n + 1\}$	[66, 67]
Scaling	$\{2^n - 1, 2^n, 2^n + 1\}$	[68, 69]
	$\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$	[70]
	$\{2^n - 1, 2^{2n}, 2^n + 1, 2^{2n+1} - 1\}$	
Sign detection	$\{2^n - 1, 2^n, 2^n + 1\}$	[71]
	$\{2^n - 1, 2^{n+k}, 2^n + 1\}$	[72]
	$\{2^{n+1} - 1, 2^n - 1, 2^n\}$	[73]
	$\{2^n - 1, 2^n + 1, 2^{2n}, 2^{2n} + 1\}$	[65]
	$\{2^n - 1, 2^n, 2^n + 1, 2^n - 2^{(n+1)/2} + 1, 2^n + 2^{(n+1)/2} + 1, 2^{n\pm1} + 1\}$	
Magnitude comparator	$\{2^n - 1, 2^n, 2^n + 1\}$	[74]
	$\{2^n - 1, 2^{n+k}, 2^n + 1\}$	[75]
	$\{2^{n+k}, 2^n - 1, 2^n + 1, 2^{n\pm1} - 1\}$	[76]

1.3 RNS Teaching Methodology

For teaching RNS-based embedded systems, professors of embedded systems courses have to be aware of the latest improvements and enhancements at the arithmetic level. The concept of residue number-based embedded systems and the RNS potential are not widely known in this domain. Whereas the tutors who actively track publications may be aware of its potential, it is hard to integrate RNS in a systematic and organized way on embedded systems courses. The tutor should be aware of the features of each RNS component and its overall structure and abilities, as well as the method for applying it to embedded systems. Due to the multi-disciplinary nature of RNS, which is based on mathematical formulation, digital design, and computer architecture, just presenting a brief about the emerging approach of RNS-based embedded system design can confuse students and does not transmit them the ability of using modular arithmetic in RNS-based embedded systems. Moreover, the usage of RNS is manifold, and covering all its applications might not be possible. RNS can be used in both parts of embedded systems: hardware and software. The RNS structures can be embedded in single-purpose and application-specific processor architecture supported on extensions to the instruction-set architecture (ISA) [77]. Also, RNS is effectively used in parallel-programming of embedded systems as shown in RNS-based cryptography implemented on GPU [78].

With the aim of teaching the RNS approach for designing embedded systems and motivating students to work and do research on it, a systematic and organized methodology is introduced in this section. This methodology consists of two main phases: first, students should become familiar with the concept and structure of RNS. It is quite important to make students familiar with an exact and best fitted structure for embedded system applications. If only lots of references are introduced to the students without introducing an example of a selected architecture, the topic might be made arid. However, it is time consuming for a teacher to visit the complete set of RNS papers to find the best architecture that fits embedded systems and the course. Therefore, this section eases this process and introduces a simplified structure for students to learn each part in order to have a deep understanding of how to design the core of the arithmetic path of embedded processors based on RNS. Moreover, we also suggest some hints and introduce some references for researchers in the area of embedded systems who want to do more investigation in this area and start research projects.

We used the moduli set of $\{2^n - 1,\ 2^k,\ 2^n + 1\}$ as the base set to learn RNS for students, because the circuits for this RNS set are simple and there are also papers proposing embedded processor based on it [57, 77]. Although these conference papers can be pointed out as good examples of embedded systems based on RNS, there are differences in comparison with the architecture we follow here. Because students may not know well carry-save arithmetic, since they are embedded systems students and it is not a course on computer arithmetic. The second part of the proposed methodology is to learn the embedded processor

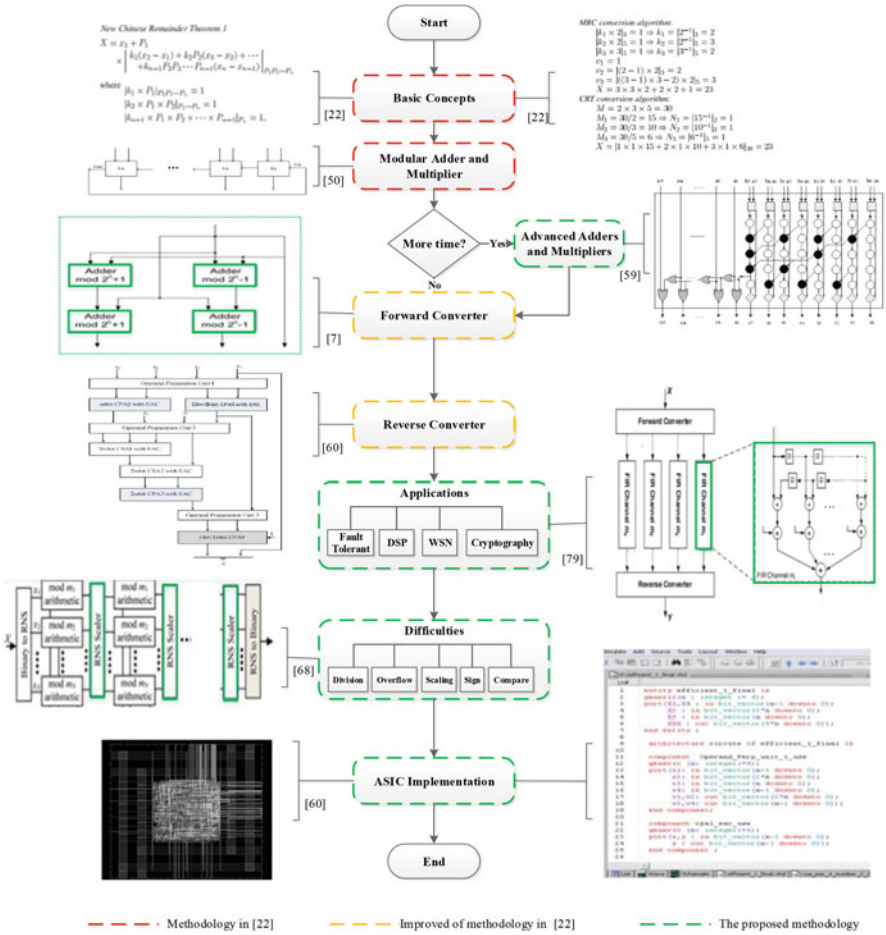

Fig. 1.2 The (hardware-oriented) methodology to teach RNS

architecture based on RNS, discuss the results, and show the superiority of RNS in real case studies through experimental results. In other words, students are taught that their knowledge in the embedded system course can be combined with modular arithmetic circuits to boost performance with high energy efficiency. The proposed methodology is depicted in Fig. 1.2, and its distinct parts are explained in the following subsections.

1.3.1 Basic Concepts

The first stage is devoted to reviewing the fundamental congruence theorems and formulas. The second chapter of the book [7] can be used with this aim, since it provides the most important formulas in a convenient way. It should be mentioned that numerical examples can play a significant role to make it easier for the students to understand the subject. Numerical examples should be used for all RNS parts, so that before achieving hardware implementations students understand the operation and functionality of each RNS part.

It is recommended that teachers first select a sample moduli set, and then show the residues of numbers in RNS in a table. Then, after each operation, the table can be used to show the correctness of operation. This is especially important in learning the difficult RNS operation's concepts. After introducing the dynamic range table, the tutor can select two RNS numbers without showing the table, and ask the students to compare them and recognize their signs. These easy questions make students familiar with RNS hard operations. Moreover, the lack of a sign bit, significance, and relation between residues can be mentioned to illustrate the difficulties in implementing inter-modulo operations. Numerical examples can be used as well as materials presented in [22] to facilitate student learning. At the end of this stage students should be able to understand the mathematical basis of RNS and the operation involved in each part, the moduli set selection procedure, and how to achieve simplified formulas of forward and reverse conversions for carefully selected moduli sets. It is also recommended that teachers give a brief explanation about RNS floating-point numbers [80]. If this point is well developed, it can be greatly suitable for energy-efficient floating-point units. This part of the curriculum was previously denoted by researchers [22]. Here, it is complemented by adding some important features such as a discussion about difficult RNS operations concepts, using a dynamic range table, and floating-point operations. Spending time only explaining formulas without considering details of hardware implementations may be the main reason for students to misunderstand the RNS capabilities, which are considered in more detail in the next sections.

1.3.2 Modular Adders and Multipliers

The second stage of the methodology is about modular adders and multipliers suitable for RNS. It is suggested that one first describes adders and then multipliers. Here, moduli $2^n - 1$ and $2^n + 1$ adders are presented and their operations are explained in detail, since they are significant and applicable in real implementations of RNS. The hardware architectures used to teach these adders can be ripple-carry adder (RCA)-based, because of their simplicity especially for beginners. Papers [33] and [50] can be good references in this stage. It is better to start with other moduli that have recently attracted research, such as $2^n \pm 3$, which can remain as

a topic for the course's projects. If more time can be spent in this stage, more advanced structures such as hybrid adders and high-radix booth recoded modulo multipliers can be presented. Besides this, talking about the circuit's parameters of distinct structures and the impact of using different structures should be certainly considered also at this stage. Tutors should help students understand the advantages and disadvantages of the different structures, and prepare them for application-specific component selection. As an example, a teacher can introduce parallel-prefix modulo adders as high-speed adders but with large power consumptions. On the other hand, ripple-carry adders with end-around carry consume much less energy at the cost of higher delay. Explaining the hybrid structures of [59] and using the experimental results to show the trade-off between delay and power consumption can clearly show the impact of adder structures on circuits design and efficiency. Similar examples for multipliers can be picked up from [36]. In this paper, the authors increase the delay of the modulo $2^n - 1$ multiplier with the aim of reducing the area and power consumption, whenever the multiplier is not on the critical path. It is interesting for students to see how circuits' parameters may change the architecture of the multiplier.

1.3.3 Forward Converter

At the time students are familiar with adders and multipliers, one can explain the forward converter implementation. Since the forward converter is typically composed of multi-operand modular adders, it is suggested to teach this part after explaining adders, on the contrary to what was previously proposed [22]. Since understanding forward converters hardware is simpler after knowing well modular additions, the professors can easily show the complete architectures of forward converters for different moduli.

1.3.4 Reverse Converter

Afterwards, it's the moment to explain one of the most vital and complex operations in RNS. It is absolutely necessary students become familiar with the design and implementation of reverse converters. Reverse converters heavily depend on the moduli set. Many other factors such as the adopted algorithm and components also affect reverse converter performance. During the first stage, students became familiar with moduli set selection and different reverse conversion mathematical algorithms. Herein, the aim is to show the students how to implement simplified final reverse conversion equations using modular adders. It is not possible to teach this part in a general way, with its all details, because of the dependency between the converter architecture and the selected moduli set. Therefore, papers with specific

moduli sets (see Table 1.1) can be a good choice for teaching this section. A helpful exercise for this part is to pick up another specific moduli set, and ask students to find the simplified formulas and implement them.

1.3.5 Application: RNS-Based Embedded Systems Design

After familiarizing the students with the interesting features and structures of the residue number system, it is possible to show them how RNS can be used in embedded system design. First, the teacher can depict a structure of a typical embedded system architecture with the usual parts, such as an analog-to-digital converter (ADC), a digital-to-analog converter (DAC), a DSP, cryptography processing cores, and memory. Then, he can describe how to use the corresponding RNS structures to enhance the embedded system.

In the past, analog signals were converted to digital ones using regular ADCs, and then the digital signal transformed to the residues using forward converters. However, recently analog to residue converters (ARC) [81] have been designed in order to directly convert analog signals to digital residues. Therefore, the delay and cost of forward converters can be included in the ADC with this approach. However, all the elements of the system should operate over residues. Similarly, a digital residues to analog signal converter has been proposed (RAC) [82] that eliminates the need of a reverse converter. Various RNS-based DSPs [5] and cryptographic [10] processing circuits together with RNS-based memory [83] are available which can be used in the structure of an embedded system to achieve distinct applications.

At this point, the required concepts for understanding RNS as a part of a computer arithmetic and embedded system course are described. Additionally, if more time can be spent on teaching RNS, the two following parts can also be introduced into the proposed methodology.

1.3.6 Hard RNS Operations

In some computation-intensive applications, other operations such as sign detection, magnitude comparison, overflow detection, scaling, and division are required. These hard RNS operations were not covered in previous syllabus and books. Although there are suggested methods to overcome the difficulties and implement them, they have not yet been efficiently solved. Consequently, on-going research can be found in these fields. The tutors are suggested to explain the procedure of designing each of them based on recent papers (see Table 1.1). The advantage of explaining difficult RNS operations is not only to answer students' questions about them but also to motivate them to research on these challenging parts.

1.3.7 ASIC/FPGA Implementation

The final and optional, but also recommended, stage of the proposed methodology is showing the complete process of designing a hardware implementation of an RNS system, which includes the design, hardware description language (HDL) coding, functional verification using simulation tools, synthesis of the logical circuits, and experimental evaluation. Illustrating these phases using an example can provide a practical view to students and motivate them. It is supposed that students have background in digital circuits design and HDL coding. However, some brief descriptions about different types of HDL coding and the selection of either structural or behavioral modeling according to the situation can be helpful. Moreover, this stage can be a good opportunity for discussion about the effect of careful and wisely placement of different hardware components in the structure of RNS circuits to achieve suitable trade-offs between area, delay, and power consumption on application-specific integrated circuits (ASIC) or field-programmable gate arrays (FPGA). Note that it is not mandatory to implement all parts of RNS, since it is time consuming. It is enough to show a sample, e.g., prepared HDL codes for one important part of RNS, such as for the reverse converter, which can be complemented with ASIC or FPGA implementation results.

1.4 Conclusion

The RNS special abilities, such as carry-free arithmetic and fault-tolerant computations, fit high-performance modern embedded systems requirements. Research on RNS has been applied to different fields over the past decades, proving its importance. Unfortunately, the traditional RNS teaching methods focused on the mathematical foundations, which is not appealing for students interested in embedded systems. In order to solve this issue, a methodology is proposed, which integrates hardware structures, implementations, and applications together with the mathematical basis of RNS. By applying this method, it is easier for engineers to understand the potential of RNS, and motivate them to apply RNS for designing more efficient embedded systems.

References

1. P. Marwedel, *Embedded system design: embedded systems foundations of cyber-physical systems* (Springer International Publishing, Dordrecht, 2011)
2. S. Yang, Toward a wireless world. IEEE Technol. Soc. Mag. **26**(2), 32–42 (2007)
3. H.L. Garner, The residue number system. IRE Trans. Electron. Comput. **8**(2), 140–147 (1959)
4. T. Stouraitis, V. Paliouras, Considering the alternatives in low-power design. IEEE Circuits Devices **7**, 23–29 (2001)

5. C.H. Chang, A.S. Molahosseini, A.A.E. Zarandi, T.F. Tay, Residue number systems: a new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. IEEE Circuits Syst. Mag. **15**(4), 26–44 (2015)
6. P.V.A. Mohan, *Residue number systems: Algorithms and architectures* (Kluwer, Boston, 2002)
7. A. Omondi, B. Premkumar, *Residue number systems: Theory and implementations* (Imperial College Press, London, 2007)
8. J. Chen, J. Hu, Energy-efficient digital signal processing via voltage-over scaling-based residue number system. IEEE Trans. Very Large Scale Integr. Syst. **21**(7), 1322–1332 (2013)
9. S. Antão, L. Sousa, The CRNS framework and its application to programmable and reconfigurable cryptography. ACM Trans. Archit. Code Optim. **9**(4), 1–33 (2013)
10. L. Sousa, S.F. Antão, P.S.A. Martins, Combining residue arithmetic to design efficient cryptographic circuits and systems. IEEE Circuits Syst. Mag., to appear, 2016
11. T.F. Tay, C.H. Chang, A non-iterative multiple residue digit error detection and correction algorithm in RRNS. IEEE Trans. Comput. **65**(2), 396–408 (2016)
12. R. Ye1, A. Boukerche, H. Wang, X. Zhou, B. Yan, RESIDENT: a reliable residue number system-based data transmission mechanism for wireless sensor networks, Springer Wireless Netw., to appear, 2016
13. X. Zheng, B. Wang, C. Zhou, X. Wei, Q. Zhang, Parallel DNA arithmetic operation with one error detection based on 3-moduli set, IEEE Trans. Nano Biosci., to appear, 2016
14. A. Celesti, M. Fazio, M. Villari, A. Puliafito, Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. J. Netw. Comput. Appl. **59**, 208–218 (2016)
15. M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, K. Navi, Efficient RNS implementation of elliptic curve point multiplication over GF(p). IEEE Trans. Very Large Scale Integr. Syst. **21**(8), 1545–1549 (2013)
16. Y. Liu, E.M.K. Lai, Moduli set selection and cost estimation for RNS-based FIR filter and filter bank design. Des. Autom. Embedded Syst. **9**, 123–139 (2004)
17. M. Dasygenis, I. Petrousov, A generic moduli selection algorithm for the residue number system, in *Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015
18. A. Persson, L. Bengtsson, Forward and reverse converters and moduli set selection in signed-digit residue number systems. J. Signal Process. Syst. **56**(1), 1–15 (2009)
19. J.C. Bajard, M. Kaihara, T. Plantard, Selected RNS bases for modular multiplication, in *Proceedings of the 19th IEEE symposium on Computer Arithmetic*, 2009
20. J. Ramírez, U. Meyer-Bäse, F. Taylor, A. García, A. Lloris, Design and implementation of high-performance RNS wavelet processors using custom IC technologies. J. VLSI Signal Process. Syst. Signal Image Video Technol. **34**(3), 227–237 (2003)
21. F. Gandino, F. Lamberti, G. Paravati, J.C. Bajard, P. Montuschi, An algorithmic and architectural study on montgomery exponentiation in RNS. IEEE Trans. Comput. **61**(8), 1071–1083 (2012)
22. K. Navi, A.S. Molahosseini, M. Esmaeildoust, How to teach residue number system to computer scientists and engineers. IEEE Trans. Educ. **54**(1), 156–163 (2011)
23. F. Pourbigharaz, H.M. Yassine, Simple binary to residue transformation with respect to 2/sup m/+1 moduli. IEE Proc. Circuits Devices Syst. **141**(6), 522–526 (1994)
24. S. Piestrak, Design of residue generators and multioperand modular adders using carry-save adders. IEEE Trans. Comput. **43**(1), 68–77 (1994)
25. A.B. Premkumar, E.L. Ang, E.M.-K. Lai, Improved memoryless RNS forward converter based on the periodicity of residues. IEEE Trans. Circuits Syst. II Express Briefs **53**(2), 133–137 (2006)
26. C. Efstathiou, N. Moschopoulos, K. Tsoumanis, K. Pekmestzi, On the design of configurable modulo 2ˆn ± 1 residue generators, in *Proceedings of euromicro conference on digital system design (DSD)*, 2012, pp. 50–56
27. H.T. Vergos, D. Bakalis, C. Efstathiou, Fast modulo $2^n + 1$ multi-operand adders and residue generators. Integr. VLSI J. **43**(1), 42–48 (2010)

28. A.A. Hiasat, Arithmetic binary to residue encoders for moduli ($2^n \pm 2^k + 1$). IEE Proc. Comput. Digit. Tech. **150**(6), 369–374 (2003)
29. P.M. Matutino, R. Chaves, L. Sousa, Arithmetic-based binary-to-RNS converter modulo ($2^n \pm k$) for *jn*-bit dynamic range. IEEE Trans. Very Large Scale Integr. Syst. **23**(3), 603–607 (2015)
30. K. Shirakawa, T. Uemura, Y. Iguchi, A realization method of forward converters from multiple-precision binary numbers to residue numbers with arbitrary mutable modulus, in *Proceedings of IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, 2011
31. G. Petrousov, M. Dasygenis, A unique network EDA tool to create optimized ad hoc binary to residue number system converters, in *Proceedings of International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2014.
32. J.Y.S. Low, C.H. Chang, A new approach to the design of efficient residue generators for arbitrary moduli. IEEE Trans. Circuits and Syst.–I **60**(9), 2366–2374 (2013)
33. R. Zimmermann, Efficient VLSI implementation of modulo ($2^n \pm 1$) addition and multiplication, in *Proceedings of the IEEE International Symposium on Computer Arithmetic*, 1999, pp. 158–167.
34. L. Kalampoukas et al., High-speed parallel-prefix modulo $2^n - 1$ adders. IEEE Trans. Comput. **49**(7), 673–680 (2000)
35. R.A. Patel, M. Benaissa, S. Boussakta, Fast parallel-prefix architectures for modulo $2^n - 1$ addition with a single representation of zero. IEEE Trans. Comput. **56**(11), 1484–1492 (2007)
36. R. Muralidharan, C.H. Chang, Radix-8 booth encoded modulo $2^n - 1$ multipliers with adaptive delay for high dynamic range residue number system. IEEE Trans. Circuits Syst.–I **58**(5), 982–993 (2011)
37. L.S. Didier, L. Jaulmes, Fast modulo $2^n - 1$ and $2^n + 1$ adder using carry-chain on FPGA, in Proc. of Asilomar Conference on Signals, Systems and Computers, 2013, pp. 1155–1159
38. R. Muralidharan, C.H. Chang, Area-power efficient modulo $2^n - 1$ and modulo $2^n + 1$ multipliers for $\{2^n - 1,\ 2^n,\ 2^n + 1\}$ based RNS. IEEE Trans. Circuits Syst.–I **59**(10), 2263–2274 (2012)
39. H.T. Vergos, G.N. Dimitrakopoulos, On modulo $2^n + 1$ adder design. IEEE Trans. Comput. **61**(2), 173–186 (2012)
40. H.T. Vergos, Area-time efficient end-around inverted carry adders. Integr. VLSI J. **45**(4), 388–394 (2012)
41. S.M. Mirhosseini, A.S. Molahosseini, M. Hosseinzadeh, L. Sousa, P. Martins, A reduced-bias approach with a lightweight hard-multiple generator to design radix-8 modulo $2^n + 1$ multiplier. IEEE Trans. Circuits Syst.-II, to appear, 2016
42. C.H. Chang, S. Menon, B. Cao, T. Srikanthan, A configurable dual moduli multi-operand modulo adder, in *Proceedings of IEEE International Symposium on Circuits and Systems*, 2005, pp. 1630–1633
43. E. Vassalos, D. Bakalis, H.T. Vergos, On the design of modulo $2^n \pm 1$ subtractors and adders/subtractors. Circuits Syst. Signal Process. **30**(6), 1445–1461 (2011)
44. C. Efstathiou, K. Tsoumanis, K. Pekmestzi, I. Voyiatzis, Modulo $2^n \pm 1$ fused add-multiply units, in *Proceedings of IEEE Computer Society Annual Symposium on VLSI,* 2015, pp. 91–96
45. H.T. Vergos, D. Bakalis, Area-time efficient multi-modulus adders and their applications. Microprocessors Microsystems Embedded Hardware Des. **36**(5), 409–419 (2012)
46. R. Muralidharan, C.H. Chang, Radix-4 and radix-8 Booth encoded multi-modulus multipliers. IEEE Trans. Circuits Syst.–I **60**(11), 2940–2952 (2013)
47. H. Pettenghi, S. Cotofana, L. Sousa, Efficient method for designing modulo $\{2^n \pm K\}$ multipliers. J. Circuits Syst. Comput. **23**(1), 1450001 (2014)
48. G. Jaberipur, S.H.F. Langroudi, (4 + 2\log n)\delta G parallel prefix modulo—($2^n - 3$) adder via double representation of residues in [0, 2]. IEEE Trans. Circuits Syst.-II **62**(6), 583–587 (2015)
49. S. Ma, J.H. Hu, C.H. Wang, A novel modulo $2^n - 2^k - 1$ adder for residue number system. IEEE Trans. Circuits Syst.-I **60**(11), 2962–2972 (2013)

50. J.L. Beuchat, Some modular adders and multipliers for field programmable gate arrays, in *Proceedings of Parallel and Distributed Processing Symposium*, 2003, pp. 1–8
51. R.A. Patel, M. Benaissa, N. Powell, S. Boussakta, Novel power-delay-area-efficient approach to generic modular addition. IEEE Trans. Circuits Syst.-I **54**, 1279–1292 (2007)
52. H. Nakahara, T. Sasao, A deep convolutional neural network based on nested residue number system, in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–6
53. Y. Wang, X. Song, M. Aboulhamid, H. Shen, Adder based residue to binary numbers converters for $(2^n - 1, 2^n, 2^n + 1)$. IEEE Trans. Signal Process. **50**(7), 1772–1779 (2002)
54. P.V.A. Mohan, RNS-to-binary converter for a new three-moduli set $\{2^{n+1} - 1, 2^n, 2^n - 1\}$. IEEE Trans. Circuits Syst.-II **54**(9), 775–779 (2007)
55. A. Hariri, K. Navi, R. Rastegar, A new high dynamic range moduli set with efficient reverse converter. J. Comput. Math. Appl. **55**(4), 660–668 (2008)
56. A.S. Molahosseini, K. Navi, O. Hashemipour, A. Jalali, An efficient architecture for designing reverse converters based on a general three-moduli set. J. Syst. Archit. **54**, 929–934 (2008)
57. R. Chaves, L. Sousa, Improving RNS multiplication with more balanced moduli sets and enhanced modular arithmetic structures. IET Comput. Digital Tech. **1**(5), 472–480 (September 2007)
58. P. Patronik, S.J. Piestrak, Design of reverse converters for general RNS moduli sets $\{2^k, 2^n - 1, 2^n + 1, 2^{n+1} - 1\}$ and $\{2^k, 2^n - 1, 2^n + 1, 2^{n-1} - 1\}$ (n even). IEEE Trans. Circuits Syst.-I **61**(6), 1687–1700 (2014)
59. A.A.E. Zarandi, A.S. Molahosseini, M. Hosseinzadeh, S. Sorouri, S.F. Antão, L. Sousa, Reverse converter design via parallel-prefix adders: novel components, methodology and implementations. IEEE Trans. Very Large Scale Integr. Syst. **23**(2), 374–378 (2015)
60. A.S. Molahosseini, K. Navi, C. Dadkhah, O. Kavehei, S. Timarchi, Efficient reverse converter designs for the new 4-moduli sets $\{2^n - 1, 2^n, 2^n + 1, 2^{2n+1} - 1\}$ and $\{2^n - 1, 2^n + 1, 2^{2n}, 2^{2n} + 1\}$ based on new CRTs. IEEE Trans. Circuits Syst.-I **57**(4), 823–835 (2010)
61. L. Sousa, S. Antao, MRC-based RNS reverse converters for the four-moduli sets $\{2^n + 1, 2^n - 1, 2^n, 2^{2n+1} - 1\}$ and $\{2^n + 1, 2^n - 1, 2^{2n}, 2^{2n+1} - 1\}$. IEEE Trans. Circuits Syst. II **59**(4), 244–248 (2012)
62. H. Pettenghi, R. Chaves, L. Sousa, RNS reverse converters for moduli sets with dynamic ranges up to $(8n + 1)$-bit. IEEE Trans. Circuits Syst. I **60**(6), 1487–1500 (2013)
63. C.C.H. Chang, T. Srikanthan, A residue-to-binary converter for a new 5-moduli set. IEEE Trans. Circuits Syst.–I **54**(5), 1041–1049 (2007)
64. M.H. Sheu, S.H. Lin, C. Chen, S.W. Yang, An efficient VLSI design for a residue to binary converter for general balance moduli $\{2^n - 1, 2^n + 1, 2^n - 3, 2^n + 3\}$. IEEE Trans. Circuits Syst.-II **51**(3), 152–155 (2004)
65. A.A.E. Zarandi, A.S. Molahosseini, L. Sousa, M. Hosseinzadeh, An efficient component for designing signed reverse converters for a class of RNS moduli sets with composite form {2K, 2P-1}, IEEE Trans. Very Large Scale Integr. Syst., to appear, 2016
66. L.C. Tai, C.F. Chen, Technical note. Overflow detection in a redundant residue number system. IEE Proc. Comput. Digital Tech. **131**(3), 97–98 (1984)
67. D. Younes, P. Steffan, Universal approaches for overflow and sign detection in residue number system based on $\{2^n–1, 2^n, 2^n + 1\}$, in *Proceedings of Eighth International Conference on Systems*, 2013, pp. 77–81
68. C.H. Chang, J.Y.S. Low, Simple, fast and exact RNS scaler for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. IEEE Trans. Circuits Syst.–I **58**(11), 2686–2697 (2011)
69. T.F. Tay, C.H. Chang, J.Y.S. Low, Efficient VLSI implementation of 2^n scaling of signed integer in RNS $\{2^n - 1, 2^n, 2^n + 1\}$. IEEE Trans. Very Large Scale Integr. Syst. **21**(10), 1936–1940 (2013)
70. L. Sousa, 2^n RNS scalers for extended 4-moduli sets. IEEE Trans. Comput. **64**(12), 3322–3334 (2015)
71. S. Kumar, C.H. Chang, A new fast and area-efficient adder-based sign detector for RNS $\{2^n - 1, 2^n, 2^n + 1\}$. IEEE Trans. Very Large Scale Integr. Syst. **24**(7), 2608–2612 (2016)

72. L. Sousa, P.S.A. Martins, Efficient sign identification engines for integers represented in the RNS extended 3-moduli set $\{2^n - 1, 2^{n+k}, 2^n + 1\}$. Electron. Lett. **50**(16), 1138–1139 (2014)
73. M. Xu, Z. Bian, R. Yao, Fast sign detection algorithm for the rns moduli set $\{2^{n+1} - 1, 2^n - 1, 2^n\}$. IEEE Trans. Very Large Scale Integr. Syst. **23**(2), 379–383 (2015)
74. Z. Torabi, G. Jaberipur, Low-power/cost rns comparison via partitioning the dynamic range. IEEE Trans. Very Large Scale Integr. Syst. **24**(5), 1849–1857 (2016)
75. L. Sousa, P.S.A. Martins, sign detection and number comparison on rns 3-moduli sets $\{2^n - 1, 2^{n+x}, 2^n + 1\}$, Circuits Syst. Signal Process., to appear, 2016
76. S. Kumar, C.H. Chang, T.F. Tay, New algorithm for signed integer comparison in $\{2n + k, 2n - 1, 2n + 1, 2n \pm 1 - 1\}$ and its efficient hardware implementation, IEEE Trans. Circuits Syst.–I, to appear, 2016
77. R. Chokshi, K.S. Berezowski, A. Shrivastava, S.J. Piestrak, Exploiting residue number system for power-efficient digital signal processing in embedded processors, in *Proceedings of the International conference on Compilers, architecture, and synthesis for embedded systems*, 2009, pp. 19–28
78. S.F. Antão, J.C. Bajard, L. Sousa, RNS based elliptic curve point multiplication for massive parallel architectures. Comput. J. **55**(5), 629–647 (2012)
79. E. Vassalos, D. Bakalis, CSD-RNS-based single constant multipliers. J. Signal Process. Syst. **67**(3), 255–268 (2012)
80. J.S. Chiang, M. Lu, Floating-point numbers in residue number systems. Comput. Math. Appl. **22**(10), 127–140 (1991)
81. C.H. Vun, A.B. Premkumar, W. Zhang, A new RNS based DA approach for inner product computation. IEEE Trans. Circuits Syst.-I **60**(8), 2139–2152 (2013)
82. O. Abdelfattah, Data conversion in residue number system, M.E. Thesis, Department of Electrical and Computer Engineering, McGill University, Montreal, 2011
83. N.Z. Haron, S. Hamdioui, Redundant residue number system code for fault-tolerant hybrid memories. ACM J. Emerging Technol. Comput. Syst. **7**(1), 19 (2011)

Chapter 2
RNS-Based Embedded Processor Design

Pedro Miguens Matutino, Ricardo Chaves, and Leonel Sousa

2.1 Introduction

Digital consumer electronics represents a major sector of today's world economy in a wide range of products. Digital Signal Processors (DSPs) are a key component of many digital consumer products in several application domains, such as telecommunications, digital audio, video and imaging, speech processing, cryptography, and multimedia [1–9]. The design of DSPs has evolved rapidly over the last decade, driven by the ever-increasing need to improve performance and balance power consumption, flexibility, and integration of more features.

Conventional carry propagation arithmetic, based on a weighted number system, is a widely employed and well-studied approach. However, the dependencies and the need to perform the full weighted propagation of the carry cause a significant delay in arithmetic computation, preventing the design of arithmetic units with improved performance and enhanced efficiency. Therefore, Residue Number System (RNS) has been proposed as an alternative arithmetic system for computational intensive applications. RNS is a non-weighted numbering system that uses the remainders of the division by co-prime moduli, which compose a moduli set, to represent an integer value. The multiple and smaller values used in the RNS representation allow parallelism, high-speed, and low energy computation by reducing the hardware requirements to process data. The higher parallelism results from the fact that the multiplications and additions are performed independently on each individual residue channel.

P.M. Matutino (✉)
ISEL - IPL, INESC-ID, Rua Alves Redol 9, Lisboa, Portugal
e-mail: pmmm@sips.inesc-id.pt

R. Chaves • L. Sousa
IST - Universidade de Lisboa, INESC-ID, Rua Alves Redol 9, Lisboa, Portugal
e-mail: ricardo.chaves@inesc-id.pt; las@inesc-id.pt

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_2

In order to facilitate the usage of RNS, its adaptability, and widen its applicability, a generic, efficient, and scalable RNS architecture supporting moduli sets with an arbitrary number of channels is required. In this chapter, a unified structure for a scalable RNS processor based on $\{2^n \pm k_i\}$ ($k_i \leq 2^{n/2} - 1$) moduli channels is proposed, allowing the design of RNS with any moduli set of the form $\{2^n \pm k_0, \cdots, 2^n \pm k_j\}$, where $j \in \mathbb{N}_0^+$. The considered moduli set allows to arbitrary increase the number of RNS channels and consequently, increase the Dynamic Range (DR) or reduce the width of the channels leading to a reduction in delay and area cost. The proposed RNS architecture provides a complete processing system supporting the computation of conversions, namely from binary-to-RNS, RNS-to-binary, and base extension. The proposed architecture has the capability of computing the conversion by reusing the arithmetic units of each channel, thus allowing the design of a more compact RNS processor. The arithmetic operations supported at the channel level include the addition, subtraction, and multiplication, with accumulation capability. For the reverse conversion two algorithms are considered: one based on the Chinese Remainder Theorem (CRT) and another on the Mixed Radix Converter (MRC). An Instruction Set Architecture (ISA) is also proposed, in order to provide a simple and independent interface to the proposed generic RNS architecture. The proposed RNS architecture and resulting processor implement the ISA without the need of explicitly defining the number of channels, allowing to generate generic code independently of the DR, making this transparent to the programmer.

This chapter is organized as follows. Section 2.2 presents the proposed RNS architecture, followed by the definition of the ISA in Sect. 2.3. The arithmetic operations including the conversions to and from binary are described in Sect. 2.4. Section 2.5 details the control units of the proposed processor. Section 2.6 presents a performance comparison with the most relevant state of the art, and Sect. 2.7 summarizes the main conclusions regarding the work herein proposed.

2.2 Processor Architecture

The RNS processor herein described is to be used as a coprocessor of generic Central Processing Units (CPUs), targeting intensive arithmetic computations. The first step of an RNS computation process is the conversion from binary-to-RNS, followed by the arithmetic operations performed in each channel, and finally the conversion from RNS-to-binary. Consequently, one of the main tasks of the RNS processor is to receive code and data (typical in binary) from the CPU and send it back after processing. Towards a more compact structure the approach herein presented considers that the number of required conversions is less than the number of arithmetic operations performed in the channels, which is typical of several applications, such as cryptographic operations. Furthermore, this allows for more compact structures since conversion steps can be executed in the hardware resources of the modular channels. Consequently, these conversions are executed in a sequential manner.

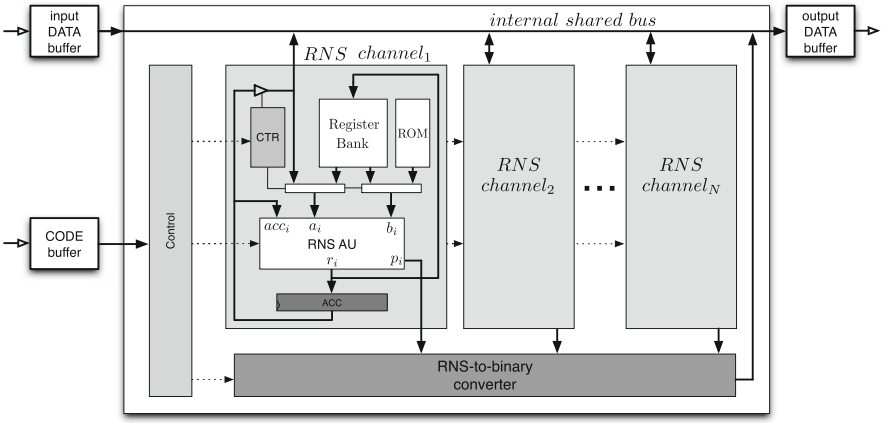

Fig. 2.1 Proposed RNS architecture

The proposed architecture is organized in three main blocks, depicted in Fig. 2.1: (1) channel arithmetic blocks; (2) RNS-to-binary converter; and (3) global control unit. The channel arithmetic blocks perform the modular additions, subtractions, and multiplications on each channel. These were the chosen modular operations, since they are the basic arithmetic operations required in digital signal processing [3, 6, 7, 10, 11] and in applications, such as in asymmetrical cryptography [8, 9]. These arithmetic blocks are also used to perform the binary-to-RNS conversion.

The conversion from RNS-to-binary is herein performed in a two step algorithm. The first step converts the RNS representation into a Mixed Radix System (MRS) representation, followed by a second step that converts the MRS values into the binary value. The MRS conversion is fully computed in the arithmetic moduli channels. The second step of the reverse conversion, which cannot be performed using just the arithmetic operations available in the RNS channels, is computed on the RNS-to-binary converter module, containing the additional logic required for the conversion.

The MRS operation is herein also used to perform base extensions, implementing the conversion between moduli sets. The base extension allows to convert number representation between two moduli sets without performing the conversion from RNS-to-binary in the origin moduli set, and the conversion from binary-to-RNS in the destination. The base extension operations are intensively used in the Montgomery Modular Multiplication [Montgomery Multiplication (MM)] [12, 13], one of the main operations when computing asymmetrical cryptographic algorithms [14] on RNS. These base extension operations require more than one moduli set to be computed, although each RNS processor only has one moduli set, consequently the proposed RNS computation requires more than one processor. Given this, the proposed RNS architecture allows multiple RNS processors in the RNS computing system.

Considering that the RNS architecture is to be used as a coprocessor of generic CPUs, the communication between them can be established by Fist In First Out (FIFO) buffers, implementing the communication and data flow control, also acting as synchronization points.

The following sections describe the considered ISA and the proposed architecture, detailing the designed computation logic used to perform the binary-to-RNS conversion, the modular arithmetic operations, and RNS-to-binary conversion.

2.3 Instruction Set Architecture

In order to provide a simple and independent interface to the proposed RNS architecture, an ISA is defined. This ISA considers a scalable RNS architecture, without the need to explicitly define the number of channels of the implemented RNS and resulting DR. This allows to generate generic code, making it transparent to the programmer. Apart from the conversion operations, the programmer can code as for a typical arithmetic binary processor, without the need to know any details of RNS arithmetic. The defined ISA supports the following arithmetic operations: addition, subtraction, and multiplication, with and without accumulation capability, as further described in Sect. 2.4.2. Alongside these arithmetic operations, conversion operations to and from binary, and base extension, are also considered and further detailed in Sects. 2.4.1 and 2.4.3. Since these conversion operations are implemented by arithmetic computations, the number of these computations and consequently the number of clocks cycles are directly dependent on the number of channels. The conversion computation requires specific constants, which depend on the used moduli set and do not change for the entire life-cycle of the RNS processor. If these are defined programmatically and sent by the CPU to the coprocessor, overhead is introduced due to data transfers. Considering this, and in order to maintain a simple interface, these constants and the operation micro-code are built-in into the RNS processor. With this, the computation of the conversions to and from binary, and possible base extension, can be performed without external support. As stated before, the conversion operations are implemented in a sequential manner, requiring more than one cycle to be completed. Given this, the RNS processor instructions are divided into two main categories: namely, single cycle and multi-cycle instructions. The first one requires a single clock cycle to complete the arithmetic computation, independently of the number of RNS channels, while the multi-cycle instructions directly dependent on the number of channels, imposing a number of cycles equal to the number of channels.

Herein, a 32-bit instruction size is considered, since it is the most common length used nowadays on embedded CPUs [15]. The proposed instruction format is divided in four fields, namely: (1) register source one (*rs*1); (2) register source two (*rs*2); (3) destination register (*rd*); and (4) operation code (*opcode*), as depicted in Fig. 2.2. Each register field has a length of 8 bits, allowing to address up to 256 registers. The operation code field is divided into two sub-fields: the first one used for RNS processor enabling, and the second one to define the arithmetic operation.

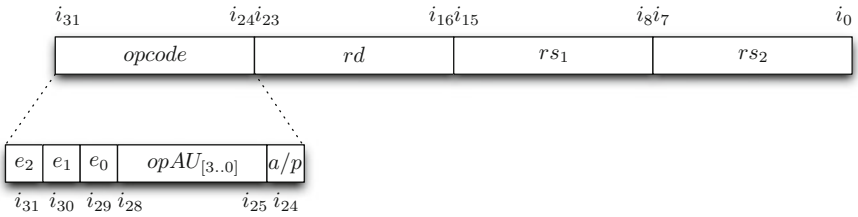

Fig. 2.2 Instruction format of the proposed RNS processor

Table 2.1 RNS processor ISA

#	mnemonic	instruction										
		i_{31}	i_{30}	i_{29}	i_{28}	i_{27}	i_{26}	i_{25}	i_{24}	$i_{23}\cdots i_{16}$	$i_{15}\cdots i_8$	$i_7\cdots i_0$
1	add/a rd,rs1,rs2	e_2	e_1	e_0	0	0	0	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
2	sub/a rd,rs1,rs2	e_2	e_1	e_0	0	0	0	1	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
3	mul/a rd,rs1,rs2	e_2	e_1	e_0	0	0	1	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
4	aadd/a rd,rs1,rs2	e_2	e_1	e_0	1	0	0	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
5	asub/a rd,rs1,rs2	e_2	e_1	e_0	1	0	0	1	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
6	amul/a rd,rs1,rs2	e_2	e_1	e_0	1	0	1	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
7	sadd/a rd,rs1,rs2	e_2	e_1	e_0	1	1	0	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
8	ssub/a rd,rs1,rs2	e_2	e_1	e_0	1	1	0	1	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
9	smul/a rd,rs1,rs2	e_2	e_1	e_0	1	1	1	0	a	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$
10	bToRNS rd	e_2	e_1	e_0	0	1	0	0	p	$rd_{[7:0]}$		
11	mToRNS rd	e_2	e_1	e_0	0	1	0	1	p	$rd_{[7:0]}$		
12	rnsToM rd,rs1	e_2	e_1	e_0	0	1	1	0	p	$rd_{[7:0]}$	$rs1_{[7:0]}$	
13	mToBIN rs1	e_2	e_1	e_0	0	1	1	1	0		$rs1_{[7:0]}$	

The RNS architecture allows multiple RNS processors in the RNS computing system using processor enabling flags, up to three are allowed with this codification. This maximum value is considered given the typical RNS based cryptographic implementations [16]. The arithmetic operations (*opAU*) are specified by four bits, with an additional bit to enable accumulation, as depicted in Fig. 2.2 and detailed in Table 2.1. This additional bit is also used to define if the internal shared bus propagates the data, used in multi-cycle operations such as the base extension, allowing for data transfers between two RNS processors.

Table 2.1 details the proposed ISA, divided into single and multi-cycle instructions. The first three operations (1–3) implement the modular addition, subtraction, and multiplication between *rs*1 and *rs*2, storing the result in the defined *rd*, and in the accumulator if the accumulator flag *a* is active. The following three operations (4–6) preceded by '*a*' execute the same operations as the previous ones, with the particularity that the accumulator value is also added. Once more the final value is stored in the destination register, and in the accumulator if the flag *a* is set. The last

three single cycle instructions (7–9), preceded by '*s*', have the same behaviour as the previous ones, but instead of adding the result of the operation between *rs*1 and *rs*2, subtracts that value from the accumulator. In these instructions, the flag *a* has the same functionality, i.e., besides the destination register, the result of the operation is also stored in the accumulator if *a* is set.

The first two multi-cycle instructions (10–11) convert from binary and from a mixed-radix representation, respectively, to the moduli set in the RNS processor. These instructions receive data from the shared bus and require only the definition of the destination register. The third multi-cycle instruction (*mToRNS rd,rs1*) executes the conversion from RNS-to-MRS, requiring a source register and a destination register. This instruction is described in detail in Sect. 2.4.3. Note that this conversion changes the representation from RNS to a Mixed Radix System. The instruction *mToBin* (MRS-to-binary) is used to perform the second step of the conversion from RNS-to-binary, that was started by a RNS-to-MRS instruction. Note that, herein the RNS-to-binary conversion is considered as a two step operation, as further described in Sect. 2.4.3. The MRS-to-binary instruction only requires the definition of *rs*1, without any destination register, since the computed values are written in output data buffer.

In order to support this ISA, each RNS channel is organized in four main blocks, as depicted in Fig. 2.1, namely: (1) register bank and accumulator, for data storage; (2) constant memory Read-Only-Memory (ROM); (3) RNS Arithmetic Unit (AU), where the modular arithmetic operations are executed; and (4) control (CTR).

The data information flow from and to the CPU defines the behaviour of the global control unit. Remember that the RNS architecture is to be used as a coprocessor of generic CPUs, and one of its main tasks is to receive code and data from the CPU and send it back after processing. In order to simplify the communication and data flow control, buffers are herein used as synchronization points. With this, the RNS processor stalls whenever data or code to be processed are missing. Furthermore, when the output data buffers are full the RNS processor also stalls, avoiding in this way losing processed data. In order to implement this data control flow, one unidirectional and one bidirectional FIFO buffers are herein considered for code and data, respectively, as depicted in Fig. 2.1.

2.4 RNS Arithmetic Operations

This section describes the formulation and resulting modular hardware structures required to compute the binary-to-RNS conversion, the modular arithmetic operations in each moduli channel, the RNS-to-binary conversion, and base extension operation.

2.4.1 Binary-to-RNS Conversion

The first typical operation in an RNS processor is the conversion to the RNS representation. This is achieved by computing the remainder of the division of each operand by $\{2^n \pm k_i\}$. The residue modulo $\{2^n \pm k_i\}$ of the input value X can be achieved by computing the integer division of X by $\{2^n \pm k_i\}$. However, obtaining the remainder in this way is a costly operation. Nevertheless, this operation can be implemented considering only modular addition operations, such that the residue x_i of an integer X with $(N \cdot n)$-bit inputs can be calculated as:

$$x_i = \langle X \rangle_{2^n \pm k_i} = \left\langle \sum_{b=0}^{N \cdot n-1} \left\langle 2^b \cdot X_{[b]} \right\rangle_{2^n \pm k_i} \right\rangle_{2^n \pm k_i}, \tag{2.1}$$

where $X_{[b]}$ represents the bth-bit of the binary integer X, and $\langle X \rangle_{2^n \pm k_i}$ represents the residue of X modulo $\{2^n \pm k_i\}$.

Given Eq. (2.1), the conversion of a binary number X with $(N \cdot n)$ bits to RNS for modulo $\{2^n - k\}$ can be computed as:

$$\langle X \rangle_{2^n - k} = \left\langle \sum_{i=0}^{N-1} k^i X_{[(i+1) \cdot n-1:i \cdot n]} \right\rangle_{2^n - k}. \tag{2.2}$$

where $X_{[b:c]}$ represents the bit vector from bth-bit to cth-bit. Similarly to modulo $\{2^n - k\}$, the binary-to-RNS conversion modulo $\{2^n + k\}$ can be computed as:

$$\langle X \rangle_{2^n + k} = \left\langle \sum_{i=0}^{\lfloor \frac{N-1}{2} \rfloor} k^{2i} X_{[(2i+1) \cdot n-1:2i \cdot n]} + \sum_{i=0}^{\lfloor \frac{N-2}{2} \rfloor} k^{2i+1} \overline{X_{[(2i+2) \cdot n-1:(2i+1) \cdot n]}} + \right.$$
$$\left. + \sum_{i=0}^{\lfloor \frac{N-2}{2} \rfloor} k^{2i+1} (k+1) \right\rangle_{2^n + k}. \tag{2.3}$$

Note that $\overline{X}$ represents the not bitwise operation, and the last term is a constant value that can be pre-calculated.

2.4.2 Arithmetic Channels

Arithmetic operations in Residue Number System are one of the most important aspects to take into account when optimizing computations. An RNS is composed of several smaller arithmetic channels. Typically, the modular arithmetic structures, implementing these channels, are more complex than the binary equivalent with the

same bit width. A suitable implementation of these arithmetic structures can lead to overall improvements, compensating the conversion overheads. Even though most of the state of the art still focus on modulo channels of the form $\{2^n \pm 1\}$, given their simplicity, the use of moduli sets with modulo $\{2^n \pm k\}$ channels, with unrestricted k values, can be rather useful in the definition of RNS with larger moduli sets. With this, arithmetic systems with better performances can be obtained, given that the operands in each channel require a smaller number of bits.

The following presents the formulation and resulting structures used in the arithmetic channels performing the required addition, subtraction, and multiplication, with and without accumulation, for modulo $\{2^n - k\}$ and $\{2^n + k\}$.

2.4.2.1 Modulo $\{2^n - k\}$

The addition modulo $\{2^n - k\}$ of two residue values, can be easily formulated as:

$$\langle a + b \rangle_{2^n - k} = \begin{cases} a + b & , \ a + b < 2^n - k \\ a + b - (2^n - k) & , \ a + b \geq 2^n - k \end{cases} . \tag{2.4}$$

For the subtraction of two residue values:

$$\begin{aligned} \left\langle -b_{[n-1:0]} \right\rangle_{2^n - k} &= \left\langle 2^n - 1 - b_{[n-1:0]} + 1 - k \right\rangle_{2^n - k} \\ &= \left\langle \overline{b_{[n-1:0]}} + 1 - k \right\rangle_{2^n - k} . \end{aligned} \tag{2.5}$$

Therefore:

$$\begin{aligned} \langle a - b \rangle_{2^n - k} &= \langle a - b \rangle_{2^n - k} \\ &= \left\langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 - k \right\rangle_{2^n - k} . \end{aligned} \tag{2.6}$$

The subtraction between the residue a and b with accumulation can be formulated as:

$$\begin{aligned} \mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q + a - b \right\rangle_{2^n - k} \\ &= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 - k \right\rangle_{2^n - k} . \end{aligned} \tag{2.7}$$

Identically, the subtraction of $(a + b)$ from the accumulated value is given by:

$$\begin{aligned} \mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q - (a + b) \right\rangle_{2^n - k} \\ &= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + 2 \cdot (1 - k) \right\rangle_{2^n - k} . \end{aligned} \tag{2.8}$$

The multiplication of the residue a by b, with positive accumulation, can be formulated as:

$$\begin{aligned}
\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q + a \times b \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + p_{[2n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + 2^n \cdot p_{[2n-1:n]} + p_{[n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + (2^n - k + k) \cdot p_{[2n-1:n]} + p_{[n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + k \cdot p_{[2n-1:n]} + p_{[n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + m^1_{[n+w_k-1:0]} + p_{[n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + k \cdot m^1_{[n+w_k-1:n]} + m^1_{[n-1:0]} + p_{[n-1:0]} \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + m^2_{[2w_k-1:0]} + m^1_{[n-1:0]} + p_{[n-1:0]} \right\rangle_{2^n-k} .
\end{aligned} \tag{2.9}$$

Note that the width of k is represented by $w_k = \lceil \log_2(k) \rceil \leq n/2$ bit, in order to use the arithmetic units proposed in [17, 18], and the value $p_{[2n-1:0]}$ results from the binary multiplication of $a \times b$, that can be computed by an $n \times n$-bit binary multiplier. Consequently, the $m^1_{[n+w_k-1:0]}$ represents the result of the multiplication of the k constant by $p_{[2n-1:n]}$, and similarly the $m^2_{[2w_k-1:0]}$ represents the constant multiplication of the constant k by $m^1_{[n+w_k-1:n]}$.

The same multiplication but with negative accumulation can be obtained by computing:

$$\begin{aligned}
\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q - a \times b \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} - \left(m^2_{[2w_k-1:0]} + m^1_{[n-1:0]} + p_{[n-1:0]} \right) \right\rangle_{2^n-k} \\
&= \left\langle \mathrm{acc}_{q[n-1:0]} + \overline{m^2_{[2w_k-1:0]}} + \overline{m^1_{[n-1:0]}} + \overline{p_{[n-1:0]}} + 3 \cdot (1-k) \right\rangle_{2^n-k} .
\end{aligned} \tag{2.10}$$

2.4.2.2 Modulo $\{2^n + k\}$

Similarly to modulo $\{2^n - k\}$, the addition modulo $\{2^n + k\}$ of two residue values can be formulated as:

$$\langle a + b \rangle_{2^n+k} = \begin{cases} a + b & , a + b < 2^n + k \\ a + b - (2^n + k) & , a + b \geq 2^n + k \end{cases} . \tag{2.11}$$

Once more to formulate the subtraction of two residue values, let us start by deriving the symmetric of a residue as:

$$\begin{aligned}\left\langle -b_{[n:0]}\right\rangle_{2^n+k} &= \left\langle -2^n \cdot b_{[n]} - b_{[n-1:0]}\right\rangle_{2^n+k} \\ &= \left\langle -(2^n + k - k) \cdot b_{[n]} + 2^n - 1 - b_{[n-1:0]} + 1 + k\right\rangle_{2^n+k} \\ &= \left\langle k \cdot b_{[n]} + \overline{b_{[n-1:0]}} + 1 + k\right\rangle_{2^n+k} \ . \end{aligned} \tag{2.12}$$

Thus, the subtraction between the residue a and b can be described as:

$$\begin{aligned}\langle a - b\rangle_{2^n+k} &= \langle a - b\rangle_{2^n+k} \\ &= \left\langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 + k - k \cdot a_{[n]} + k \cdot b_{[n]}\right\rangle_{2^n+k} \\ &= \left\langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 + k \cdot (1 - a_{[n]} + b_{[n]})\right\rangle_{2^n+k} \ . \end{aligned} \tag{2.13}$$

The subtraction operations, with accumulation, for channels modulo $\{2^n + k\}$ can be formulated as:

$$\begin{aligned}\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q + a - b\right\rangle_{2^n+k} \\ &= \left\langle \mathrm{acc}_{q_{[n:0]}} + a_{[n:0]} - b_{[n:0]}\right\rangle_{2^n+k} \\ &= \left\langle 2^n \cdot \left(\mathrm{acc}_{q_{[n]}} + a_{[n]} - b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} - b_{[n-1:0]}\right\rangle_{2^n+k} \\ &= \left\langle (2^n + k - k) \cdot \left(\mathrm{acc}_{q_{[n]}} + a_{[n]} - b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \right. \\ &\quad \left. + \ 1 + k\right\rangle_{2^n+k} \\ &= \left\langle (-k) \cdot \left(\mathrm{acc}_{q_{[n]}} + a_{[n]} - b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \right. \\ &\quad \left. + \ 1 + k\right\rangle_{2^n+k} \\ &= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + k \cdot \left(-\mathrm{acc}_{q_{[n]}} - a_{[n]} + b_{[n]}\right) + \right. \\ &\quad \left. + \ k + 1\right\rangle_{2^n+k} \\ &= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \right. \\ &\quad \left. + \ \left(k \cdot \left(1 + b_{[n]} - \mathrm{acc}_{q_{[n]}} - a_{[n]}\right) + 1\right)\right\rangle_{2^n+k} \ . \end{aligned} \tag{2.14}$$

Note that the last term is one of the eight possible constants dependent on $a_{[n]}$,$b_{[n]}$, and $\mathrm{acc}_{q_{[n]}}$. These eight values can be pre-calculated.

Using the same approach, the subtraction of $(a + b)$ with accumulation can be derived as:

$$\begin{aligned}\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q - (a + b)\right\rangle_{2^n+k} \\ &= \left\langle \mathrm{acc}_{q_{[n:0]}} - a_{[n:0]} - b_{[n:0]}\right\rangle_{2^n+k} \end{aligned}$$

$$
\begin{aligned}
&= \left\langle 2^n \cdot \left(\mathrm{acc}_{q_{[n]}} - a_{[n]} - b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} - a_{[n-1:0]} - b_{[n-1:0]} \right\rangle_{2^n+k} \\
&= \left\langle -k \cdot \left(\mathrm{acc}_{q_{[n]}} - a_{[n]} - b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} + \overline{a_{[n-1:0]}} + 1 + k + \overline{b_{[n-1:0]}} + \right. \\
&\quad \left. + \ 1 + k \right\rangle_{2^n+k} \\
&= \left\langle k \cdot \left(-\mathrm{acc}_{q_{[n]}} + a_{[n]} + b_{[n]}\right) + \mathrm{acc}_{q_{[n-1:0]}} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + \right. \\
&\quad \left. + \ 2 + 2k \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + \right. \\
&\quad \left. + \ \left(2 + k \cdot \left(2 + b_{[n]} - \mathrm{acc}_{q_{[n]}} + a_{[n]}\right)\right) \right\rangle_{2^n+k} \ . \qquad (2.15)
\end{aligned}
$$

In a similar manner to the multiplication modulo $\{2^n - k\}$ depicted in (2.9), the multiplication of a by b, with positive accumulation modulo $\{2^n + k\}$, can be formulated as:

$$
\begin{aligned}
\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q + a \times b \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n:0]}} + a_{[n:0]} \times b_{[n:0]} \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n:0]}} + p_{[2n+1:0]} \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n:0]}} + 2^{2n} \cdot p_{[2n+1:2n]} + 2^n \cdot p_{[2n-1:n]} + p_{[n-1:0]} \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n:0]}} + k^2 \cdot p_{[2n+1:2n]} - k \cdot p_{[2n-1:n]} + p_{[n-1:0]} \right\rangle_{2^n+k} \\
&= \left\langle 2^n \cdot \mathrm{acc}_{q_{[n]}} + \mathrm{acc}_{q_{[n-1:0]}} + k^2 \cdot p_{[2n+1:2n]} + k \cdot \overline{m^1_{[n+w_k-1:n]}} + m^1_{[n-1:0]} + \right. \\
&\quad \left. + \ p_{[n-1:0]} + (k+1) \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + m^2_{[2w_k-1:0]} + m^1_{[n-1:0]} + p_{[n-1:0]} + k^2 \cdot p_{[2n+1:2n]} + \right. \\
&\quad \left. + \ (k+1) - k \cdot \mathrm{acc}_{q_{[n]}} \right\rangle_{2^n+k} \ . \qquad (2.16)
\end{aligned}
$$

Using the subtractive inverse to rewrite (2.16) with negative accumulation results in:

$$
\begin{aligned}
\mathrm{acc}_{q+1} &= \left\langle \mathrm{acc}_q - a \times b \right\rangle_{2^n+k} \\
&= \left\langle \mathrm{acc}_{q_{[n-1:0]}} + \overline{m^2_{[2w_k-1:0]}} + \overline{m^1_{[n-1:0]}} + \right. \\
&\quad \left. + \ \overline{p_{[n-1:0]}} + (k+1) - k^2 \cdot p_{[2n+1:2n]} - k \cdot \mathrm{acc}_{q_{[n]}} \right\rangle_{2^n+k} \ . \qquad (2.17)
\end{aligned}
$$

2.4.2.3 Proposed Arithmetic Structures

Given the above formulation, it is possible to derive a single structure capable of computing all these operations. This structure uses a binary multiplier to compute $a_i \times b_i$, two constant multipliers [for computing $(k \cdot p_{[2n-1:n]})$ and $(k \cdot m^1_{[n+w_k-1:n]})$], and one 5:1 modular adder (for computing the addition of five input vectors to one output vector) to add all the resulting terms. This 5:1 modular adder can be implemented using one 3:2 modular compression and one 4:1 modular adder, proposed in [17]. The arithmetic structure for channels modulo $\{2^n - k\}$ and $\{2^n + k\}$ are similar. However, for channels modulo $\{2^n + k\}$, a $((n+1) \times (n+1))$-bit binary multiplier is used, instead of a $(n \times n)$-bit multiplier. For the $\{2^n + k\}$ modulo, two constant multipliers are also used, but the constant block depends on more selection bits ($\text{acc}_{i_{[n]}}$, $a_{i_{[n]}}$, $b_{i_{[n]}}$, and $p_{[2n+1:2n]}$). The resulting structures are depicted in Fig. 2.3.

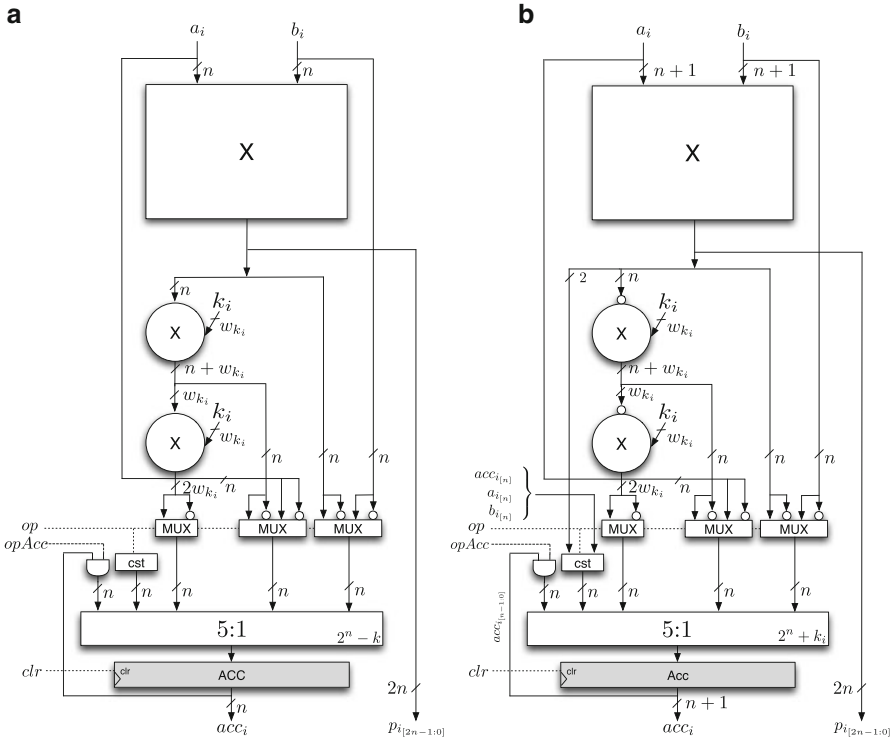

Fig. 2.3 Channel structure. (**a**) Modulo $\{2^n - k\}$. (**b**) Modulo $\{2^n + k\}$

2.4.3 RNS-to-Binary Conversion

Most reverse converters of the state of the art are based on the Chinese Remainder Theorem (CRT) [19], on the Mixed Radix Converter (MRC) [19], and on the more recent New CRT [20]. Herein, both the CRT and MRC algorithms are considered. The first approach, based on the CRT, results in a more parallel computation while the second one, based on the MRC, results in a more sequential approach but requiring simpler modular arithmetic operations. Both algorithms allow to reuse the modular arithmetic units of each channel to perform part of the conversion from RNS-to-binary. This allows to reduce the system's overall circuit area. Nevertheless, the CRT approach requires additional hardware to implement a modular adder modulo M, necessary for the computation of the final binary value. The New Chinese Remainder Theorem I (new-CRT-I) [20] and the New Chinese Remainder Theorem II (new-CRT-II) [20] are herein not considered. This is due to the fact that new-CRT-I requires modulo channels satisfying the condition $m_i > 2m_{i-1}$, that imposes unbalanced systems, and the new-CRT-II requires intermediate modular operations, which are not suitable to be implemented in the modular channels.

2.4.3.1 Proposed CRT Approach

The computation of the binary result (X) using the CRT algorithm, for N residues, can be described by [19]:

$$\begin{aligned}\langle X\rangle_M &= \left\langle \sum_{i=1}^{N} \left\langle x_i \cdot \left\langle M_i^{-1}\right\rangle_{m_i}\right\rangle_{m_i} \cdot M_i \right\rangle_M \\ &= \left\langle \sum_{i=1}^{N} x_i \cdot \left\langle M_i^{-1}\right\rangle_{m_i} \cdot M_i \right\rangle_M \\ &= \left\langle \sum_{i=1}^{N} x_i \cdot W_i \right\rangle_M ,\end{aligned} \tag{2.18}$$

where W_i is the weight of the residue m_i, given by:

$$W_i = \left\langle M_i^{-1}\right\rangle_{m_i} \cdot M_i , \tag{2.19}$$

where M_i^{-1} represents the multiplicative inverse of M_i, such that $M_i = M/m_i$ and $\left\langle M_i^{-1} M_i\right\rangle_{m_i} = 1$.

Given x_i, the n-bit remainder of the division of X by m_i, and W_i, a constant multiplication factor, with a dynamic range of $(N \cdot n)$ bits:

$$x_i = \sum_{j=0}^{n-1} x_{i[j]} 2^j , \quad W_i = \sum_{j=0}^{N\cdot n-1} W_{i[j]} 2^j , \tag{2.20}$$

the product $x_i \cdot W_i$ can be computed by decomposing the operand W_i into blocks of n bits, from most to least significant bits, where W_{i_l} represents the $[l \cdot (n+1) - 1 : l \cdot n]$ bits of W_i, given by:

$$\begin{aligned} W_i &= (2^n)^{N-1} \cdot \sum_{j=(N\cdot n)-n}^{N\cdot n-1} W_{i_{[j]}} 2^{j-(N\cdot n)+n} + \cdots + \sum_{j=0}^{n-1} W_{i_{[j]}} 2^j \\ &= 2^{(N-1)\cdot n} \cdot W_{i_{(N-1)}} + \cdots + 2^n \cdot W_{i_1} + W_{i_0} \ . \end{aligned} \tag{2.21}$$

Applying this decomposition to the modular multiplication in (2.18), the computation of X is given by:

$$\begin{aligned} \langle X \rangle_M &= \left\langle \sum_{i=1}^{N} x_i \cdot \left(2^{(N-1)\cdot n} \cdot W_{i_{(N-1)}} + \cdots + 2^n \cdot W_{i_1} + W_{i_0}\right) \right\rangle_M \\ &= \left\langle 2^{(N-1)\cdot n} \cdot \sum_{i=1}^{N} x_i \cdot W_{i_{(N-1)}} + \cdots + \sum_{i=1}^{N} x_i \cdot W_{i_0} \right\rangle_M \ . \end{aligned} \tag{2.22}$$

In the proposed RNS architecture, this computation can be performed by the multiplication units of each RNS channel, computing $x_i W_i$, followed by a final dedicated adder-tree modulo M. Note that the complexity of the final modular adder increases with the growth of the DR. To compute (2.22), N steps are required to perform all the constant multiplications. On each iteration the resulting constant multiplication values are added on a binary adder-tree, compressing the N input values, with $2n$-bit length, into one vector of $2n+e$ bits, as depicted in Fig. 2.4. Note that $e = \lceil \log_2(N) \rceil$ represents the extra bits resulting from the binary addition tree. This result is then shifted to compute the multiplication by the values 2^n to $2^{(N-1)n}$, depending on the iteration. The selection of the values to be used is performed using N 4:1 multiplexers. These multiplexers split the $2n + e$ bits vector into two groups of n bits and one of e bits, and each group is then selected by the step decode circuit, as depicted in Fig. 2.4. The shifted value is fed into the m-bit modulo M adder to compute the binary value of X. After these N steps, an additional step is required to perform the last modulo M reduction, totalling $N+1$ steps to compute the conversion of X from RNS into its binary representation, using the CRT conversion algorithm.

2.4.3.2 Proposed MRC Approach

The other considered reverse conversion approach uses the MRC algorithm [19]. This approach starts by computing a mixed-radix representation from a residue representation. Considering the moduli set $\{m_1, m_2, \cdots, m_N\}$, with N channels, and z_i the mixed-radix related to m_i $(0 \leq z_i < m_i)$, X can be computed as:

$$X = z_N \cdot m_{N-1} \cdot m_{N-2} \cdots m_1 + \cdots + z_2 \cdot m_1 + z_1 \ . \tag{2.23}$$

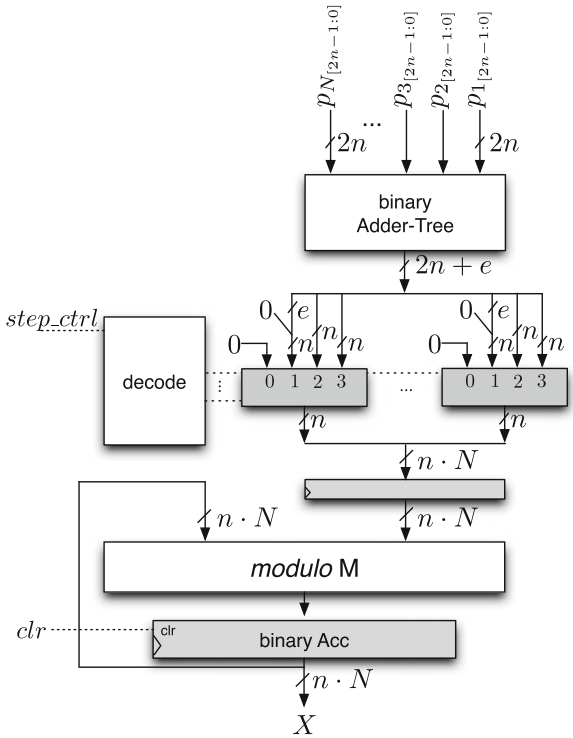

Fig. 2.4 CRT based conversion block

The mixed-radix digits (z_i) can be iteratively calculated by [19]:

$$z_i = \left\langle \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} \left(x_i - \langle z_{i-1} \cdot m_{i-2} \cdots m_1 + \cdots + z_2 \cdot m_1 + z_1 \rangle_{m_i} \right) \right\rangle_{m_i} . \tag{2.24}$$

Note that this algorithm is sequential, since the computation of z_i is dependent of z_{i-1}.

This iterative process requires $2N$ cycles to compute all the mixed-radix digits, positioning X in MRS representation. This can be performed using only the arithmetic channels of the proposed RNS structure. The needed iterations are illustrated in Table 2.2. All the multiplicative inverse values required in (2.24) can be pre-computed and stored in memory.

The final X value can be computed by multiplying the mixed-radix digits (z_i) by a constant factor (W'_i), as shown in (2.23). Considering z_i and W'_i as:

$$z_i = \sum_{j=0}^{n-1} z_{i_{[j]}} 2^j \ , \quad W'_i = \prod_{j=1}^{i-1} m_j = \sum_{j=0}^{n\cdot(i-1)} W'_{i_{[j]}} 2^j \ . \tag{2.25}$$

Table 2.2 Mixed Radix Converter algorithm execution on the proposed RNS architecture

step	channel N	···	channel 3	channel 2	channel 1
0	x_N		x_3	x_2	x_1
1	$x_N - z_1$		$x_3 - z_1$	$x_2 - z_1$	z_1
2				$acc_2 \cdot \langle m_1^{-1} \rangle_{m_2}$	
3	$acc_N - z_2 \cdot m_1$		$acc_3 - z_2 \cdot m_1$	z_2	
4			$acc_3 \cdot \langle m_2 m_1^{-1} \rangle_{m_2}$		
5	$acc_N - z_3 \cdot m_2 m_1$		z_3		
···					
2N-1	$acc_N - z_{N-1} \cdot m_{N-2} \cdots m_1$				
2N	$acc_N \cdot \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N}$				
2N	z_N		z_3	z_2	z_1
2N+1	$z_N \cdot m_N \cdots m_2 m_{1\,[n-1:0]}$		$z_3 \cdot m_2 m_{1\,[n-1:0]}$	$z_2 \cdot m_1$	z_1
2N+2	$z_N \cdot m_N \cdots m_2 m_{1\,[2n-1:n]}$		$z_3 \cdot m_2 m_{1\,[2n-1:n]}$		
···					
3N-1	$z_N \cdot m_N \cdots m_2 m_{1\,[N \cdot n-1:(N-1) \cdot n]}$				

The multiplication of a digit z_i by its weight W'_i can be computed identically to CRT (2.22), consequently X can be computed as:

$$X = 2^{(N-1) \cdot n} \cdot \sum_{i=1}^{N} z_i \cdot W'_{i_{(N-1)}} + \cdots + \sum_{i=1}^{N} z_i \cdot W'_{i_0} \ . \tag{2.26}$$

The final value X can thus be computed by a binary adder-tree, compressing the $N + 1$ input values, with $2n$ bits length, into one vector of $2n + t$ bits, with $t = \lceil log_2(N + 1) \rceil$ representing the resulting additional bits. The binary adder-tree is fed with the results given by the binary multiplier output $p_{i_{[2n-1:0]}}$ of each arithmetic channel, as depicted in Figs. 2.3 and 2.5.

In each cycle, the n Least Significant Bits (LSB) of the binary adder-tree are stored into a register, and the previous value is shifted into another register. The computation of (2.26) requires N additional steps to conclude the computation of X. With this approach, the complete conversion from RNS-to-binary requires a total of $3N$ cycles to compute X. The scheduling of operations used to perform the MRC conversion, based on the proposed RNS architecture, is depicted in Table 2.2.

The hardware overhead of this solution (Fig. 2.5) is reduced when compared with the CRT algorithm, which requires a final modulo M adder (Fig. 2.4). However, to compute X a total of $3N$ cycles are required. Nevertheless, this computational

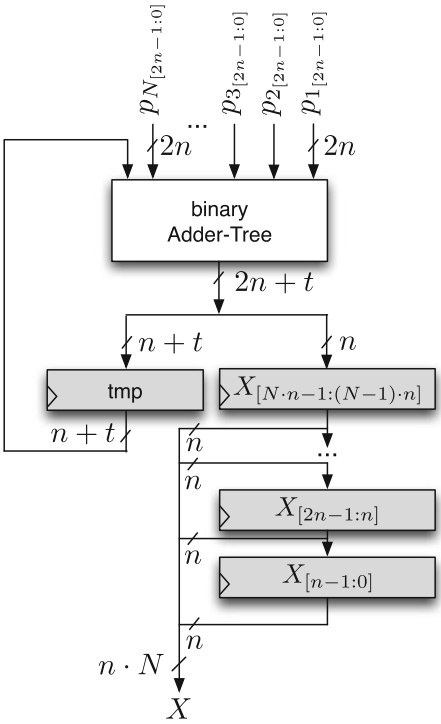

Fig. 2.5 MRC based conversion block

cost can be optimized, if the last modular multiplication in the mixed-radix-digit computation is performed on each iteration, as given by:

$$z_i = \left\langle x_i \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} - z_{i-1} \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \cdot m_{i-2} \cdots m_1 \right\rangle_{m_i} - \cdots - \right.$$
$$\left. - z_2 \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \cdot m_1 \right\rangle_{m_i} - z_1 \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} \right\rangle_{m_i} . \tag{2.27}$$

With this optimization the iterative process requires only N cycles to compute the z_i mixed-radix digits, as depicted in Table 2.3.

The final computation of X is performed using the same approach as in the non-optimized conversion computation, and thus performing the RNS-to-binary conversion in $2N$ cycles.

The following section describes the control unit for the proposed scalable RNS processor.

Table 2.3 Optimized Mixed Radix Converter algorithm execution on the proposed RNS architecture

step	channel N	···	channel 2	channel 1
0	x_N		x_2	x_1
1	$x_N \cdot \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N}$		$x_2 \cdot \langle m_1^{-1} \rangle_{m_2}$	$x_1 \cdot 1$
2	$acc - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot z_1$		$acc_2 - \langle m_1^{-1} \rangle_{m_2} \cdot z_1$	z_1
3	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_1 \cdot z_2$		z_2	
4	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_2 m_1 \cdot z_3$			
···				
N	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_{N-2} \cdots m_1 \cdot z_{N-1}$			
N	z_N		z_2	z_1
N+1	$z_N \cdot m_N \cdots m_2 m_{1\,[n-1:0]}$		$z_2 \cdot m_1$	z_1
N+2	$z_N \cdot m_N \cdots m_2 m_{1\,[2n-1:n]}$			
···				
2N-1	$z_N \cdot m_N \cdots m_2 m_{1\,[N \cdot n-1:(N-1) \cdot n]}$			

2.5 Control Units

In the RNS processor herein described, two types of control units are used: one for processor control denominated global control and another distributed control block (*CTR*, in Fig. 2.1) used in each arithmetic channel. The global control unit manages the fetching of code and the decoding of the respective instructions, alongside the management of the transfers to and from the data buffers. The single cycle arithmetic instructions perform the computation using data from registers and store the result into a destination register, without accessing the data bus, nor the constant memory. The multi-cycle instructions are controlled by a step counter, incremented at each iteration of the operation. When this counter achieves the number of iterations required to complete a given instruction (which depends on the number of channels in the RNS processor) the multi-cycle instruction is concluded, and another instruction can be fetched from the code buffer. The multi-cycle instructions are implemented by arithmetic operations partially supported by the arithmetic channels. The definition of these multi-cycle instructions is set by the micro-code to be executed. This micro-code is stored on the dedicated micro-code memory.

Table 2.4 depicts the operations performed on the arithmetic channels for the multi-cycle instructions, namely: (1) binary-to-RNS (mnemonic *bToRNS*); (2) MRS-to-RNS (*mToRNS*), used to convert from a mixed-radix representation to the RNS processor moduli set; (3) RNS-to-MRS (*rnsToM*); and (4) MRS-to-binary (*mToBIN*). Since only three types of arithmetic operations are required, and they only differ in the first steps of each instruction, the needed micro-code can be

Table 2.4 Condensed micro-code for multi-cycle operations

Mnemonic	Operation step	Instructions
bToRNS rd,#,#	0	mula rd,#,#
mToRNS rd,#,#	$\neq 0$	amula rd,#,#
rnsToM rd,rs1,#	0	mula rd,rs1,#
mToBIN #,rs1,#	$\neq 0$	smula rd,rs1,#

reduced to a relative small memory, as described in Table 2.4. Instructions *bToRNS* and *mToRNS* actually have the exact same micro-code, differing only in the used constants. The same happens with the instructions *rnsToM* and *mToBIN*.

Alongside the arithmetic operations, used in each multi-cycle instruction, the operands source is also defined, namely: (1) the data buffer (for external data); (2) the internal shared bus; (3) the register file; and (4) the internal memory (for the constants stored in memory).

The binary-to-RNS and the MRS-to-RNS conversions instructions receive data from the data buffer and the internal memory, as source one and two, respectively. The RNS-to-MRS conversion instruction uses the internal shared bus as the source for the first operand and the internal memory as source for the second operand. The data presented in the internal bus is obtained from the arithmetic channels and defined by the operation step. This is used to propagate the mixed-radix-digits, z_i, through the other RNS channels, as depicted in Table 2.3. The last multi-cycle instruction, the MRS-to-binary computation, uses the register file and the internal memory as the operands source. Additionally, this last instruction disables the register update flag, since no register is to be updated. This conversion operation only uses the binary multipliers in the channels to compute X, as described in (2.23). Recall that the RNS-to-binary conversion is computed by two multi-cycle instructions: the first converts from RNS-to-MRS, computing the mixed-radix digits, and the second instruction converts from MRS-to-binary. The herein proposed architecture allows to implement Base Extension operations, supported by the RNS-to-MRS instruction in the source moduli set and by the MRS-to-RNS instruction in the destination moduli set.

Given the considered approach, the conversion instruction only depends on the step counter and on the stored constants for the chosen moduli set. With this, the proposed architecture becomes generic and scalable, supporting moduli sets of the form $\{2^n \pm k_i\}$. However, the number of moduli in the moduli set is limited by the fact that the moduli must be co-prime and of the restriction of $w_k = \lceil \log_2(k) \rceil \leq n/2$, given the used arithmetic units [17, 18]. As such, the larger the width of each channel, n, the higher the number of existing co-prime numbers and moduli that can co-exist.

Table 2.5 depicts the number of relative co-prime numbers that can be defined according to the allowed channel width. This table illustrates the number of existing co-prime values when considering moduli channels with only moduli of the form $\{2^n - k_i\}$ and with moduli of the form $\{2^n \pm k_i\}$.

Table 2.5 Number of relative prime numbers available for n-bit channel length

Channel width n [bit]	8	12	16	20	$\cdots$	32
$\{2^n - k\}$	6	14	43	129	$\cdots$	4715
$\{2^n \pm k\}$	11	25	83	235	$\cdots$	8677

Moduli sets composed only of $\{2^n - k_i\}$ moduli are particularly interesting, given that they allow for simpler and more compact arithmetic units when compared with modulo $\{2^n + k_i\}$ arithmetic units, as described in Sect. 2.4.2.

2.6 State-of-the-Art Analysis

In order to compare the herein described scalable and programmable RNS processor, the relevant state of the art is herein analysed, namely other programmable RNS architectures. Three other such systems are presented in the existing state of the art that satisfy this, namely: Residue Digital Signal Processor (RDSP), a general processing architecture proposed in [21]; the well-known Cox–Rower architecture proposed in [22]; and a more recent work implementing a unified computing system [23], where the code is defined as data transfers between registers and functional units. The last two architectures target the computation of asymmetric encryption algorithms, such as RSA [24] and ECC [25] algorithms.

2.6.1 State of the Art

The RDSP [21] is a 32-bit pipelined Reduced Instruction Set Computer (RISC) based processor with 5 pipeline stages. The arithmetic operations are performed by arithmetic units supporting the balanced moduli set $\{2^n - 1, 2^{2n}, 2^n + 1\}$, with $n = 8$. A particularity of this processor is that it also supports binary arithmetic operations. This processor supports the needed conversion operations, from binary-to-RNS and RNS-to-binary. For this, dedicated arithmetic units are used. The processor supports modular addition and multiplication operations, with and without accumulation. Furthermore, this processor offers flow control instructions (such as brunch instructions), feature not present in the remaining state of the art and in the processor herein described.

The main drawback of this architecture is that the supported moduli set only allows for the parallelism of three channels, providing a relatively small DR. In order to increase the DR and/or to reduce the width of the RNS channels, different RNS moduli sets could be considered for this processor, such as $\{2^n - 3, 2^n - 1, 2^n + 1, 2^n + 3\}$ [26, 27]. However, this moduli set has the disadvantage of more complex modulo arithmetic, since the modular reduction of $\{2^n \pm 3\}$ requires more area and imposes higher delay costs [28]. Considering only moduli of the form $\{2^n\}$ and

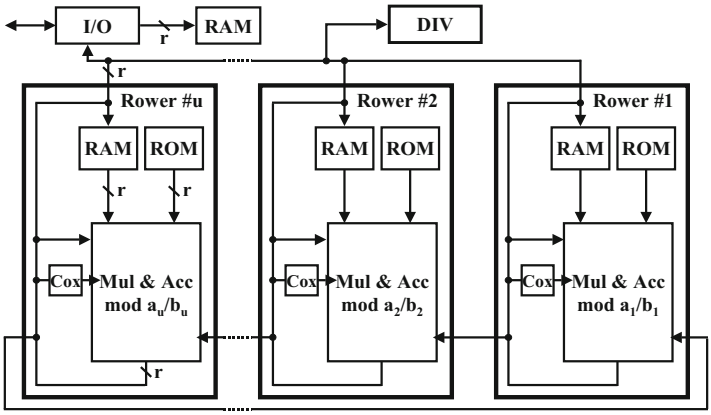

Fig. 2.6 Cox–Rower architecture, from [16]

$\{2^n \pm 1\}$, given its less complex structures, moduli sets with DRs up to $(8n)$-bit can be deployed, such as $\{2^n-1, 2^n, 2^n+1, 2^{n+1}-1\}$ [29], $\{2^n-1, 2^n, 2^n+1, 2^{2n}+1\}$ [30], $\{2^n-1, 2^n, 2^n+1, 2^{n+1}-1, 2^{n-1}-1\}$ [31], and $\{2^{n-5}-1, 2^{n-3}-1, 2^{n-3}+1, 2^{n-2}+1, 2^{n-1}-1, 2^{n-1}+1, 2^n, 2^n+1\}$ [32]. Nevertheless, the predefined datapath of the RDSP processor does not allow to easily scale to larger DR.

The initial Cox–Rower architecture [22] is composed of a single Cox unit and by several Rower units. Each Rower unit is composed of a modular multiplier-and-accumulator, computing the channels RNS arithmetic. Note that each Rower unit is a dual-moduli channel, able to perform computations in two moduli sets (base one and two for the Montgomery Multiplication (MM) [13, 14]). The Cox unit is used to compute a correction factor, required in the MM. The Cox implementation is based on a simpler adder unit, computing the correction factor with bits received each cycle from the Rower units. An optimization to the initial architecture is proposed in [16], and corresponds to embedding a Cox unit in each Rower, as depicted in Fig. 2.6.

This approach avoids the broadcast of the correction factor data. To further improve data transfers, a ring connection is used, instead of a shared bus.

The authors also proposed an improved Rower unit, considering a three stage pipeline unit, towards a higher throughput and allowing to reduce the number of required Rower units in half. This unit implements a 32 bit long modular multiplier-and-accumulator, as depict in Fig. 2.7a. In the first stage of the pipeline, the Rower unit computes the binary multiplication and addition operations, with or without accumulation. The modular reduction is computed in the second and third stages of the pipeline. This reduction unit is implemented by three dual-constant multipliers and three adders, as depicted in Fig. 2.7b. Note that the Rower units are dual-moduli ($2^r - \mu_i$ and $2^r - \nu_i$ modulo), each supporting only two moduli.

More recently, [33, 34] increased the number of pipeline stages of the Rower units, reaching higher frequencies. The pipeline was increased in four stages, implemented in the arithmetic block of the Rower unit. It also implements the

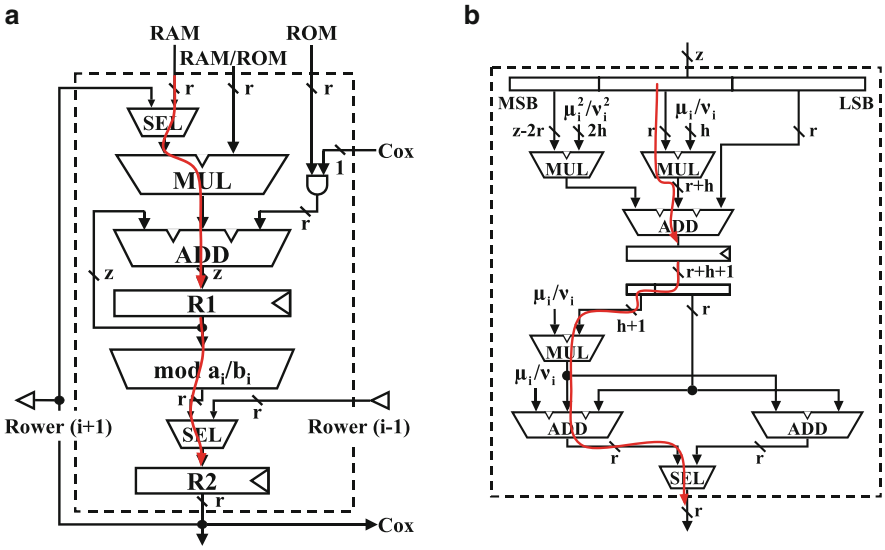

Fig. 2.7 Rower architecture, from the Cox–Rower [16]. (**a**) Rower architecture. (**b**) Modular reduction

modular computation using the leak resistant arithmetic proposed in [35]. This modular arithmetic implements a countermeasure in the Cox–Rower architecture to provide better protections against side channel attacks.

Another RNS processor has been proposed in [36], based on a Transport Triggered Architecture (TTA), computing the base extensions using the same approach as [37]. The TTA approach allows for a higher instruction level parallelism, where the code is defined as data transports between registers and functional units. Operations start as a side-effect of transporting an operand to a "triggered port" of the functional unit. The presented structure deploys eight main functional units for the RNS channel arithmetic computation, for any modulo m_i of the used base. The number of functional units is upper bounded to eight, given that more functional units increase the number of buses. Moreover, in TTA [36] more buses require wider instruction words, resulting in a larger instruction memory. The main functional units of this RNS processor [36] are implemented by the Modular Multiplication-and-Accumulation (MMAC) units, as depicted in Fig. 2.8. However, these eight MMAC usually are less than the number of modulus, reducing the exploited RNS parallelism, since only eight modulo operations can be done simultaneously. In order to overcome this limitation, the MMAC is implemented as a three stage pipeline unit, allowing to take advantage of the inherent independence between RNS channels. The critical path of the MMAC functional unit is in the second pipeline stage, responsible for computing the modular reduction, as depicted in Fig. 2.8. The critical path is given by the delay of two multipliers, with 32×14 and 15×14 bits, and three binary adders, one with 46 bit length and two with 32 bit length.

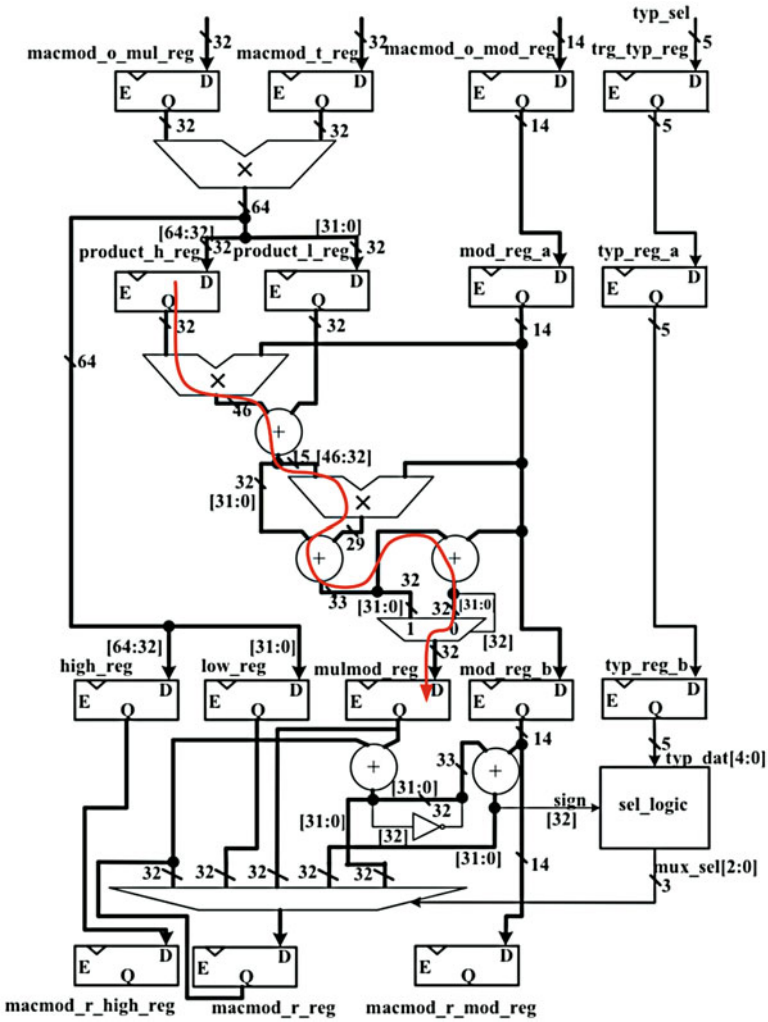

Fig. 2.8 MMAC architecture, from *uRNS* [23]

This work [36] has been further extended to a unified cryptographic processor [23], herein designated as *uRNS*, capable of computing RSA and ECC. In the *uRNS* the base extension is computed by the method proposed in [16]. The MMAC units are distributed in groups of four units each, up to 16 groups, allowing for different optimizations and performance levels, according to the key lengths. However, in the presented implementation the number of groups is restricted to 2, in order to maintain the number of buses and the instruction memory bounded, to allow it to be implemented.

2.6.2 Analysis Based on the Arithmetic Units

Given the related state of the art, only the two last processors are herein considered for a theoretical assessment, due to their support of large Dynamic Ranges. In order to compare these two last processors with the proposed RNS processor, only their modular arithmetic units are considered, since only these units are detailed in the literature [16, 23]. Note that the bus architectures details and operations scheduling are missing in the relevant related state of the art, not allowing for a full processor analysis. To obtain a technology independent assessment of the resulting arithmetic RNS architecture, an analysis is carried out using a neutral Full Adder based model [27, 38–40]. The estimation model considers that the area of 1-bit Full-Adder (FA) is represented by Δ_{FA}, and the τ_{FA} represents its delay. For the n-bit binary multipliers a delay of $2n\tau_{\mathrm{FA}}$ and an area of n^2 FA are considered [41]. For the multiplication of an n-bit operand by a w_k-bit constant, a similar binary multiplier estimation is considered, with $nw_k\Delta_{\mathrm{FA}}$ of area resources usage and $(n + w_k)\tau_{\mathrm{FA}}$ of delay. The area resources and delay cost of registers are not taken into account in this estimation model. Considering this, the assessment of the related state of the art is based on the evaluation of the estimation costs for area and delay of the channel's arithmetic structures.

The evaluation of the described RNS architecture is based on the estimation cost of the channel's arithmetic structures, described in Sect. 2.4.2. Given that the area cost estimation for the modulo $\{2^n - k\}$ channel arithmetic block is imposed by the area of one binary multiplier ($n^2\Delta_{\mathrm{FA}}$), two constant multipliers, contributing with $2nw_k\Delta_{\mathrm{FA}}$, and one 5:1 modular adder, implemented with one 4:1 modular adder and one modular Carry-Save-Adder (CSA) [41], contributing with $(8n + 2w_k)\Delta_{\mathrm{FA}}$. As illustrated in Fig. 2.3, the critical path of the resulting channel arithmetic structure is imposed by the binary multiplier, contributing with $2n\tau_{\mathrm{FA}}$, the two constant multipliers, contributing with $2(n + w_k)\tau_{\mathrm{FA}}$, and the 5:1 modular adder, imposing a delay of $(n + 5)\tau_{\mathrm{FA}}$. Note that these units do not have internal registers, since no pipeline is considered for them in the herein proposed architecture. An identical analysis can be performed for the arithmetic channel modulo $\{2^n + k\}$, resulting in the values depicted in Table 2.6.

The estimated area cost for the Rower unit of the Cox–Rower architecture, using the same FA model, is imposed by the area of the three pipeline stages. For the total area estimation cost the first pipeline stage contributes with the area of one binary multiplier ($r^2\Delta_{\mathrm{FA}}$, where r is the bit width of the moduli) and one 3:1 binary compression unit ($2z\Delta_{\mathrm{FA}}$, where z represents a bit length greater than $2r$, considered to allow accumulation operations). The second pipeline stage is implemented by two binary multiplier and one 3:1 binary compression unit, contributing to the estimation

Table 2.6 Area and delay estimation for the channel arithmetic structures

Modulo	Δ (area) [Δ_{FA}]	τ (delay) [τ_{FA}]
$2^n - k$	$n^2 + 8n + (2n + 2)w_k$	$5n + 2w_k + 5$
$2^n + k$	$(n + 1)^2 + (2n + 2)w_k + 8n$	$5n + 2w_k + 7$

cost with $(2h(z-2r)\Delta_{\mathrm{FA}})$, $((h \cdot r)\Delta_{\mathrm{FA}})$, and $(2(r+h)\Delta_{\mathrm{FA}})$, respectively (note that h represents the bit length of the μ_i and ν_i), as depicted in Fig. 2.7b. Finally, the third stage contributes with the area of one binary multiplier, one 3:1 binary compression unit, and one binary adder, which are given by $((h^2+h)\Delta_{\mathrm{FA}})$, $(2r\Delta_{\mathrm{FA}})$, and $(r\Delta_{\mathrm{FA}})$, respectively. The estimation delay cost is given by the maximum critical path, located in the first pipeline stage as depicted in Fig. 2.7 and (2.29). The total estimations for area and delay, for the Rower unit of the Cox–Rower architecture, are given by:

$$\begin{aligned}
\Delta_{\mathrm{Rower}} &= \Delta_{\text{1st pipeline stage}} + \Delta_{\text{2nd pipeline stage}} + \Delta_{\text{3rd pipeline stage}} \\
&= \left[\left(r^2+2z\right) + (2h\,(z-2r) + h \cdot r + 2\,(r+h)) + \left(h^2+h+2r+r\right)\right] \cdot \Delta_{\mathrm{FA}} \\
&= \left(r^2+h^2+2z+5r+3h+2hz-3hr\right) \cdot \Delta_{\mathrm{FA}} \qquad (2.28)
\end{aligned}$$

$$\begin{aligned}
\tau_{\mathrm{Rower}} &= \max\left(\tau_{\text{1st pipeline stage}}, \tau_{\text{2nd pipeline stage}}, \tau_{\text{3rd pipeline stage}}\right) \\
&= \max\left(2r+z+1, 2r+2h+1, 2h+2r+2\right) \cdot \tau_{\mathrm{FA}} \\
&= (2r+z+1) \cdot \tau_{\mathrm{FA}}\ . \qquad (2.29)
\end{aligned}$$

where $z = 72$, $r = 32$, and $h = 10$, for a 32 bit channels. Note that the area and delay of the registers are not taken into account in the considered model.

Similar analysis, to the Cox–Rower, is applied to the unified cryptographic processor (*uRNS*) [23], where the estimation area cost is given by: (1) one 32 bit binary multiplier, in the first pipeline stage; (2) one 32 by 14 bit binary multiplier, one 46 bit binary adder, one 15 by 14 bit binary multiplier, and two 32 bit binary adders, in the second stage; and (3) two 32 bit binary adders, in the third pipeline stage. Note that the MMAC unit proposed in *uRNS* [23] has a fixed bit width. Given this, fixed values are herein considered in the estimated cost analysis, instead of a generic form. Recall that the estimation delay cost for the *uRNS* is given by the critical path of the second pipeline stage, as depicted in Fig. 2.8. The estimation costs for the *uRNS* [23] are summarized as:

$$\begin{aligned}
\Delta_{uRNS} &= \Delta_{\text{1st pipeline stage}} + \Delta_{\text{2nd pipeline stage}} + \Delta_{\text{3rd pipeline stage}} \\
&= \left[\left(32^2\right) + (32 \cdot 14 + 46 + 15 \cdot 14 + 32 + 32) + (32+32)\right] \cdot \Delta_{\mathrm{FA}} \\
&= 1856\Delta_{\mathrm{FA}} \qquad (2.30)
\end{aligned}$$

$$\begin{aligned}
\tau_{uRNS} &= \max\left(\tau_{\text{1st pipeline stage}}, \tau_{\text{2nd pipeline stage}}, \tau_{\text{3rd pipeline stage}}\right) \\
&= \max\left(2 \cdot 32, 32+14+46+15+14+32+32, 32+32\right) \cdot \tau_{\mathrm{FA}} \\
&= \max\left(64, 185, 64\right) \cdot \tau_{\mathrm{FA}} \\
&= 185\tau_{\mathrm{FA}} \qquad (2.31)
\end{aligned}$$

Table 2.7 Theoretical analysis for the RNS arithmetic channels

Architecture	Pipeline stages	Δ (area) [Δ_{FA}]	τ (delay) [τ_{FA}]
Cox–Rower [16]	3	1938	137
uRNS [23]	3	1856	185
Proposed	0	1940	185

Table 2.7 summarizes and compares the estimated cost for these architectures, considering channels with 32 bit of length, the channel's fixed width in the *uRNS* [23] architecture. Additionally, in order to make a fair comparison, k values up to 10 bits in length ($w_{k_i} \leq 10$) are herein considered for the proposed arithmetic unit, since this is the maximum bit length allowed in the Cox–Rower architecture, namely as h in [16]. However, in *uRNS* [23] the bit length is less restricted, achieving values up to 14 bits.

From this theoretical analysis it can be concluded that the proposed RNS arithmetic units require similar area resources to those in the state of the art. Delay wise, the obtained metrics suggest that the Cox–Rower units are about 25% faster when using a three stage pipeline. However, these units are only able to compute two distinct modulus, not allowing to fully take advantage of the existing pipeline. On the other hand, the arithmetic units from the *uRNS* architecture are equally fast and also support generic modulus. However, this delay is achieved by using a three stage pipeline, while the arithmetic unit herein presented achieves identical metrics without pipeline. Furthermore, besides also allowing the usage of pipeline the herein considered RNS arithmetic units implement several operations (such as modular addition, subtraction, multiplication, and accumulation), unlike the one in the related state of the art that supports only one single operation. These features facilitate the implementation of more complex operations and a higher level of programmability.

Architecture wise, the Cox–Rower and *uRNS* architectures are very oriented to the computation of asymmetric cryptographic algorithms lacking adaptability to the computation of other algorithms. Data transfer wise, the optimized Cox–Rower architecture considers a ring bus, minimizing the impact of the bus in the system but significantly restricting the possible data transfers. On the other hand, the *uRNS* architecture presents a very powerful bus network, allowing the wide data transfer between units, needed for the TTA approach. The consequence of this highly interconnected bus is the lack of scalability, only supporting up to 8 arithmetic units, without causing significant data transfer delays.

The RNS architecture herein considered presents a trade-off in terms of data transfer and performance, allowing for a versatile data transfer and scalability. Moreover, the processor herein presented is fully programmable architecture, allowing it to be used in the computation of a wide range of algorithms, thus offering a programmable computing system supported by RNS.

2.7 Summary

The work described in this chapter presents the first version of a flexible and adaptable comprehensive framework to automatically design RNS processors. This RNS architecture can also be used to promote the development of novel moduli sets and optimized reverse converters.

Herein a compact and scalable RNS architecture is described, allowing for the design of RNS-based processors for any co-prime moduli set using moduli of the form $\{2^n \pm k_i\}$. This processor allows for modular additions, subtractions, and multiplication operations, with or without accumulation, supported by the proposed unified arithmetic units, while also providing the needed forward, reverse, and base extension conversions. With the proposed architecture the delay and area cost of dedicated processors are reduced while further exploring the parallelism and carry-free characteristic of RNS.

In conclusion, the proposed RNS processor allows for generic, scalable, and compact implementations with competitive performances when compared with the commonly used binary based systems and the related RNS state of the art, showing that efficient programmable RNS architectures can be designed.

References

1. S.R. Barraclough, M. Sotheran, K. Burgin, A.P. Wise, A. Vadher, W.P. Robbins, R.M. Forsyth, The design and implementation of the IMS A110 image and signal processor, in *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1989, pp. 24.5/1–24.5/4
2. W.A. Chren, RNS-based enhancements for direct digital frequency synthesis. IEEE Trans. Circuits Syst. II Analog Digit. Signal Process. **42**(8), 516–524 (1995)
3. F. Piazza, E. Di Claudio, G. Orlandi, Fast combinatorial RNS processor for DSP applications. IEEE Trans. Comput. **44**(5), 624–633 (1995)
4. T.J. Slegel, R.J. Veracca, Design and performance of the IBM enterprise system/9000 type 9121 vector facility. IBM J. Res. Dev. **35**, 367–381 (1991)
5. C.-L. Wang, New bit serial VLSI implementation of RNS FIR digital filters. IEEE Trans. Circuits Syst. II Exp. Briefs **41**(11), 768–772 (1994)
6. P.M. Matutino, L. Sousa, An RNS based specific processor for computing the minimum sum-of-absolute-differences, in *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, September 2008, pp. 768–775
7. J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, T. Wedi, Video coding with H.264/AVC: tools, performance, and complexity. IEEE Circuits Syst. Mag. **4**(1), 7–28 (first 2004)
8. J. Bajard, L. Imbert, A full RNS implementation of RSA. IEEE Trans. Comput. **53**(6), 769–774 (2004)
9. Antão, S., Bajard, J.-C., Sousa, L.: RNS based elliptic curve point multiplication for massive parallel architectures. Comput. J. 2011 - Oxf. J. **55**(5), 629–647 (2011)
10. G.C. Cardarilli, A. Nannarelli, M. Re, Residue number system for low-power DSP applications, in *Asilomar Conference on Signals, Systems and Computers - ACSSC 2007* (2007), pp. 1412–1416
11. P. Garai, C.B. Dutta, RNS based reconfigurable processor for high speed signal processing, in *IEEE Conference TENCON 2014*, October 2014, pp. 1–6

12. P.L. Montgomery, Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
13. J. Bajard, L.-S. Didier, P. Kornerup, An RNS montgomery modular multiplication algorithm, in *13th IEEE Symposium on Computer Arithmetic (ARITH)*, July 1997, pp. 234–239
14. K.C. Posch, R. Posch, Modulo reduction in residue number systems. IEEE Trans. Parallel Distrib. Syst. **6**(5), 449–454 (1995)
15. Advanced RISC Machines Ltd (ARM), *ARM7 Data sheet*. Advanced RISC Machines Ltd (ARM), ARM DDi 0020C edition, December 1994
16. H. Nozaki, M. Motoyama, A. Shimbo, S. Kawamura, Implementation of RSA algorithm based on RNS montgomery multiplication, in *Cryptographic Hardware and Embedded Systems - CHES 2001*, ed. by Ç. Koç, D. Naccache, C. Paar. Lecture Notes in Computer Science, vol. 2162 (Springer, Berlin, Heidelberg, 2001), pp. 364–376
17. P. Miguens Matutino, R. Chaves, L. Sousa, An efficient scalable RNS architecture for large dynamic ranges. J. Signal Process. Syst. **77**(1–2), 191–205 (2014)
18. P.M. Matutino, R. Chaves, L. Sousa, Arithmetic-based Binary-to-RNS converter modulo $\{2^n \pm k\}$ for jn-bit dynamic range. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **23**(3), 603–607 (2015)
19. A. Omondi, B. Premkumar (eds.), *Residue Number Systems: Theory and Implementation* (Imperial College Press, London, 2007)
20. Y. Wang, Residue-to-binary converters based on new Chinese remainder theorems. IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process. **47**(3), 197–205 (2000)
21. R. Chaves, L. Sousa, RDSP: A RISC DSP based on Residue Number System, in *Proceedings of the Euromicro Symposium on Digital System Design*, September 2003, pp. 128–135, ed. by IEEE
22. S. Kawamura, M. Koike, F. Sano, A. Shimbo, Cox-Rower architecture for fast parallel montgomery multiplication, in *Advances in Cryptology - EUROCRYPT 2000*, ed. by B. Preneel. Lecture Notes in Computer Science, vol. 1807 (Springer Berlin, Heidelberg, 2000), pp. 523–538
23. J. Wei, W. Guo, H. Liu, Y. Tan, A unified cryptographic processor for RSA and ECC in RNS, in *Computer Engineering and Technology*, ed. by W. Xu, L. Xiao, C. Zhang, J. Li, L. Yu. Communications in Computer and Information Science, vol. 396 (Springer, Berlin, Heidelberg, 2013), pp. 19–32
24. R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **26**(1), 96–99 (1983)
25. R. Schroeppel, H. Orman, S. O'Malley, O. Spatscheck, Fast key exchange with elliptic curve systems, in *Advances in Cryptology – Crypto ´95*. Lecture Notes in Computer Science, vol. 963 (Springer, Berlin, 1995), pp. 43–56
26. P.V. Ananda Mohan, Reverse converters for the moduli sets $\{2^{2N}-1, 2^N, 2^{2N}+1\}$ and $\{2^N-3, 2^N+1, 2^N-1, 2^N+3\}$, in *SPCOM '04*, December 2004, pp. 188–192
27. M.-H. Sheu, S.-H. Lin, C. Chen, S.-W. Yang, An efficient VLSI design for a residue to binary converter for general balance moduli $\{2^n-3, 2^n+1, 2^n-1, 2^n+3\}$. IEEE Trans. Circuits Syst. II: Express Briefs **51**(3), 152–155 (2004)
28. P. Miguens Matutino, R. Chaves, L. Sousa, Arithmetic units for RNS moduli $\{2^n-3\}$ and $\{2^n+3\}$ operations, in *13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, September 2010, pp. 243–246
29. A.B. Premkumar, A.P. Vinod, A memoryless reverse converter for the 4-moduli superset $\{2^n-1, 2^n, 2^n+1, 2^{n+1}-1\}$. J. Circuits Syst. Comput. **10**(01n02), 85–99 (2000)
30. B. Cao, C.-H. Chang, T. Srikanthan, An efficient reverse converter for the 4-moduli set $\{2^n-1, 2^n, 2^n+1, 2^{2n}+1\}$ based on the new Chinese Remainder Theorem. IEEE Trans. Circuits Syst. I Fundam. Theory Appl. **50**(10), 1296–1303 (2003)
31. B. Cao, C.-H. Chang, T. Srikanthan, A residue-to-binary converter for a new five-moduli set. IEEE Trans. Circuits Syst. I: Regul. Pap. **54**(5), 1041–1049 (2007)
32. A. Skavantzos, M. Abdallah, T. Stouraitis, D. Schinianakis, Design of a balanced 8-modulus RNS, in *16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2009*, December 2009, pp. 61–64

33. N. Guillermin, A high speed coprocessor for elliptic curve scalar multiplications over Fp, in *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems, CHES'10* (Springer-Verlag, Berlin, Heidelberg, 2010) pp. 48–64
34. N. Guillermin, A coprocessor for secure and high speed modular arithmetic. Cryptology ePrint Archive, Report 2011/354, July 2011
35. J.-C. Bajard, L. Imbert, P.-Y. Liardet, Y. Teglia, Leak resistant arithmetic, in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ed. by M. Joye, J.-J. Quisquater. Lecture Notes in Computer Science, vol. 3156 (Springer, Berlin, Heidelberg, 2004), pp. 62–75
36. W. Guo, Y. Liu, S. Bai, J. Wei, D. Sun, Hardware architecture for rsa cryptography based on residue number system. Trans. Tianjin Univ. **18**(4), 237–242 (2012)
37. J.-C. Bajard, L.-S. Didier, P. Kornerup, Modular multiplication and base extensions in residue number systems, in *IEEE 15TH Symposium on Computer Arithmetic* (IEEE, New York, 2001), pp. 59–65
38. P.V. Ananda Mohan, New reverse converters for the moduli set $\{2^n-3, 2^n-1, 2^n+1, 2^n+3\}$. AEU - Int. J. Electron. Commun. **62**(9), 643–658 (2008)
39. H. Pettenghi, R. Chaves, L. Sousa, RNS reverse converters for moduli sets with dynamic ranges up to (8n+1)-bit. IEEE Trans. Circuits Syst. I Regul. Pap. **PP**(99), 1–14 (2012)
40. G. Jaberipur, H. Ahmadifar, A rom-less reverse RNS converter for moduli set $\{2^q \pm 1, 2^q \pm 3\}$. IET Comput. Digit. Tech. **8**(1), 11–22 (2014)
41. P.M. Matutino, H. Pettenghi, R. Chaves, L. Sousa, RNS arithmetic units for modulo $\{2^n \pm k\}$, in *15th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, September 2012, pp. 795–802

Chapter 3
Non-Modular Operations of the Residue Number System: Functions for Computing

Giuseppe Pirlo

Nomenclature

- $m_1, m_2, \ldots, m_n$: set of pairwise relatively prime moduli (n integer, $n \geq 2$)
- $M = \prod_{i=1}^{n} m_i$
- $I = [0, M-1]$: dynamic range of the RNS (set of integers)
- $X_i = |X|_{m_i}$: the least positive residue of X modulo m_i, $i = 1,2,..,n$
- $M_i = \frac{M}{m_i}$, $i = 1,2,\ldots,n$
- $SQ = \sum_{i=1}^{n} M_i$: the 'diagonal modulus' of the RNS
- $J_{a,b} = \left|\frac{1}{a}\right|_b$: the multiplicative inverse of a modulo b (i.e. $|a \cdot \left|\frac{1}{a}\right|_b|_b = 1$)
- $\lfloor a \rfloor$: the largest integer not exceeding a
- $\lceil b \rceil$: rounding to a higher integer
- $\equiv$: modular congruence
- MOMA (n, W): multi-operand modulo W adder for n operands.

3.1 Introduction

In the Residue Number System (RNS) based on the set of pairwise relatively prime moduli $m_1, m_2, \ldots, m_n$, each integer $X \in I$, $I = [0, M-1]$, $M = m_1 \times m_2 \times \ldots \times m_n$, is uniquely represented by the n-tuple of residue digits as $X \rightarrow$ (RNS) $(x_1, x_2, .., x_n)$

G. Pirlo (✉)
Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro,
Via Orabona, Bari 70125 Italy
e-mail: giuseppe.pirlo@uniba.it

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_3

[1], where $X_i = |X|_{m_i}$ is the residue of X modulo m_i, $i = 1,2,..,n$. In RNS, additive and multiplicative operations can be performed in a carry-free mode over each modulus. In other words let be $X \rightarrow$ (RNS) $(x_1,x_2,..,x_n)$ and $Y \rightarrow$ (RNS) $(y_1,y_2,..,y_n)$ two integers in residue representation, it results $X \circ Y \rightarrow$ (RNS) $(|x_1 \circ y_1|_{m_1}, |x_2 \circ y_2|_{m_2}, .., |x_n \circ y_n|_{m_n})$ where $\circ$ is an operation of addition or multiplication [1].

Unfortunately, difficulty in computing non-modular operations, like residue-to-binary conversion and magnitude comparison, prevented RNS from receiving a major influence in the progress of general purpose computers [2]: Residue-to-binary conversion is necessary for the use of RNS arithmetic units into general purpose computers, which are based on the binary number system. Magnitude comparison supports other logic operations (like overflow detection) that are complex in the RNS due to the difficulty in defining an order relation on quotient sets [1, 2].

Traditional implementations of non-modular operations in the RNS are based on the Mixed-Radix Conversion (MRC), that follows a strictly sequential process, or the Chinese Remainder Theorem (CRT), that provides a parallel conversion formula but requires dedicated extra-hardware. In particular, the direct implementation of the CRT is based on a modulo M operation of n operands [1, 3]. Therefore, several approaches have been proposed to perform the CRT more efficiently by imposing severe constraints on the set of moduli that must be of the type $\{2^k - 1, 2^k, 2^k + 1\}$ [4, 5], $\{2^k, 2^k - 1, 2^{k-1} - 1\}$ [6], $\{2^{k-1}, 2^k, 2^k + 1, 2^{k+1} - 1, 2^{k+1} - 1\}$ [7, 8] (k integer), $\{2^{k(1)} - 1, 2^{k(1)} + 1, 2^{k(2)} - 1, 2^{k(2)} + 1, \ldots, 2^{k(L)} - 1, 2^{k(L)} + 1\}$ ($k(j)$ integer for $j = 1,2,\ldots,L$) [9, 10], $\{2^m - 1, 2^{20m} + 1, 2^{21m} + 1, 2^{22m} + 1, \ldots, 2^{2km} + 1\}$ (k and m integers) [11]. Other approaches exploit the potential of recent Multi-Operand Modular Adders (MOMA) for performing effectively modulo M operations [12–14].

Another class of approaches to perform magnitude comparison in the RNS is based on the 'core functions' defined from the RNS to the integer [15]. Unfortunately the 'core functions', that were first proposed by Akushskii et al [16], use an iterative process of descend and lift when 'critical core' values are found. Miller et al. [17] presented an improved version of the 'core functions', able to skips the 'core value' conditions at the cost of using only small non-critical regions of the dynamic range.

Recently, new functions have been proposed, defined from the RNS to the integers. The 'diagonal function' of the RNS proposed by Pirlo et al. [18] supports the implementation of non-modular operations without reducing the dynamic range of the RNS in non-critical regions. Moreover the computation of the 'diagonal function' requires an extra 'diagonal modulus', named SQ, that is usually smaller than the modulus M required by the CRT [2]. The 'diagonal function' exploits the observation that the integers in residue representation dispose themselves on diagonals when they are arranged in the multi-dimensional discrete space associated with the RNS [18, 19]. Unfortunately, although the 'diagonal function' is a powerful tool for magnitude comparison, it does not support effectively residue-to-binary conversion [18, 20]. More recently, the 'quotient functions' of the RNS was defined

by Pirlo et al. [20]. The 'quotient function' can support effectively—using the same circuit—both magnitude comparison and residue-to-binary conversion [21].

This chapter presents an overview of the monotone functions for the implementation of non-modular operations in the RNS. The organization of the chapter is as follows: Sect. 3.2 presents the traditional approaches for performing non-modular operations, based on the Mixed-Radix Conversion and the Chinese Remainder Theorem. Section 3.3 presents the 'diagonal function' and describes some properties useful to implement magnitude comparison in the RNS. Section 3.4 presents the 'quotient function' of the RNS and describes how the 'quotient function' supports magnitude comparison and residue-to-binary conversion. A comparative analysis of the diverse implementations of non-modular operations in the RNS is discussed in Sect. 3.5. The conclusion of the chapter is reported in Sect. 3.6.

3.2 Non-Modular Operation in the RNS

In the RNS, traditional implementations of non-modular operations are based on the conversion of the operands from the RNS to the Integers. Conversion can be performed according to two main techniques: the Mixed-Radix Conversion (MRC) and the Chinese Remainder Theorem (CRT) [1, 2].

3.2.1 Mixed-Radix Conversion

In the Mixed-Radix System (MRS) based on the set of integer values R_1, R_2, ..., R_i, ..., R_n, a positive integer $X \in \left[0, \prod_{i=1}^{n} R_i - 1\right]$ is represented by the n-tuple $X \rightarrow$ (MRS) (a_n, a_{n-1}, ...,a_3,a_2,a_1) of mixed-radix digits, for which it results:

$$X = a_n \times \prod_{i=1}^{n-1} R_i + a_{n-1} \times \prod_{i=1}^{n-2} R_i + \cdots + a_3 \times R_1 \times R_2 + a_2 \times R_1 + a_1$$

where $0 \leq a_i \leq R_i$, $i = 1,2,\ldots,n$. Now, let m_1, m_2, ...,m_n be a set of pairwise relatively prime moduli, $M = m_1 \cdot m_2 \cdot \ldots \cdot m_i \cdot \ldots \cdot m_n$. The MRS associated with the RNS based on $m_1 \cdot m_2 \cdot \ldots \cdot m_i \cdot \ldots \cdot m_n$ is defined as the MRS for which $R_i = m_i$, $i = 1,2,\ldots,n$. In this case let $X \in [0, M-1]$, $X \rightarrow$ (RNS) (x_1, x_2, ...,x_i, ...,x_n), the conversion from the residue to the mixed-radix representation of X can be performed according to the following steps:

$$a_1 = x_1$$

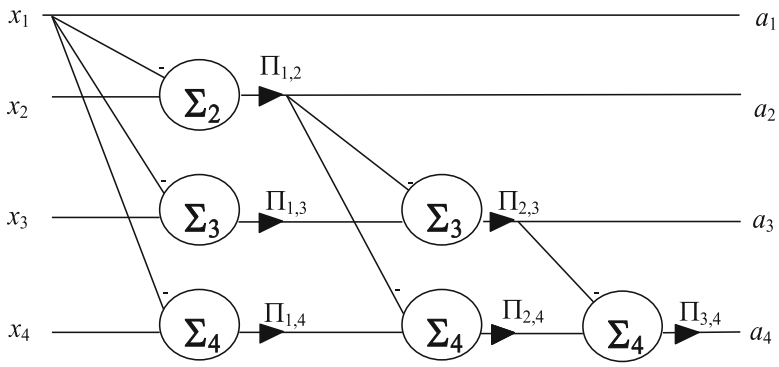

Fig. 3.1 The mixed-radix conversion process

$$a_2 = \left| \frac{X_M - a_1}{m_1} \right|_{m_2}$$

$$a_i = \left| \frac{|X|_M - a_1 - \sum_{j=2}^{i-1} \left(a_j \times \prod_{k=1}^{j-1} m_k \right)}{\prod_{j=1}^{i-1} m_j} \right|_{m_i} \qquad i = 3, 4, \ldots, n$$

Figure 3.1 shows the residue to mixed-radix conversion for an RNS of four moduli (Σ_j) means addiction modulo m_j, $\Pi_{i,j}$ means multiplication by $\left|\frac{1}{m_i}\right|_{m_j}$ modulo m_j.

3.2.2 Chinese Remainder Theorem

Let be $m_1, m_2, \ldots, m_n$ the set of moduli of the RNS and $X \in I, X \rightarrow (\text{RNS})(x_1, x_2, .., x_n)$, $I = [0, M-1]$, $M = m_1 \cdot m_2 \cdot \ldots \cdot m_n$. The CRT provides the traditional formula for residue-to-binary conversion [2]:

$$\forall X \in I \quad X \rightarrow (\text{RNS})\,(x_1, x_2, \ldots, x_n) \qquad X = \left| \sum_{i=1}^{n} N_i \times x_i \right|_M$$

where

$$N_i = M_i \times \left| \frac{1}{M_i} \right|_{m_i}, \; M_i = \frac{M}{m_i}, \; i = 1, \; 2 \ldots, n$$

Therefore, CRT supports non-modular operations by performing the complete residue-to-binary conversion of the operands.

For instance, magnitude comparison between $X \rightarrow (\text{RNS})(x_1,x_2,..,x_n)$ and $Y \rightarrow (\text{RNS})(y_1,y_2,..,y_n)$ can be performed as follows:

- Step 1. Computation of X and Y on the modulus M
- Step 2. Comparison between X and Y.

Now, let be $L(l,w)$ a look-up table of l locations with w-bit word length, and MOMA (n,A) an n-operand parallel adder modulo A, the implementation of the CRT requires n look-up tables $L(m_i, \lceil \log_2 M \rceil)$, $i = 1,2,..,n$ to store the products $N_i \times x_i$, for $i = 1,2,..,n$, as function of x_i, and a MOMA(n, M) for parallel summation [2].

3.3 The 'Diagonal Function' of the RNS

The 'diagonal function' exploits the observation that the integers in residue representation dispose themselves on diagonals when they are arranged in the multi-dimensional discrete space associated with the RNS [18, 19]. Figure 3.2 shows the space associated with the RNS with moduli $m_1 = 7$ and $m_2 = 5$. In this case $M = 35$ and the dynamic range I of the RNS is equal to $I = [0, 34]$. Figure 3.3 reports the labels of the diagonals of the space that are the outputs of the 'diagonal function' of the RNS.

Formally, let be $m_1,m_2,\ldots,m_n$ the set of moduli of the RNS. The 'diagonal function' is defined from the RNS to the integers as [16]:

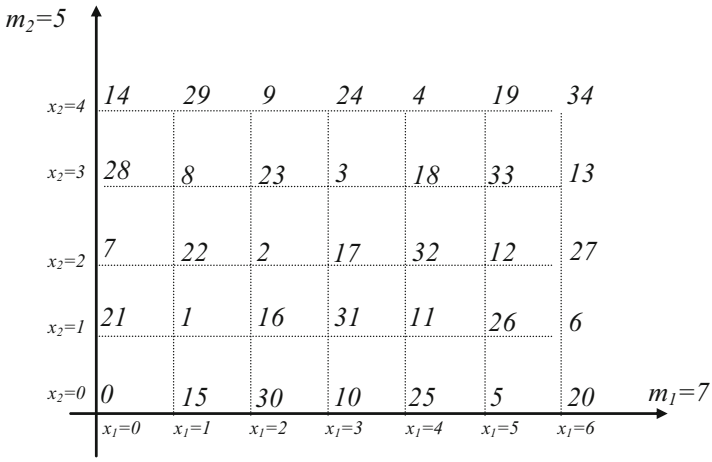

Fig. 3.2 Integers in RNS representation ($m_1 = 7$, $m_2 = 5$)

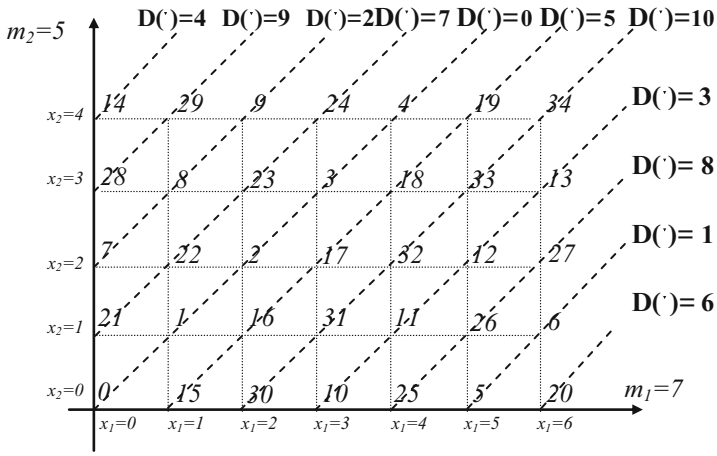

Fig. 3.3 The 'diagonal function': labels of the diagonals

$$\forall X \in I\ X \rightarrow (\text{RNS})\ (x_1, x_2, \ldots, x_n) : D(X) = \left| \sum_{i=1}^{n} k_i \times x_i \right|_{SQ}$$

where:

- $SQ = \sum_{i=1}^{n} M_i$, is the 'diagonal modulus' of the RNS $\left(M_i = \frac{M}{m_i},\ i = 1, 2, \ldots, n\right)$;
- $k_i = \left|-J_{m_i,\text{SQ}}\right|_{\text{SQ}}$, $\quad$ i $= 1, 2, \ldots, n$ ($J_{m_i,\text{SQ}}$ denotes the multiplicative inverse of m_i modulo SQ).

The following theorems define the specific characteristics of the 'diagonal function':

Theorem 1 (Characterization of the 'Diagonal Modulus'). *Let $m_1, m_2, \ldots, m_i, \ldots, m_n$ be a set of relatively prime moduli and let $SQ = \sum_{i=1}^{n} M_i$,, where $M_i = \frac{M}{m_i},\ i = 1, 2, \ldots, n$. Then it results:*
$SQ, m_1, m_2, \ldots, m_i, \ldots, m_n$ are relatively prime

Proof. According to the hypothesis $m_1, m_2, \ldots, m_i, \ldots, m_n$ are pairwise relatively prime.

Assume that SQ be not relatively prime with m_1. Then $\alpha, \beta, \Gamma \in N$ exist for which it results:

$$SQ = \alpha \times \beta \text{ and } m_1 = \alpha \times \Gamma$$

where

$$\beta = \frac{SQ}{\alpha} \text{ and } \Gamma = \frac{m_1}{\alpha}$$

Since

$$SQ = M_1 + M_2 + \cdots + M_n = M_1 + m_1 \times \left(\frac{M_2}{m_1} + \frac{M_3}{m_1} + \cdots + \frac{M_n}{m_1}\right)$$

It follows that:

$$\alpha \times \beta = m_2 \times m_3 \times \cdots \times m_n + \alpha \times \Gamma \times \left(\frac{M_2}{m_1} + \frac{M_3}{m_1} + \cdots + \frac{M_n}{m_1}\right)$$

hence

$$m_2 \times m_3 \times \cdots \times m_n = \alpha \times \left(\beta - \Gamma \times \left(\frac{M_2}{m_1} + \frac{M_3}{m_1} + \cdots + \frac{M_n}{m_1}\right)\right)$$

Considering that $\beta, \Gamma \in N$ and $\frac{M_j}{m_1} \in N$ for $j = 2,3,\ldots,n$, it follows that an integer $j \in [2, n]$ exists for which m_j is divisible by α. Hence m_1 is not relatively prime with m_j. Of course, the same consideration is true for each $i = 1,2,\ldots,n$. This contradicts the assumptions. □

Corollary (on the Existence of the 'Diagonal Function'). *Let $m_1, m_2, \ldots, m_i, \ldots, m_n$ be a set of relatively prime moduli of the RNS and let* $\mathrm{SQ} = \sum_{i=1}^{n} M_i$,, *where $M_i = \frac{M}{m_i}$, $i = 1,2,\ldots,n$. Let be $J_{m_i,SQ} = \left|\frac{1}{m_i}\right|_{SQ}$ for $i = 1,2,\ldots,n$, then $k_1, k_2, \ldots, k_N \in [0, SQ)$, exist so that:*

$$k_i = |-J_{m_i,SQ}|_{\mathrm{SQ}} \text{ for } i = 1, 2, \ldots, n$$

that is

$$|-k_i \times m_i|_{SQ} \equiv 1 \text{ for } i = 1, 2, \ldots, n$$

and they are unique (note that, for $i = 1,2,\ldots,n$, k_i is the additive inverse of the multiplicative inverse of m_i with respect to SQ).

Proof. The proof follows directly from Theorem 1 and from the property that for each couple of relatively prime numbers α and β, the multiplicative inverse $J_{a,b}$ exists and it is unique [1]. □

Theorem 2 (Property of the Coefficients of the 'Diagonal Function'). *Let m_1, $m_2, \ldots, m_i, \ldots, m_n$ be a set of relatively prime moduli and let $SQ = \sum_{i=1}^{n} M_i$,, where $M_i = \frac{M}{m_i}$, $i = 1,2,\ldots,n$. Let be $k_i = |-J_{m_i,SQ}|_{SQ}$, for $i = 1, 2, \ldots, n$, then:*

$$\left|\sum_{i=1}^{N} k_i\right|_{SQ} \equiv 0$$

Proof. From Theorem 1 it results that SQ is relatively prime with m_i, $i = 1,2,\ldots,n$. So SQ is relatively prime with M. Then $J_{M,\mathrm{SQ}}$ exists so that $|M \times J_{M,SQ}|_{SQ} \equiv 1$. Taking into consideration

$$\begin{aligned}
&\left|\sum_{i=1}^{n} k_i\right|_{SQ} = \left|\left(J_{M,SQ} \times M\right)\sum_{i=1}^{n} k_i\right|_{SQ} = \left|J_{M,SQ} \times \left(M \times \sum_{i=1}^{n} k_i\right)\right|_{SQ} = \\
&= |J_{M,SQ} \times (M_1 m_1 k_1 + M_2 m_2 k_2 + \cdots + M_N m_N k_N)|_{SQ} \\
&= \left|J_{M,SQ} \times \left(-\sum_{i=1}^{n} M_i\right)\right|_{SQ} = |-J_{M,SQ} \times (SQ)|_{SQ} \equiv 0.
\end{aligned}$$

□

Theorem 3 (Monotonicity of the 'Diagonal Function'). *Let m_1, $m_2, \ldots, m_i, \ldots, m_n$ be a set of relatively prime moduli and let $SQ = \sum_{i=1}^{n} M_i$,, where $M_i = \frac{M}{m_i}$, $i = 1,2,\ldots,n$. Let be $k_i = |-J_{m_i,SQ}|_{SQ}$, for $i = 1, 2, \ldots, n$, then the 'diagonal function' defined as:*

$$\forall X \in I\ X \to (\mathrm{RNS})\ (x_1, x_2, \ldots, x_n)\ D(X) = \left|\sum_{i=1}^{n} k_i \times x_i\right|_{SQ}$$

is monotonic growing in $I = [0, M-1]$.

Proof.

$$D(X) = \left|\sum_{i=1}^{n} k_i \times x_i\right|_{SQ} = \left|\sum_{i=1}^{n} k_i \times \left(X - \left\lfloor \frac{X}{m_i} \right\rfloor \times m_i\right)\right|_{SQ} = \left|X\sum_{i=1}^{n} k_i - \sum_{i=1}^{n} k_i m_i \left\lfloor \frac{X}{m_i} \right\rfloor\right|_{SQ}$$

From Corollary 1 and Theorem 2 it follows that:

$$D(X) = \left|\sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor\right|_{SQ}$$

Moreover, from $\left\lfloor \frac{X}{m_i} \right\rfloor < M_i$, $i = 1,2,\ldots,n$, it follows that

$$\sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor < \sum_{i=1}^{n} M_i = SQ \Rightarrow \left| \sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor \right|_{SQ} = \sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor$$

So, having $X < Y$, it follows that

$$\left\lfloor \frac{X}{m_i} \right\rfloor \leq \left\lfloor \frac{Y}{m_i} \right\rfloor, \quad i = 1, 2, \ldots, n$$

hence

$$\sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor \leq \sum_{i=1}^{n} \left\lfloor \frac{Y}{m_i} \right\rfloor$$

That is $D(X) \leq D(Y)$

□

Theorem 4 *Let $m_1, m_2, \ldots, m_i, \ldots, m_n$ be a set of relatively prime moduli and let $SQ = \sum_{i=1}^{n} M_i$,, where $M_i = \frac{M}{m_i}$, $i = 1,2,\ldots,n$. Let be $k_i = |-J_{m_i,SQ}|_{SQ}$, $i = 1, 2, \ldots, n$, and D(X) the diagonal function of the RNS.*

If X, Y∈I so that X < Y and D(X) = D(Y), then

$$x_i < y_i \text{ for } i = 1, 2, \ldots, n$$

Proof. If $D(X) = D(Y)$, then

$$\sum_{i=1}^{n} \left\lfloor \frac{X}{m_i} \right\rfloor = \sum_{i=1}^{n} \left\lfloor \frac{Y}{m_i} \right\rfloor$$

This means that

$$\left\lfloor \frac{X}{m_i} \right\rfloor = \left\lfloor \frac{Y}{m_i} \right\rfloor \text{ for } i = 1, 2, \ldots, n$$

In fact, assuming that an integer $i = 1,2,\ldots,n$ exists for which

$$\left\lfloor \frac{X}{m_i} \right\rfloor < \left\lfloor \frac{Y}{m_i} \right\rfloor,$$

it follows that an integer $j = 1,2,\ldots,n$, $j \neq i$ exists for which $\left\lfloor \frac{X}{m_i} \right\rfloor > \left\lfloor \frac{Y}{m_i} \right\rfloor$ and hence $X > Y$. This contradicts the hypothesis.

Now, let be

$$X = \left\lfloor \frac{X}{m_i} \right\rfloor \times m_i + x_i \quad \text{for } i = 1, 2, \ldots, n$$

$$Y = \left\lfloor \frac{Y}{m_i} \right\rfloor \times m_i + y_i \quad \text{for } i = 1, 2, \ldots, n$$

From the hypothesis $X < Y$ we have

$$\left\lfloor \frac{X}{m_i} \right\rfloor \times m_i + x_i < \left\lfloor \frac{Y}{m_i} \right\rfloor \cdot m_i + y \text{ for } i = 1, 2, \ldots, n$$

And hence, it follows that

$$x_i < y_i \text{ for } i = 1, 2, \ldots, n.$$

□

From the properties of the 'diagonal function' of the RNS, magnitude comparison between $X \rightarrow (\text{RNS})(x_1,x_2,..,x_n)$ and $Y \rightarrow (\text{RNS})(y_1,y_2,..,y_n)$ can be performed as follows [18]:

- Step 1. Computation of $D(X)$ and $D(Y)$ on the modulus SQ.
- Step 2. Comparison between X and Y through the scheme:
  - If $D(X) < D(Y)$, then $X < Y$;
  - If $D(X) > D(Y)$, then $X > Y$;
  - If $D(X) = D(Y)$, then, for any one of the integers $i = 1,2,..,n$:

    If $x_i < y_i$, then $X < Y$;
    If $x_i > y_i$, then $X > Y$;
    If $x_i = y_i$, then $X = Y$.

Concerning the implementation of the 'diagonal function', it requires n look-up tables L(m_i, $\lceil \log_2 \text{SQ} \rceil$), i=1,2,..,n to store the products $k_i \times x_i$, for $i = 1,2,..,n$, as function of x_i, and a MOMA(n, SQ) for parallel summation [2].

3.4 The 'Quotient Function' of the RNS

The 'quotient function' of the RNS exploits the consideration that the labels of the space associated with the RNS representation can be labelled according to various profitable strategies [21]. Figure 3.4 shows a possible labelling that can be obtained through a well-defined 'quotient function'.

Formally, let be $m_1,m_2, \ldots,m_n$ the set of moduli of the RNS. Moreover, let be $S \subset \{1,2,..,n\}$, $S \neq \varnothing$, the quotient function F_S is defined as follows [21, 22]:

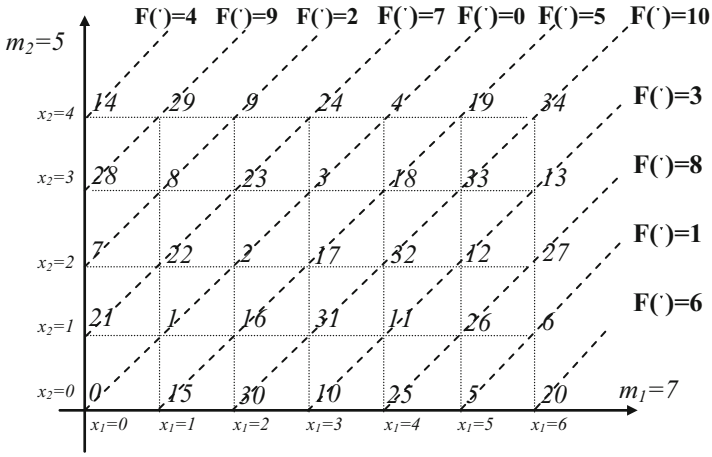

Fig. 3.4 A 'quotient function': labels of the diagonals

$$\forall X \in \mathrm{R} \quad X \to (\mathrm{RNS})\,(x_1, x_2, \ldots, x_n) \qquad F_S(X) = \sum_{i \in S} \left\lfloor \frac{X}{m_i} \right\rfloor.$$

Theorem 5 shows an effective scheme to compute $F_S(X)$ directly from the RNS representation of X [21].

Theorem 5 *Let m_1, m_2,.., m_n be the set of pairwise relatively prime moduli of the RNS, let be $S \subset \{1,2,..,n\}$, $S \neq \varnothing$, and*

$$M_i = \tfrac{M}{m_i},\ i = 1, 2, \ldots, N \ M_S = \sum_{i \in S} M_i, \qquad S_{\mathrm{INV}} = \sum_{i \in S} \left| \frac{1}{m_i} \right|_{M_S}$$

If $X \in [0, M-1]$, $X \to (RNS)\ (x_1, x_2, .., x_N)$, the value $F_S(X)$ can be computed directly from the residue representation of X as follows:

$$F_S(X) = \left| \sum_{i=1}^{n} b_i \times x_i \right|_{M_S}$$

with:

- $$b_i = -\left| \frac{1}{m_i} \right|_{M_S}, i \in S$$

- $b_j = \left| M_j \times S_{\mathrm{INV}} \times \left| \frac{1}{M_j} \right|_{m_j} \right|_{M_S}$, $j \in J$, *with* $J = \{1,2,\ldots,n\}$-S *(hence* $S \cup J = \{1,2,\ldots,n\}$ *and* $S \cap J = \varnothing$*).*

Proof. From $x_i = X - m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor$, $i = 1,2,\ldots,n$, it follows that:

$$\left| \sum_{i=1}^{S} b_i \times x_i \right|_{M_S} = \left| \sum_{i=1}^{n} b_i \times \left(X - m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor \right) \right|_{M_S} = \left| \sum_{i=1}^{n} b_i \times X - \sum_{i=1}^{n} b_i \times m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor \right|_{M_S}$$

$$= \left| \sum_{i \in S} b_i \times X + \sum_{j \in J} b_j \times X - \sum_{i \in S} b_i \times m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor - \sum_{j \in J} b_j \times m_j \times \left\lfloor \frac{X}{m_j} \right\rfloor \right|_{M_S}$$

$$= \left| \sum_{i \in S} \left| -\frac{1}{m_i} \right|_{M_S} \times X + \sum_{j \in J} M_j \times \left| \frac{1}{M_j} \right|_{m_j} \times S_{INV} \times X - \sum_{i \in S} \left| -\frac{1}{m_i} \right| \times m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor - \sum_{j \in J} M_j \right.$$
$$\left. \times \left| \frac{1}{M_j} \right|_{m_j} \times S_{INV} \times m_j \times \left\lfloor \frac{X}{m_j} \right\rfloor \right|_{M_S}$$

$$= \left| -S_{\mathrm{INV}} \times X + S_{\mathrm{INV}} \times X \times \left(\sum_{j \in J} \left| \frac{1}{M_j} \right|_{m_j} \right) + \sum_{i \in S} \left\lfloor \frac{X}{m_i} \right\rfloor - \sum_{j \in J} S_{\mathrm{INV}} \times X - \sum_{j \in J} (M_j \times m_j) \cdot S_{\mathrm{INV}} \right.$$
$$\left. \times \left| \frac{1}{M_j} \right|_{m_j} \times \left\lfloor \frac{X}{m_j} \right\rfloor \right|_{M_S}$$

$$= \left| X \times S_{\mathrm{INV}} \times \left(-1 + \sum_{j \in J} M_j \times \left| \frac{1}{M_j} \right|_{m_j} \right) + \sum_{i \in S} \left\lfloor \frac{X}{m_i} \right\rfloor - \sum_{j \in J} M \times S_{\mathrm{INV}} \times \left| \frac{1}{M_j} \right|_{m_j} \times \left\lfloor \frac{X}{m_j} \right\rfloor \right|_{M_S}$$

$= $ (See Note A, B)

$$= \left| 0 + \sum_{i \in S} \left\lfloor \frac{X}{m_i} \right\rfloor - 0 \right|_{M_S} = \left| \sum_{i \in S} \left\lfloor \frac{X}{m_i} \right\rfloor_{M_S} \right| = F_I(X).$$

□

Note A. Since $S \cup J = \{1,2,\ldots,n\}$, let $P_S = \prod_{i \in S} m_i$, $P_J = \prod_{j \in J} m_j$ then $P_S \times P_J = M$. Therefore, from the CRT it follows that an integer K exists so that [13]:

$$\left| \left(-1 + \sum_{j \in J} M_j \times \left| \frac{1}{M_j} \right|_{m_j} \right) \right|_{M_S} \equiv |K \times P_J|_{M_S} \equiv \left| K \times M \times \left| \frac{1}{P_S} \right|_{M_S} \right|_{M_S}.$$

Hence
$$\left| |X|_M \times S_{\mathrm{INV}} \times \left(-1 + \sum_{j \in J} M_j \times \left| \frac{1}{M_j} \right|_{m_j} \right) \right|_{M_S}$$
$$\equiv \left| |X|_M \times K \times S_{\mathrm{INV}} \times M \times \left| \frac{1}{P_S} \right|_{M_S} \right|_{M_S}.$$
$$\equiv \left| |X|_M \times K \times \left| \frac{1}{P_S} \right|_{M_S} \times M_S \right|_{M_S} \equiv 0$$

Note B. From $|M \times S_{\mathrm{INV}}|_{M_S} \equiv |M_S|_{M_S} = 0$, it results $\left|\sum_{j\in J} M \times S_{\mathrm{INV}} \times \left|\frac{1}{M_j}\right|_{m_j} \times \left\lfloor \frac{|X|_M}{m_j} \right\rfloor\right|_{M_S} \equiv \left|M_S \times \sum_{j\in J} \left|\frac{1}{M_j}\right|_{m_j} \times \left\lfloor \frac{|X|_M}{m_j} \right\rfloor\right|_{M_S} \equiv 0.$

Moreover, the coefficients b_k, $k\in\{1,2,\ldots,n\}$ are well defined if there exist the multiplicative inverses $\left|\frac{1}{M_j}\right|_{m_j}, j \in J$ and $\left|\frac{1}{m_i}\right|_{M_S}, i \in I$ (and hence $S_{\mathrm{INV}} = \sum_{i\in S} \left|\frac{1}{m_i}\right|_{M_S}$ is well defined too) [3]. This is true if and only if m_j,M_j and m_i,M_I are couples of relatively prime integers [1]. This is shown in Theorem A.

Theorem 6 *Let $m_1, m_2,.., m_n$ be the set of pairwise relatively prime moduli of the RNS and $S\subset\{1,2,..,n\}$, $S\neq\varnothing$. Let $M = \prod_{i=1}^{n} m_i$, $\mathrm{M_i} = \frac{M}{m_i}$ for $i = 1,2,\ldots,n$, and $M_S = \sum_{i\in S} M_i$, the following conditions are true:*

(a) *m_j and M_j are relatively prime;*
(b) *m_i and M_I are relatively prime.*

Proof.

(a) Assuming that m_j and M_j are not relatively prime, since $M_j = m_1 \cdot m_2 \cdot \cdots \cdot m_{j-1} \cdot m_{j+1} \cdot \cdots \cdot m_N$, *a* modulus m_i must exist, $i = 1,2,..,n$, $i\neq j$, which is not relatively prime with m_j. This contradicts the hypothesis.
(b) Assuming that m_i and M_I are not relatively prime, it follows that three integers $\overline{\alpha}, \overline{\beta}, \overline{\delta}$ exist so that $m_i = \overline{\alpha} \times \overline{\beta}$ and $M_S = \overline{\alpha} \times \overline{\delta}$ (with $\overline{\alpha} \neq 1$). Now, since

$$M_S = \sum_{k\in S} M_k = \sum_{k\in S} \frac{M}{m_k} = \left(\frac{M}{m_i} + m_i \times \sum_{\substack{k \in S \\ k \neq i}} \frac{M_i}{m_k} \right)$$

substituting $m_i = \overline{\alpha} \times \overline{\beta}$ and $M_S = \overline{\alpha} \times \overline{\delta}$ it results:

$$\overline{\alpha}\times\overline{\delta} = \frac{M}{m_i} + \overline{\alpha}\times\overline{\beta}\times \sum_{\substack{k=S \\ k\neq i}} \frac{M_i}{m_k} \quad \Rightarrow \quad \overline{\alpha}\times \left(\overline{\delta} - \overline{\beta}\times \sum_{\substack{k=S \\ k\neq i}} \frac{M_i}{m_k} \right) = \frac{M}{m_i} = \prod_{\substack{k=S \\ k\neq i}}^{N} m_k.$$

Thus, *a* modulus m_k exists, $k \neq i$, so that $\overline{\alpha}$ divides m_k. This means that m_k and m_i are not relatively prime moduli. This contradicts the hypothesis. □

The 'quotient function' is able to support both magnitude comparison and residue-to-binary conversion [21, 22].

Concerning magnitude comparison, let $X,Y \in [0,M-1]$ be two integers whose RNS representation is $X \rightarrow (\text{RNS})(x_1,x_2,\ldots,x_n)$ and $Y \rightarrow (\text{RNS})(y_1,y_2,\ldots,y_n)$, respectively. It results $X < Y \Rightarrow F_S(X) < F_S(Y)$ or $(F_S(X) = F_S(Y)$ and $x_i < y_i$, $i \in S)$. In fact, since $X = \left\lfloor \frac{X}{m_i} \right\rfloor \times m_i + x_i$, $Y = \left\lfloor \frac{Y}{m_i} \right\rfloor \times m_i + y_i$, it follows that if $X < Y$ and $F_S(X) = F_S(Y)$ (i.e. $\left\lfloor \frac{X}{m_i} \right\rfloor = \left\lfloor \frac{Y}{m_i} \right\rfloor$, for $i \in I$) it results that $x_i < y_i$, $i \in S$. Therefore, magnitude comparison can be performed as follows:

STEP 1. Compute $F_S(X)$ and $F_S(Y)$
STEP 2. Compare $F_S(X)$ and $F_S(Y)$; if $F_S(X) = F_S(Y)$ then compare x_i and y_i.

Concerning residue-to-binary conversion, let be $S = \{i\}$, it results $F_S(X) = \left\lfloor \frac{X}{m_i} \right\rfloor$. Now, since $X = m_i \times \left\lfloor \frac{X}{m_i} \right\rfloor + x_i$, $i = 1,2,\ldots,n$, it results:

$$X = m_i \times F_S(X) + x_i$$

If the modulus m_i is a power of 2, i.e., $m_i = 2^k$ (k integer), the implementation of X implies shift-left operation rather than ordinary multiplication and the binary representation of X is obtained by concatenating the binary representations of $F_S(X)$ (most significant bits of X) and x_i (least significant bits of X). In this case X is obtained as follows:

STEP 1. Compute $F_S(X)$, then do $X = F_S(X) \cup x_i$;
(where $F_S(X) \cup x_i$ is the concatenation of the binary representations of $F_S(X)$ and x_i).

Concerning the implementation of the 'diagonal function', it requires n look-up tables $L(m_i, \lceil \log_2 M_S \rceil)$, $i = 1,2,..,n$ to store the products $b_i \times x_i$, for $i = 1,2,..,n$, as function of x_i, and a MOMA(n, M_S) for parallel summation [22].

3.5 Performance Analysis

Let $m_1,m_2,\ldots,m_n$ be the set of relatively prime moduli of the RNS and let $X \in [0,M-1]$ be an integer whose RNS representation is $X \rightarrow (\text{RNS})(x_1,x_2,\ldots,x_n)$. Table 3.1 compares the MRC, the CRT, the 'diagonal function', and the 'quotient function' of the RNS. The MRC is based on a strictly sequential process. It has a time delay $O(n)$ and its ROM requirement is $O(n)$. The implementation of the CRT, 'diagonal function' and 'quotient function' have a time complexity $O(\log n)$, since the addition of n values can be performed in parallel using a tree of

Table 3.1 Performance analysis

		Time complexity	ROM	R-to-B conversion	Magnitude comparison
Serial technique	MRC	$O(n)$	$\Omega(n)$	Y	Y
Parallel techniques	CRT	$O(\log n)$	$\Omega(n^2)$	Y	Y
	D	$O(\log n)$	$\Omega(n^2)$	Not supported	Y
	F_S	$O(\log n)$	$\Omega(n^2)$	Y	Y

adders. Moreover, since the RNS moduli are pairwise relatively prime, it follows that necessarily m_i must be greater or equal than i, $\forall i = 1,2,\ldots,n$ (for instance, we have: $m_1 \geq 2$, $m_2 \geq 3$, $m_3 \geq 5$, $m_4 \geq 7$, and so on). Therefore, the total storage ROM is greater than $(1+2+3+\ldots+n) = \Omega(n^2)$ [3]. Moreover, the 'diagonal function' (D) does not support residue-to-binary conversion, whereas the CRT and the F_S (for $S = \{i\}$ and $m_i = 2^k$, k integer) support both magnitude comparison and residue-to-binary conversion [1, 18].

Anyway, it is worth noting that the 'diagonal function' and the 'quotient function' represent interesting mathematical tools not only for supporting non-modular operations in the RNS but also because they offer the possibility to understand better specific properties of the integers. Therefore it is not surprising that this topic attracts several researchers still today [23] even if the development of dedicated hardware for the implementation of new multi-operand modular adders makes possible CRT implementation more and more effective [14].

3.6 Conclusion

This chapter discusses some monotone functions recently introduced for supporting the implementation of non-modular operations in the Residue Number System. In particular two different classes of function are discusses: the 'diagonal function' and the 'quotient function' of the RNS. The functions are defined from the RNS to the Integers and exploit the observation that the integers in residue representation dispose themselves on diagonals when they are arranged in the multi-dimensional discrete space associated with the RNS. Both functions are monotone increasing and support effective implementations of non-modular operation in the RNS, also with respect to traditional approaches based on Mixed-Radix Conversion and Chinese Remainder Theorem.

References

1. S. Szabó, R.I. Tanaka, *Residue arithmetic and its applications to computer technology* (McGraw-Hill, New York, 1967)
2. F.J. Taylor, Residue arithmetic: a tutorial with examples. Computer **17**(5), 50–62 (1984)

3. A.A. Albert, *Fundamental concepts of higher algebra* (University of Chicago Press, Chicago, 1956)
4. S. Andraos, H. Ahmed, A new efficient memoryless residue to binary converter. IEEE Trans. Circ. Syst. **35**, 1441–1444 (1988)
5. K.M. Ibrahim, S.N. Saloum, An efficient residue to binary converter design. IEEE Trans. Circ. Syst. **CAS-35**, 1156–1158 (1988)
6. S. Chen, S. Wei, Weighted-to-residue and residue-to-weighted converters with three-moduli ($2^n - 1$, 2^n, $2^n + 1$) signed-digit architectures, in *Proceedings of 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, Island of Kos, Greece, 21–24 May 2006
7. W.W. Swamy, M.N.S. Ahmad, M.O.Y. Wang, A high-speed residue-to-binary converter for three-moduli ($2^k - 1, 2^k - 1, 2^{n-1} - 1$) RNS and a scheme for its VLSI implementations. IEEE Trans. Circuits Syst. II: Express Briefs **47**(12), 1576–1581 (2000)
8. B. Cao, C.-H. Chang, T. Srikanthan, A residue-to-binary converter for a new five-moduli set. IEEE Trans. Circuits Syst. I **54**(5), 1041–1049 (2007)
9. B. Cao, T. Srikanthan, C.-H. Chang, Design of residue-to-binary converter for a new 5-moduli superset residue number system, in *Proceedings of the International Symposium on Circuits and Systems (ISCAS 2004)*, Vancouver, Canada, vol. 2, pp. II-841-4, 23–26 May 2004
10. A. Skavantzos, M. Abdallah, Implementation issues of the two-level residue number system with pairs of conjugate moduli. IEEE Trans. Signal Process. **47**(3), 826–838 (1999)
11. B. Phillips, Scaling and reduction in the residue number system with pairs of conjugate moduli. in *Proceedings of the 37th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 2247–2251, 2003
12. S.J. Piestrak, Design off residue generators and multioperand modular adders using carry-save adders. IEEE Trans. Comput. **423**(1), 68–77 (1994)
13. S.J. Piestrak, A high-speed realization of a residue to binary number system converter. IEEE Trans. Circuits Syst. II **42**(10), 661–663 (1995)
14. S.J. Piestrak, A note on RNS architectures for the implementation of the diagonal function. Inform. Process. Lett. **115**(4), 453–457 (2015)
15. J. Gonnella, The application of core functions to residue number system. IEEE Trans. Signal Process. **39**(1), 69–75 (1991)
16. I.J. Akushskii, V.M. Burcev, I.T. Pak, A new positional characteristic of nonpositional codes and its applications, in *Coding Theory and the Optimization of Complex Systems*, V.M.Amerbsev ed., Kazah, 1977
17. D.D. Miller, R.E. Altschul, J.R. King, J.N. Polky, analysis of the residue class core function of akushskii, burcev and pak, in *Residue Number System Arithmetic, Modern Applications in Digital Signal Processing*, ed. by M.A.Soderstrand et al. IEEE Press,Paper 7-2, pp. 390–401
18. G. Dimauro, S. Impedovo, G. Pirlo, A new technique for fast numbers comparison in the residue number system. IEEE Trans. Comput. **42**(5), 608–612 (1993)
19. G. Dimauro, S. Impedovo, G. Pirlo, A new magnitude function for fast numbers comparison in the residue number system. Microprocess. Microprogram. **35**(1–5), 97–104 (1992)
20. G. Dimauro, S. Impedovo, G. Pirlo, A. Salzo, RNS architectures for the implementation of the diagonal function. Inform. Process. Lett. **73**, 189–198 (2000)
21. G. Dimauro, S. Impedovo, R. Modugno, G. Pirlo, R. Stefanelli, Residue-to-binary conversion by the quotient function. IEEE Trans. Circuits Syst—Part II **50**(8), 488–493 (2003)
22. G. Pirlo, D. Impedovo, A new class of monotone functions of the residue number system. Int. J. Math. Models Methods Appl. Sci. **7**, 802–809 (2013)
23. P.V. Ananda Mohan, RNS to binary conversion using diagonal function and pirlo and impedovo monotonic function. Circuits Syst. Signal Process. **35**(3), 1063–1076 (2016)

Chapter 4
Fault-Tolerant Computing in Redundant Residue Number System

Thian Fatt Tay and Chip-Hong Chang

4.1 Motivations

For the past decades, the device feature size has been aggressively scaled down to cope with the growing demands for smaller, faster, and less power hungry electronic products. Before the scaling of device feature size reaches the final 5 nm physical limit, the off-leakage current of the entire chip will become enormously large for feasible implementation [1]. The present rate scaling of device is achieved at the cost of the degradation of device reliability. When the feature size becomes smaller, the device is more susceptible to parameter variability. As a result, failure mechanisms subjected to parameter uncertainty will be magnified, which causes the reliability of the device to become increasingly unpredictable [2]. The shrinking of device node will also increase the stress factor. Due to the comparatively slower pace of operating voltage scaling, the oxide layer will experience higher electric fields, which will increase its susceptibility to device breakdown [2]. At transistor size of few tens of nano-meters and smaller, aging, noise, radiation, heat, and other dynamic conditions can easily perturb a device out of specification and cause transistors operating at low supply voltage to be susceptible to single event upsets and intermittent errors. Dependability of digital systems has to be significantly enhanced by incorporating fault tolerance into the design.

Unfortunately, classical fault-tolerant techniques such as error correction code, self-checking logic, module replication, and reconfiguration [3] are impractically expensive and non-scalable. Some of these fault diagnosis and reconfiguration approaches to counter soft errors require huge reliable memories or immense test and reconfiguration time. Representing data in Residue Number System (RNS)

T.F. Tay (✉) • C.-H. Chang
Nanyang Technological University, Singapore, Singapore
e-mail: tayt0015@e.ntu.edu.sg

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_4

can potentially relax the reliability requirement of low-level circuit by virtue of its parallel and modular arithmetic operations. Due to the isolation of carry forwarding in RNS, variations in circuit power, and timing or transient perturbation during computations may alter the states of many devices in the proximity but the cluster errors occurred in one modulus channel will not be propagated to the others [4].

By adding redundant moduli into an existing moduli set, a Redundant RNS (RRNS) is formed, which possesses error detection and correction capabilities by keeping the legitimate range for regular information processing. The residues error is detected by checking if the corresponding magnitude of the received residue representation falls into the illegitimate range. Upon confirming their existences, the residue errors have to be first located before correction. Contrary to two's complement number system (TCS), error detection and correction methods based on RRNS are able to correct arithmetical processing errors caused by circuit noise, manufacturing defects, process, voltage and temperature variations, or even deliberate fault injection in addition to transmission errors.

Despite its error resilient characteristics, RNS-based error detection and correction algorithm generally requires long computation time and large hardware implementation area. This is due to the iterative computations required to locate the erroneous residue digits sequentially. The number of computations is dependent on the size of the moduli set. Since the size of the moduli set escalates with the number of correctable residue errors, a large number of iterative computations is needed for multiple error detection and correction, causing their direct hardware implementation to be as difficult as, if not more complicated than, the implementation of several inter-modulo operations in RNS.

This chapter is organized as follows. Section 4.2 provides the fundamentals of RNS including residue representation, residue arithmetic, and residue-to-binary conversion. Section 4.3 presents the concept of detecting, locating, and correcting residue errors in redundant RNS (RRNS). The existing residue error detection and correction algorithms in RRNS can be categorized based on the correctable number of residue errors. The existing algorithms designed for single and multiple residue error detection and correction are discussed in Sect. 4.4 and 4.5, respectively. In Sect. 4.6, some applications that utilize RRNS codes in residue error detection and correction are presented. Finally, the chapter summary is given in Sect. 4.7.

4.2 Background and Preliminaries

4.2.1 Residue Number System

Numbers were created by human to count and measure objects. A numeral system is a way to represent the numbers uniquely where algebraic and arithmetic operations acted upon them can be carried out without ambiguity. Generally, there are two types

of numeral systems, i.e., weighted and non-weighted number systems. In a weighted number system with a set of weights, w_i, any number X can be expressed by

$$X = \sum_{i=1}^{n} a_i w_i \tag{4.1}$$

where a_i is a digit at the i-th position.

The most commonly used weighted number system in daily life is decimal number system. The w_i's of decimal number system are the successive powers of radix 10 and a_i is a digit in the symbol set $\{0, 1, \ldots, 9\}$. Meanwhile, the binary number system is a weighted number system widely used in computer arithmetic. It has a radix of 2 and a_i is a digit in the symbol set {0,1}. Modern-day digital processors are built using transistors which operate on the principle of two possible states, "off" and "on." Since binary number system has only two possible digits, "0" and "1," it is suitable to be used for data representation in digital processors where "0" and "1" represent the "off" and "on" states, respectively.

On the other hand, RNS is a non-weighted number system. An RNS is characterized by its base, which is a set of pair-wise relatively prime integers $\{m_1, m_2, \ldots, m_k\}$, where m_i is called modulus. An unsigned integer X within the range of $[0, M_K - 1]$ can be uniquely represented by a k-tuple $(x_1, x_2, \ldots x_k)$, where the dynamic range $M_K = \prod_{i=1}^{k} m_i$. The residue x_i is the least positive remainder of the division of X by m_i, and is usually represented as X mod m_i or $|X|_{m_i}$.

In the RNS of base $\{m_1, m_2, \cdots, m_k\}$, the residue representation of an integer $X' \in [C, M_K + C - 1]$ for any integer constant C can be mapped to the residue representation of an integer $X \in [0, M_K - 1]$ by using the following congruence:

$$X' \equiv |X - C|_{M_K} + C \tag{4.2}$$

For sign representation in RNS, C is chosen to be any integer in the range $(-M_K, 0)$. Equation (4.2) makes it possible to represent signed numbers in residue form while keeping the arithmetic units for the operations $\{+, -, \times\}$ compatible with those of the unsigned residues under the same RNS. If C is set to $-\left\lfloor \frac{M_K}{2} \right\rfloor$ where $\lfloor y \rfloor$ denotes the largest integer less than or equal to y, an almost symmetrical (for even M_K) or a symmetrical (for odd M_K) range of signed integers can be obtained. Under this signed residue representation, a signed integer $\tilde{X}$ in the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$ if M_K is even or $\left[-\frac{M_K - 1}{2}, \frac{M_K - 1}{2}\right]$ if M_K is odd can be uniquely represented by a k-tuple $(\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_k)$. Substituting $X' = \tilde{X}$ and $C = -\lfloor M_K/2 \rfloor$, Eq. (4.2) can be rewritten as

$$\tilde{X} \equiv \left| X + \left\lfloor \frac{M_K}{2} \right\rfloor \right|_{M_K} - \left\lfloor \frac{M_K}{2} \right\rfloor \tag{4.3}$$

4.2.2 Residue Arithmetic

In an RNS with moduli set $\{m_1, m_2, \ldots m_k\}$, the integers X and Y can be represented by $(x_1, x_2, \ldots, x_k)$ and $(y_1, y_2, \ldots, y_k)$, respectively. Let $Z = X \circ Y$ represents the result of the arithmetic operations acted upon X and Y. Z can be expressed as follows:

$$\begin{gathered} X \circ Y \equiv \left(|x_1 \circ y_1|_{m_1}, |x_2 \circ y_2|_{m_2}, \ldots, |x_k \circ y_k|_{m_k}\right) \\ Z \equiv (z_1, z_2, \ldots, z_k) \end{gathered} \tag{4.4}$$

where $z_i = |x_i \circ y_i|_{m_i}$ and "$\circ$" represents an arithmetic operation, such as addition, subtraction, multiplication, squaring, or exponentiation.

From Eq. (4.4), the residue digits of Z, $(z_1, z_2, \ldots, z_k)$, are computed by performing arithmetic operation on the residue digits of X and Y in the respective modulus channels. Therefore, the residue arithmetic operation in each modulus channel can be performed in parallel and independently without carry propagation across modulus channels. Since the word lengths of the residue digits are considerably shorter than those of the corresponding integers, the carry propagation delays of the residue arithmetic operations are lower than those of arithmetic operations performed in weighted number systems. More importantly, from the error resiliency point of view, the termination of carry across residue digits implies inherent error isolation and containment. The order independence of residue digits also offers a unique gracefully degradation property that enables an arithmetic system to work properly with a reduced dynamic range when any residue channel is faulty.

4.2.3 Residue-to-Binary Conversion

Since RNS is a non-weighted number system, the position of a residue digit does not represent its weight and the magnitude of a residue representation cannot be computed as straightforwardly as Eq. (4.1). The magnitude of an integer can be evaluated from its residue representation by using Mixed-Radix Conversion (MRC) or Chinese Remainder Theorem (CRT). Given the residue representation $(x_1, x_2, \ldots, x_k)$, the integer X can be computed using MRC as follows:

$$X = \sum_{i=1}^{k} a_i \prod_{j=1}^{i-1} m_j + a_1 \tag{4.5}$$

where a_i is called mixed-radix digit and it can be computed from $(x_1, x_2, \ldots, x_k)$ by

$$
\begin{aligned}
a_1 &= x_1 \\
a_2 &= \left|(x_2 - a_1)\left|m_1^{-1}\right|_{m_2}\right|_{m_2} \\
a_3 &= \left|\left((x_3 - a_1)\left|m_1^{-1}\right|_{m_3} - a_2\right)\left|m_2^{-1}\right|_{m_3}\right|_{m_3} \\
&\cdots \\
a_k &= \left|\left(\left((x_k - a_1)\left|m_1^{-1}\right|_{m_k} - a_2\right)\left|m_2^{-1}\right|_{m_k} - \cdots - a_{k-1}\right)\left|m_{k-1}^{-1}\right|_{m_k}\right|_{m_k}
\end{aligned}
\tag{4.6}
$$

MRC involves sequential computations whereby the evaluation of a_i is dependent on a_{i-1} as shown in Eq. (4.6) and its computation delay increases with the number of moduli. Therefore, MRC is not suitable for magnitude evaluation when the selected moduli set is large.

On the other hand, X can also be evaluated by CRT from its residue representation $(x_1, x_2, \ldots, x_k)$ as follows:

$$
X = \left|\sum_{i=1}^{k} M_i \left|M_i^{-1}\right|_{m_i} x_i\right|_{M_K} = \sum_{i=1}^{k} M_i \left|\left|M_i^{-1}\right|_{m_i} x_i\right|_{m_i} - r_k M_K \tag{4.7}
$$

where $M_i = M_K/m_i$, $\left|M_i^{-1}\right|_{m_i}$ is the multiplicative inverse of $|M_i|_{m_i}$, and r_k is an integer in the range $[0, k-1]$.

Using CRT, the computation of each partial sum, $M_i\left|\left|M_i^{-1}\right|_{m_i} x_i\right|_{m_i}$ can be performed in parallel and at a faster speed compared to the sequential computations of mixed-radix digits in MRC. Nonetheless, after the summation of the partial sum, $M_i\left|\left|M_i^{-1}\right|_{m_i} x_i\right|_{m_i}$, the result has to be modularly reduced by M_K to ensure that the final result X is in the range $[0, M_K - 1]$. Since M_K usually is a large integer, the modulo reduction operation is difficult to be implemented efficiently on hardware.

Residue-to-binary conversion is a bottleneck operation in RNS-based computations, and more so for error correction due to the larger dynamic range and arbitrary moduli. As will become apparent later, it is very difficult to find enough comparable word length special moduli of the form like 2^n and $2^n \pm 1$ with rich number theoretic property to simplify this operation. Consequently, look-up tables are used in place of full combinational circuits for efficient hardware implementation of error decoding in residue domain.

4.3 Redundant Residue Number System

By expanding the code space of a residue representation to include r additional residue digits into the moduli set $\{m_1, m_2, \ldots, m_k\}$, an (n, k) maximum distance separable code can be formed using an RRNS. An RRNS has a base of $n = k + r$ coprime moduli $\{m_1, m_2, \ldots, m_k, m_{k+1}, \ldots, m_n\}$ and a dynamic range of

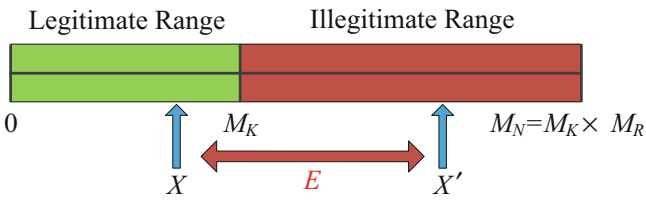

Fig. 4.1 Legitimate and illegitimate ranges of an RRNS

$M_N = \sum_{j=1}^{n} m_j$. The additional moduli, $m_{k+1}, m_{k+2}, \ldots, m_{k+r}$, are called the redundant moduli as opposed to the information moduli, $m_1, m_2, \ldots, m_k$, of the original base. The interval $[0, M_K - 1]$ of the k information residue digits of the n-tuple is called the legitimate range and the interval $[M_K, M_N - 1]$ due to the remaining r redundant residue digits is called the illegitimate range, as shown in Fig. 4.1. The legitimate range represents the useful computational range of the number system [5] whereas the illegitimate range is useful for error and overflow detection [6]. Let X be an integer within the legitimate range. When the residue digits $(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_n)$ of X are corrupted, X will be converted to $X' = X + E$, where E represents the error magnitude. X' always falls in the illegitimate range if $m_{k+1}, m_{k+2}, \ldots, m_{k+r} > m_i \ \forall i \leq k$ [7]. Thus, by checking the magnitude of X', the errors on the residue digits can be detected and then corrected. The error detection and correction capability of an RRNS is determined by its number of redundant moduli. With the addition of r redundant moduli, an RRNS is capable of detecting r and correcting up to $\lfloor r/2 \rfloor$ residue digit errors [7].

Typically, residue error detection and correction in RRNS are performed in three sequential steps: error existence check, identification of erroneous residue digits, and error correction. Residue error is detected by checking if the magnitude of the residue representation falls outside the legitimate computation range. The most straightforward method to compute magnitude is to perform full residue-to-binary conversion using CRT or MRC. These two methods usually require long computation time and complicated hardware, which escalate with the size of the chosen moduli set as indicated in Eqs. (4.6) and (4.7). On the other hand, the syndrome detection method avoids full magnitude evaluation on all received residue digits. Although more than one syndrome is required to check the existence of a residue digit error, the computations of syndromes are usually less time consuming and can be carried out in parallel. After confirming the existence of residue digit error, the next step is to identify the erroneous residue digits. This is the most difficult task among the aforementioned tasks. For single residue digit error detection and correction, the identification of the erroneous residue digit typically can be achieved by one computation. On the other hand, the identification of multiple erroneous residue digits is comparatively more time consuming and challenging due to the large number of combinations of different residue digit

locations and magnitudes. Upon identifying their locations, the erroneous residue digits can be independently corrected. The corrected output can either be computed in the form of integer or residue representation.

4.4 Single Residue Digit Error Detection and Correction Algorithms

The earliest single residue error detection and correction algorithm was introduced in [8]. By adding two redundant moduli, m_{k+1} and m_{k+2} into the RNS with base $\{m_1, m_2, \ldots, m_k\}$, the algorithm is able to detect and correct single residue digit error with the following conditions:

(a) m_{k+1} is greater than any information moduli;
(b) m_{k+2} is more than twice as large as any of the information moduli;
(c) $\prod_{i=k+1}^{k+2} m_i > 2\sum_{i=1}^{k}(m_i - 1) + \sum_{i=k+1}^{k+2}(m_i - 1)$.

A consistency checking method is introduced in the algorithm to detect erroneous residue digit. First, base extension method is used to compute y_{k+1} and y_{k+2} with respect to the received information residue digits, $X_K \equiv (x_1, x_2, \ldots, x_k)$. Then, y_{k+1} and y_{k+2} are compared with the received redundant residue digits, x_{k+1} and x_{k+2}. The differences are computed as $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$, respectively, i.e.,

$$|\Delta|_{m_{k+1}} = \left||X_K|_{m_{k+1}} - x_{k+1}\right|_{m_{k+1}} = |y_{k+1} - x_{k+1}|_{m_{k+1}} \tag{4.8}$$

$$|\Delta|_{m_{k+2}} = \left||X_K|_{m_{k+2}} - x_{k+2}\right|_{m_{k+2}} = |y_{k+2} - x_{k+2}|_{m_{k+2}} \tag{4.9}$$

The erroneous residue digit can be located by analyzing $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$. There are three possible scenarios:

(a) If both $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$ are zeros, then there is no erroneous residue digit.
(b) If either $|\Delta|_{m_{k+1}}$ or $|\Delta|_{m_{k+2}}$ is non-zero, then it can be concluded that the received residue digit of the corresponding non-zero $|\Delta|_{m_i}$ is erroneous. The erroneous residue digit is corrected by replacing it with the corresponding base extended residue digit, y_i.
(c) If both $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$ are non-zero, then one of the information residue digits is erroneous. The position of the erroneous residue digit and the error digit can be found by using $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$ as an address to a look-up table. The erroneous residue digit can be corrected by modularly subtracting the error digit from it.

Example 3.1. Let the chosen moduli set $\{m_1, m_2, m_3, m_4, m_5\}$ be {7, 8, 9, 11, 13}, where 13 and 17 are the redundant moduli. Assume that the received residue

representation is (3, 4, 1, 10, 11). The magnitude of the received information residue digits can be computed as $\tilde{X}_K \equiv (3, 4, 1) = 388$. Thus, the base extended residue digits y_4 and y_5 can be computed as $|388|_{11} = 3$ and $|388|_{13} = 11$, respectively. Since $x_4 = 10$ and $x_5 = 11$, $|\Delta|_{m_4}$ and $|\Delta|_{m_5}$ can then be computed using Eqs. (4.8) and (4.9) as $|3 - 10|_{11} = 4$ and $|11 - 11|_{13} = 0$, respectively. In this case, only $|\Delta|_{m_4}$ is non-zero, which indicates that $x_4 = 10$ is erroneous and its correct value should be $x_4 = y_4 = 3$. After the correction, the error-free residue representation is obtained as $(3, 4, 1, 3, 11) \equiv 388$.

In [9], it is proved that when there is an erroneous residue digit in an RRNS, the magnitude of the received residue representation will always fall outside the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$, where M_K is the product of the information moduli. Let $X \equiv \left(x_1, \ldots, x_p, \ldots, x_k, x_{k+1}\right)$ be the error-free residue representation and $X' \equiv \left(x_1, \ldots, x'_p, \ldots, x_k, x_{k+1}\right)$ be the residue representation with an erroneous residue digit located in the modulus channel m_p. The difference between X and X' can be computed as:

$$\begin{aligned} X - X' &\equiv \left(x_1, \ldots, x_p, \ldots, x_k, x_{k+1}\right) - \left(x_1, \ldots, x'_p, \ldots, x_k, x_{k+1}\right) \\ X - X' &\equiv \left(0, \ldots, \left|x_p - x'_p\right|_{m_p}, \ldots, 0\right) \\ X - X' &= a\left(\frac{M_K \cdot m_{k+1}}{m_p}\right) \end{aligned} \tag{4.10}$$

where a is a positive integer less than m_p. Since $a\left(\frac{M_K \cdot m_{k+1}}{m_p}\right) \geq M_K$ for $m_{k+1} \geq m_p$, X' always falls outside the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$ if X lies within the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$.

Based on this observation, a single residue error correction algorithm was proposed in [9] assuming that the two redundant moduli are m_{k+1} and m_{k+2}. The mixed-radix equivalent of X' is first computed using Eq. (4.5). If X' falls in the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$, there is no detectable error. Otherwise, there exists an erroneous residue digit, and all the numbers that are at a distance of $a\left(\frac{M_K \cdot m_{k+1} \cdot m_{k+2}}{m_i}\right)$ from X' are determined. X can then be determined by finding the values of i and a such that $\widehat{X} = \left|X' - a_i\left(\frac{M_K \cdot m_{k+1} \cdot m_{k+2}}{m_i}\right)\right|_{M_K}$ falls in the range $\left[-\frac{M_K}{2}, \frac{M_K}{2} - 1\right]$.

Figure 4.2 shows the flowchart of the single residue digit error detection and correction procedure proposed in [9].

The necessary and sufficient conditions for single residue digit error detection and correction in RRNS were derived in [4]. By using the derived minimal redundancy required for single residue digit error correction, a procedure for error correction was proposed. The procedure uses m_i-projection concept where the contribution of residue digit in the m_i modulus channel is ignored in computing the magnitude of residue representation, as shown in Fig. 4.3. For example, X_1 of m_1-projection is computed by performing the residue-to-binary computation on all residue digits except x_1. The location of the erroneous residue digit can be determined by checking X_i of the m_i-projection. If X_i falls in the legitimate range

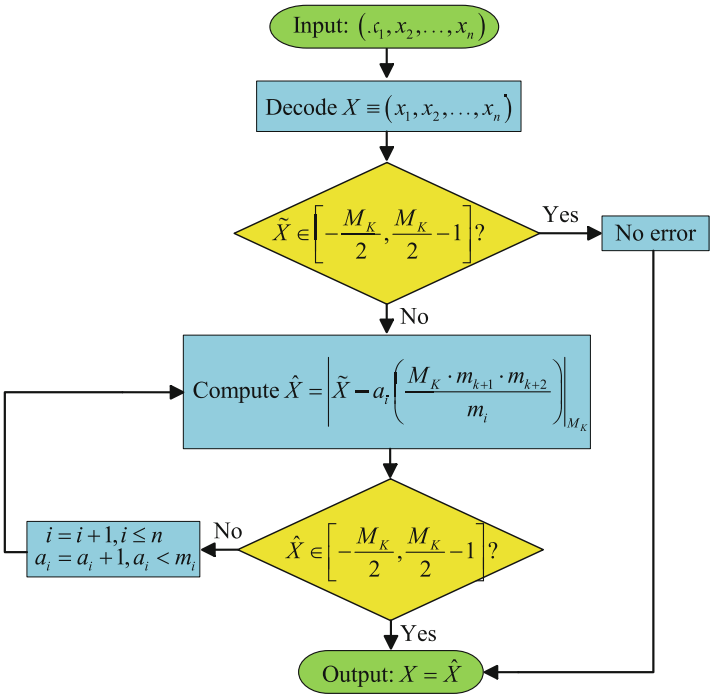

Fig. 4.2 Flowchart for detecting and correcting single residue digit error [9]

Fig. 4.3 The m_i-projection method for single residue digit error detection and correction

Moduli Set: $\{m_1, m_2, \cdots m_k, m_{k+1}, \cdots, m_n\}$

Received Residue Digits: $\{x_1, x_2, \cdots x_k, x_{k+1}, \cdots, x_n\}$

m_1-projection: $X_1 \equiv (x_2, x_3, \cdots x_k, \cdots, x_n)$

m_2-projection: $X_2 \equiv (x_1, x_3, \cdots x_k, \cdots, x_n)$

$\vdots$

m_n-projection: $X_n \equiv (x_1, x_2, \cdots x_k, \cdots, x_{n-1})$

$[0, M_K - 1]$, it can be concluded that the corresponding residue digit x_i ignored in the computation is erroneous and the magnitude of the error-free residue representation X is equal to X_i. This single residue digit error detection and correction procedure is illustrated in Fig. 4.4.

The first practical implementation of error detection and correction in RRNS was presented in [6]. The capability of overflow detection in digital filters in addition to single residue digit error correction was demonstrated by using the m_i-projection method proposed in [4] with the conditions of $m_1 < m_2 < \ldots < m_{k+r}$

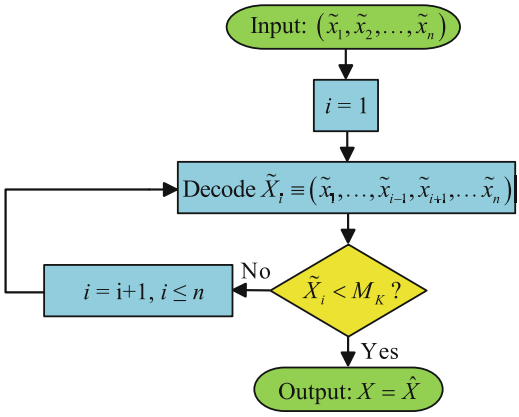

Fig. 4.4 Single residue digit error detection and correction procedure using the m_i-projection approach

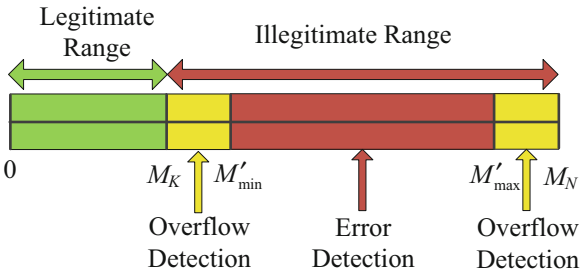

Fig. 4.5 Ranges for overflow and residue digit error detection

and the redundant moduli have to be larger than any of the information moduli. As discussed in Sect. 4.3, the existence of a residue digit error can be detected by checking if the magnitude of the received residue representation falls into the illegitimate range. According to [6], the full illegitimate range is not required for residue error detection. This is because $X' \equiv (x'_1, x'_2, \cdots, x'_{k+r})$ with a single erroneous residue digit will always reside in the range $[M'_{\min}, M'_{\max}]$, which is a subset of the illegitimate range defined by the chosen moduli set, as shown in Fig. 4.5. Therefore, the subranges $[M_K, M'_{\min} - 1]$ and $[M'_{\max} + 1, M_N - 1]$ can be used for detecting the positive or negative overflow. The algorithm proposed in [6] utilizes MRC method to check the range in which the received residue representation falls. It involves the computation of the mixed-radix digits of the redundant moduli using Eq. (4.6). If all of them are zeros, it can be concluded that the received residue representation is error-free. Otherwise, the mixed-radix digits are used to determine if there is a residue digit error or an overflow has occurred.

The authors of [10] presented a coding theory approach to error control in RRNS and introduced the concepts of Hamming weight, minimum distance, weight distribution, and error detection and correction capabilities in RRNS. Similar to the algorithms in [8], the algorithm relies on two redundant moduli and the base extension method to compute the test quantities, $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$ from the received information residue digits. From the derivations shown in [10],

$$|\Delta|_{m_{k+1}} \equiv \left| e_i^{(11)} \frac{M_N}{m_i} \right|_{m_{k+1}} = \left| \left(e_i^{(21)} - m_i \right) \frac{M_N}{m_i} \right|_{m_{k+1}} \tag{4.11}$$

$$|\Delta|_{m_{k+2}} \equiv \left| e_i^{(12)} \frac{M_N}{m_i} \right|_{m_{k+2}} = \left| \left(e_i^{(22)} - m_i \right) \frac{M_N}{m_i} \right|_{m_{k+2}} \tag{4.12}$$

where $e_i^{(11)}$, $e_i^{(12)}$, $e_i^{(21)}$, and $e_i^{(22)}$ are the error digits introduced into the modulus channel m_i. If both $|\Delta|_{m_{k+1}}$ and $|\Delta|_{m_{k+2}}$ are zeros, it can be concluded that there is no erroneous residue digit. Otherwise, $e_i^{(11)}$, $e_i^{(12)}$, $e_i^{(21)}$ and $e_i^{(22)}$ are computed from Eqs. (4.11) and (4.12) by substituting every possible m_i. The location of the erroneous residue digit can be determined by checking if $e_i^{(11)} = e_i^{(12)}$ or $e_i^{(21)} = e_i^{(22)}$. If either of the equations is satisfied, it can be confirmed that the residue digit in the modulus channel m_i is erroneous. The residue digit error can then be corrected by deducting $e_i^{(11)}$ or $e_i^{(21)}$ from the erroneous residue digit. Figure 4.6 shows the flowchart of the above-mentioned single residue digit error detection and correction algorithm.

In [11], the single residue digit error detection and correction algorithm is formulated by using two redundant moduli. Let $M_K = \prod_{j=1}^{k} m_j$ denotes the product of information moduli and $M_R = m_{k+1} \times m_{k+2}$ the product of redundant moduli. The redundant moduli have to be chosen such that the criteria, (1) $M_R > 2\sum_{j=1}^{k} \left(m_j - 1\right) + m_{k+1} + m_{k+2} - 2$ and (2) $M_R > 2 \times \max\left(m_i m_j\right)$ for $i, j \leq k$, are fulfilled. This algorithm is based on the syndrome approach to detect and locate the erroneous residue digit. The syndrome δ is computed as follows:

$$\delta = \left| \left|\tilde{X}\right|_{M_K} - \left|\tilde{X}\right|_{M_R} \right|_{M_R} \tag{4.13}$$

where $\left|\tilde{X}\right|_{M_K} \equiv (\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k)$ and $\left|\tilde{X}\right|_{M_R} \equiv (\tilde{x}_{k+1}, \tilde{x}_{k+2})$.

The aforementioned redundant moduli selection criteria ensure that the syndrome is unique. The computed syndrome is used as an address to a look-up table to retrieve the corresponding location of erroneous residue digit and the error digit. Upon identifying its location, the erroneous residue digit can be corrected by performing a simple modular subtraction:

$$x_i = |\tilde{x}_i - e_i|_{m_i} \tag{4.14}$$

where x_i, $\tilde{x}_i$, and e_i represent the error-free residue digit, the erroneous residue digit, and the error digit, respectively. The procedure for detecting and correcting a single residue error is depicted in the flowchart of Fig. 4.7. Figure 4.8 shows the hardware implementation, which consists of an RNS reverse converter for the information residues, a simple two residue CRT for the redundant residues, $k + 3$ modular subtractors, a look-up table, and simple logic gates.

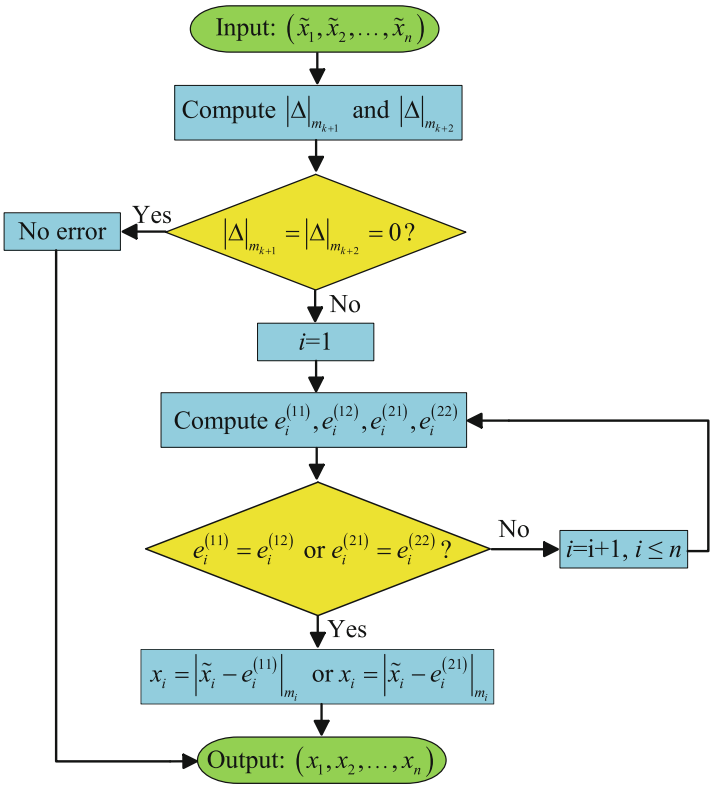

Fig. 4.6 Flowchart of single residue digit error detection and correction method [10]

In summary, the approaches taken by the existing algorithms to detect and correct single residue digit error can be divided into three categories, i.e., syndrome computation, range detection, and modulus projection as shown in Table 4.1. Among these categories, syndrome computation approach requires only a single computation to detect and locate the erroneous residue digit. Thus, the algorithms based on this approach are more amenable to hardware implementation. On the other hand, range detection and modulus projection approaches involve iterative computations. The number of computations required typically is determined by the chosen moduli set. Depending on the algorithms, after correcting the residue digit error, the output will either be represented in residue or integer form as indicated in Table 4.1. For applications which require error correction on intermediate arithmetical processing, the corrected output in residue form is more useful because the subsequent residue arithmetic operations can be carried directly without incurring any binary-to-residue conversion overhead. On the other hand, the corrected output in integer form is more useful when error correction is required in the final residue arithmetic operation or in some stages for which subsequent arithmetic operations or processing need

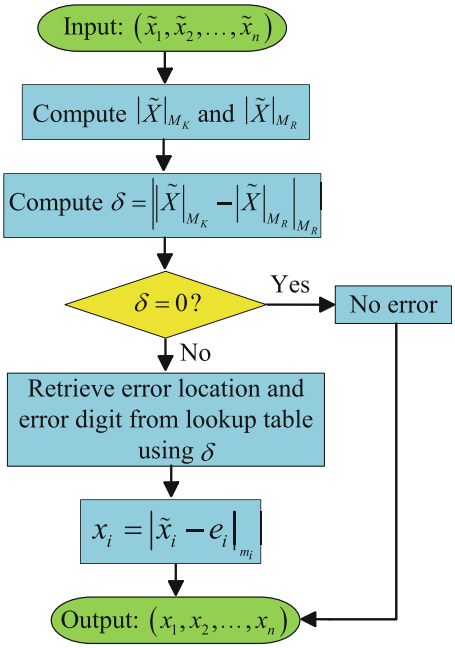

Fig. 4.7 Flowchart of the single residue digit error detection and correction algorithm [11]

to be performed in the normal binary domain. In this case, the reverse conversion overhead and latency can be eliminated.

4.5 Multiple Residue Digit Error Detection and Correction Algorithms

Detecting and correcting multiple residue digit errors normally require much more number of iterative computations than detecting and correcting single residue digit error. The complexity of the algorithm grows exponentially with the size of the chosen moduli set. The maximum number of correctable residue digit errors of an algorithm is determined by the number of redundant moduli in the chosen moduli set. The multiple residue digit errors correction algorithm proposed in [12] is capable of correcting up to t residue digit errors with $2t$ redundant moduli by using the base extension approach. Similar to the single residue digit error detection and correction algorithms that use the same approach, it requires the following computations of $|\Delta|_{m_i}$ by using the received redundant residue digits $\tilde{x}_i$ and the base extended residue digits, $|\tilde{X}_K|_{m_i}$:

$$|\Delta|_{m_i} = \left| |\tilde{X}_K|_{m_i} - \tilde{x}_i \right|_{m_i} \tag{4.15}$$

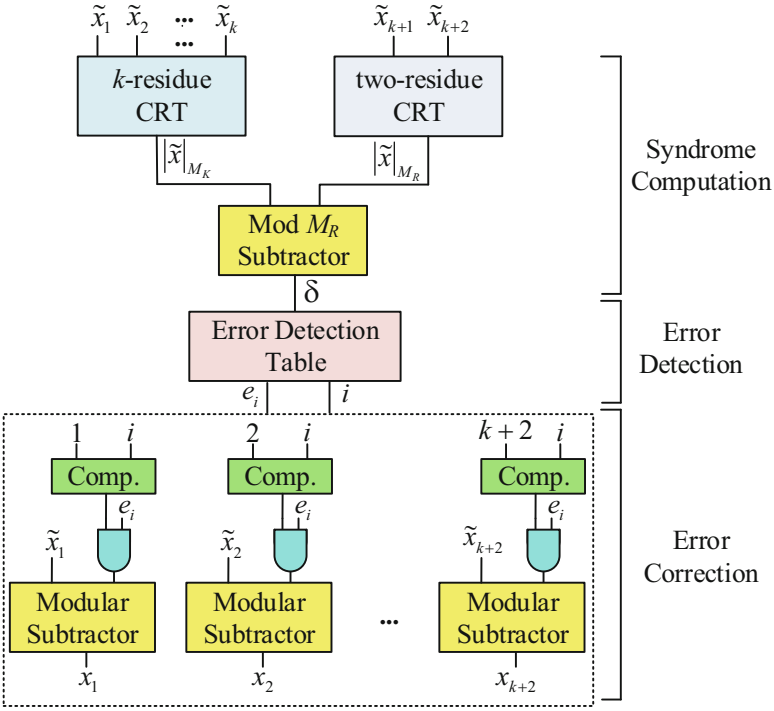

Fig. 4.8 Architecture for the hardware implementation of the residue digit error correction algorithm [11]

where $\tilde{X}_K \equiv (\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_k)$ represents the magnitude of the received information residue digits and $i = 1, 2, \ldots, k, k+1, \ldots, k+2t$.

After $|\Delta|_{m_i}$ have been computed, they are grouped into $\lceil (k+t-1)/(t+1) \rceil + 1$ groups as follows:

$$\begin{aligned} S_1 &= \left\{ |\Delta|_{m_{k+1}}, |\Delta|_{m_{k+2}}, \ldots, |\Delta|_{m_{k+2t}} \right\} \\ S_2 &= \left\{ |\Delta|_{m_{k-t}}, |\Delta|_{m_{k-t+1}}, \ldots, |\Delta|_{m_{k+t-1}} \right\} \\ S_3 &= \left\{ |\Delta|_{m_{k-2t-1}}, |\Delta|_{m_{k-2t}}, \ldots, |\Delta|_{m_{k-2}} \right\} \\ &\vdots \\ S_{\lceil (k+t-1)/(t+1) \rceil + 1} &= \left\{ |\Delta|_{m_1}, |\Delta|_{m_2}, \ldots, |\Delta|_{m_{2t}} \right\} \end{aligned} \tag{4.16}$$

The erroneous residue digits can be located by inspecting the elements of S_i. It can be concluded that the erroneous residue digits are located at modulus channels m_i if the corresponding S_i have less than or equal to t non-zero elements. The procedure for this multiple residue error detection and correction algorithm is illustrated by the flowchart in Fig. 4.9.

Table 4.1 Comparison of existing single residue error detection and correction algorithms

Algorithm	Approach	Correction output domain
[8]	Syndrome computation	Residue
[9]	Range detection	Integer
[4]	Modulus projection	Integer
[6]	Modulus projection	Integer
[10]	Syndrome computation	Residue
[11]	Syndrome computation	Residue

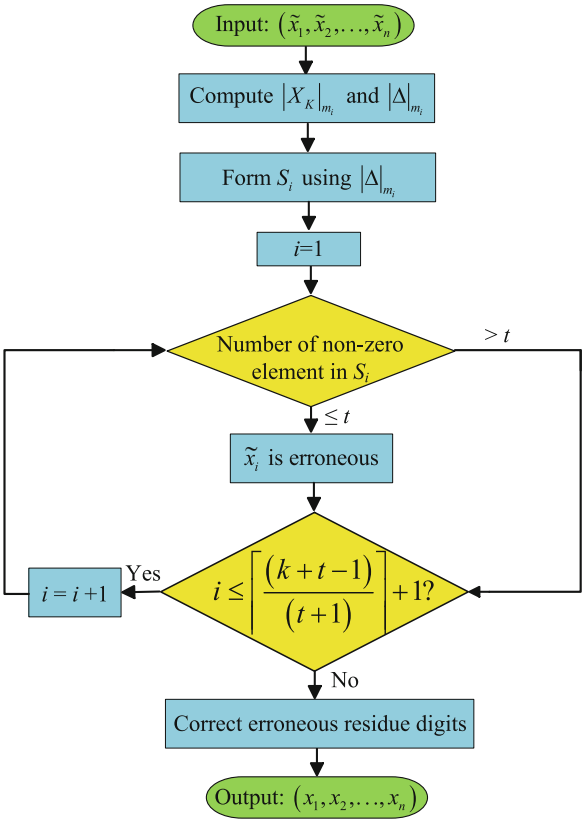

Fig. 4.9 Flowchart for the detection and correction of multiple residue digit errors [12]

The residue digit error correction algorithms proposed in [7] are based on CRT. Similar to the m_i-projection concept in [4], the algorithm computes the magnitude of the received residue by ignoring t residue digits at each iteration, where t is the number of correctable residue digit errors. If the computed magnitude of the iteration is within the legitimate range, it can be concluded that the ignored residue digits are erroneous. The maximum number of iterations required in this algorithm is ${}^{n}C_t$ where n is the total number of moduli. Let l_1, l_2, $\ldots$, l_{n-t} represent the

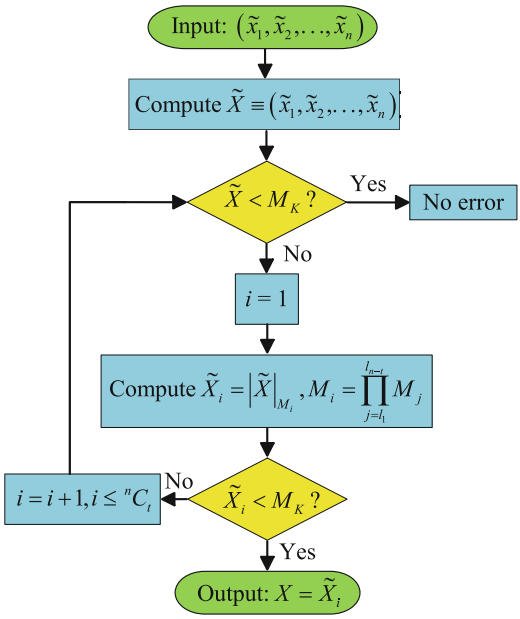

Fig. 4.10 Flowchart for multiple residue digit error correction algorithm [7] based on the m_i-projection concept

positions of the residue digits involved in each iterative computation. The multiple residue digit error detection and correction procedure is illustrated by the flowchart in Fig. 4.10.

Goh and Siddiqi [7] also proposed an extended multiple residue digit error detection and correction scheme to reduce the number of iterations by maximum likelihood decoding (MLD). The extended scheme computes the magnitude of t residue digits, where t is the number of correctable residue digits. When the computed magnitude is within the legitimate range, the remaining residue digits are computed by using the base extension method and then compared with the received residue digits. If the base extended residue digits differ from the received residue digits for t or less locations, it can be concluded that the base extended residue digits are the error-free residue digits. The number of iterations for this extended scheme is $\lceil {}^nC_t / {}^rC_t \rceil$. This is significantly lower than the first method proposed in [7], especially when the number of moduli is large. A numerical example is given below to illustrate its residue digit error detection and correction procedure.

Example 3.2. Consider the moduli set {7, 9, 11, 13, 17, 19}, where {7, 9} are the information moduli and {11, 13, 17, 19} are the redundant moduli. Since the moduli set has four redundant moduli, it is capable of correcting up to two residue digit errors. The legitimate and illegitimate ranges of the moduli set are [0, 62] and [63, 2909906], respectively. Let $X = 31 \equiv (3, 4, 9, 5, 14, 12)$. Assume that the residue digits $x_2 = 4$ and $x_5 = 14$ are corrupted such that the received residue representation becomes $\tilde{X} \equiv (3, 6, 9, 5, 10, 12)$. By using Eq. (4.7), the integer value of $\tilde{X}$ can be computed to be 19050. Since $\tilde{X}$ falls into the illegitimate range, it can be concluded

Table 4.2 The computations for locating erroneous residue digits

i	$\tilde{X}$	M_i	$Y_i = \lvert\tilde{X}\rvert_{M_i}$
1	19050	$M_1 = m_1 \times m_2 = 7 \times 9 = 63$	$Y_1 = \lvert 19050\rvert_{63} = 24$
2	19050	$M_2 = m_3 \times m_4 = 11 \times 13 = 143$	$Y_2 = \lvert 19050\rvert_{143} = 31$
3	19050	$M_3 = m_5 \times m_6 = 17 \times 19 = 323$	$Y_3 = \lvert 19050\rvert_{323} = 316$

Table 4.3 Base extension computations performed on Y_1 and Y_2

i	Y_i	$Z_i \equiv (z_1, z_2, z_3, z_4, z_5, z_6)$	$\tilde{X} \equiv (\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4, \tilde{x}_5, \tilde{x}_6)$
1	24	$Z_1 \equiv (3, 6, 2, 11, 7, 5)$	$\tilde{X} \equiv (3, 6, 9, 5, 10, 12)$
2	31	$Z_2 \equiv (3, 4, 9, 5, 14, 12)$	$\tilde{X} \equiv (3, 6, 9, 5, 10, 12)$

errors are introduced into the residue representation. The erroneous residue digits can be located by using MLD method where $Y_i = \left|\tilde{X}\right|_{M_i}$ $(M_i = \prod_{j=l_1}^{l_t} M_j)$ is first computed. According to the algorithm of [7], the number of computations required for locating the erroneous residue digits in this example is $\lceil {}^nC_t / {}^rC_t \rceil = \lceil {}^6C_2 / {}^4C_2 \rceil = 3$. Table 4.2 shows the computations of $Y_i = \left|\tilde{X}\right|_{M_i}$. Since both Y_1 and Y_2 are integers within the legitimate range, it is not possible to determine if Y_1 or Y_2 represents the magnitude of the error-free residue representation. Hence, base extension computations are required to locate the exact positions of the erroneous residue digits. The base extended residue digits z_j can be computed by performing modulo reduction on Y_i, i.e., $z_j = |Y_i|_{m_j}$ for $j = 1, 2, 3, 4, 5$, and 6. Table 4.3 shows the base extension computations performed on Y_1 and Y_2. Since Z_1 has four residue digits (more than the number of correctable residue digit errors), which are different from those of $\tilde{X}$, it can be concluded the residue representation of Z_1 is not the correct residue representation. On the other hand, Z_2 has exactly two residue digits which are different from those of $\tilde{X}$. Therefore, $Z_2 \equiv (3, 4, 9, 5, 14, 12)$ is the correct residue representation recovered by this algorithm.

The multiple residue digit error detection and correction algorithm proposed in [13] uses $2t$ number of redundant moduli to detect up to $2t$ and correct up to t residue digit errors. Upon detecting that a received residue representation is erroneous, the most straightforward method to locate the erroneous residue digit is to perform full residue-to-binary conversion on the received residue representation and use the converted magnitude as the address to a residue digit error look-up table. However, this method requires complex hardware implementation and long computation time. The algorithm described in [13] avoids these problems by dividing the received residue digits into three groups: information residue digit group (*IR*), redundant residue digit group 1 (*RR*1), and redundant residue digit group 2 (*RR*2) as shown in Fig. 4.11. By grouping the received residue digits into three distinct groups, any combination of residue digit errors can be categorically identified by one of the seven error location categories defined as follows:

$$\text{Moduli Set: } \{m_1, m_2, \cdots m_k, m_{k+1}, \cdots, m_{k+t}, m_{k+t+1}, \cdots, m_{k+2t}\}$$

$$\text{Received Residue Digits: } \left(\tilde{x}_1, \tilde{x}_2, \cdots \tilde{x}_k, \tilde{x}_{k+1}, \cdots, \tilde{x}_{k+t}, \tilde{x}_{k+t+1}, \cdots, \tilde{x}_{k+2t}\right)$$

$$IR\text{: } \tilde{X}_K \equiv \left(\tilde{x}_1, \tilde{x}_2, \cdots \tilde{x}_k\right)$$

$$RR1\text{: } \tilde{X}_U \equiv \left(\tilde{x}_{k+1}, \tilde{x}_{k+2}, \cdots, \tilde{x}_{k+t}\right)$$

$$RR2\text{: } \tilde{X}_V \equiv \left(\tilde{x}_{k+t+1}, \tilde{x}_{k+t+2}, \cdots, \tilde{x}_{k+2t}\right)$$

Fig. 4.11 The grouping of received residue digits

EL1: Erroneous residue digit(s) in *IR* only.
EL2: Erroneous residue digit(s) in *RR*1 only.
EL3: Erroneous residue digit(s) in *RR*2 only.
EL4: Erroneous residue digits in *IR* and *RR*1.
EL5: Erroneous residue digits in *IR* and *RR*2.
EL6: Erroneous residue digits in *RR*1 and *RR*2.
EL7: Erroneous residue digits in *IR*, *RR*1, and *RR*2.

To identify which error location category the erroneous residue digits fall into, the following three syndromes are computed:

$$\delta_1 = \left|\tilde{X}_U - \tilde{X}_K\right|_{M_U} \tag{4.17}$$

$$\delta_2 = \left|\tilde{X}_V - \tilde{X}_K\right|_{M_V} \tag{4.18}$$

$$\delta_3 = \left|\tilde{X}_V - \tilde{X}_U\right|_{M_V} \tag{4.19}$$

where $M_K = \prod_{i=1}^{k} m_i$, $M_U = \prod_{i=k+1}^{k+t} m_i$, and $M_V = \prod_{i=k+t+1}^{k+2t} m_i$. The error location category can be identified by checking if δ_1, δ_2, and/or δ_3 are non-zeros. There are five possible scenarios:

(a) $\delta_1 = \delta_2 = \delta_3 = 0$: there is no residue error.
(b) $\delta_1 \neq 0$, $\delta_2 \neq 0$, and $\delta_3 = 0$: the erroneous residue digit(s) is(are) in **EL1**.
(c) $\delta_1 \neq 0$, $\delta_2 = 0$, and $\delta_3 \neq 0$: the erroneous residue digit(s) is(are) in **EL2**.
(d) $\delta_1 = 0$, $\delta_2 \neq 0$, and $\delta_3 \neq 0$: the erroneous residue digit(s) is(are) in **EL3**.
(e) $\delta_1 \neq 0$, $\delta_2 \neq 0$, and $\delta_3 \neq 0$: the erroneous residue digit(s) could be in **EL4**, **EL5**, **EL6**, or **EL7**.

No error correction is needed for scenario (a). For scenario (b), (c), or (d), either one of the non-zero syndromes can be used as the address to an error look-up table to retrieve the error vector. When scenario (e) happens, all three syndromes are

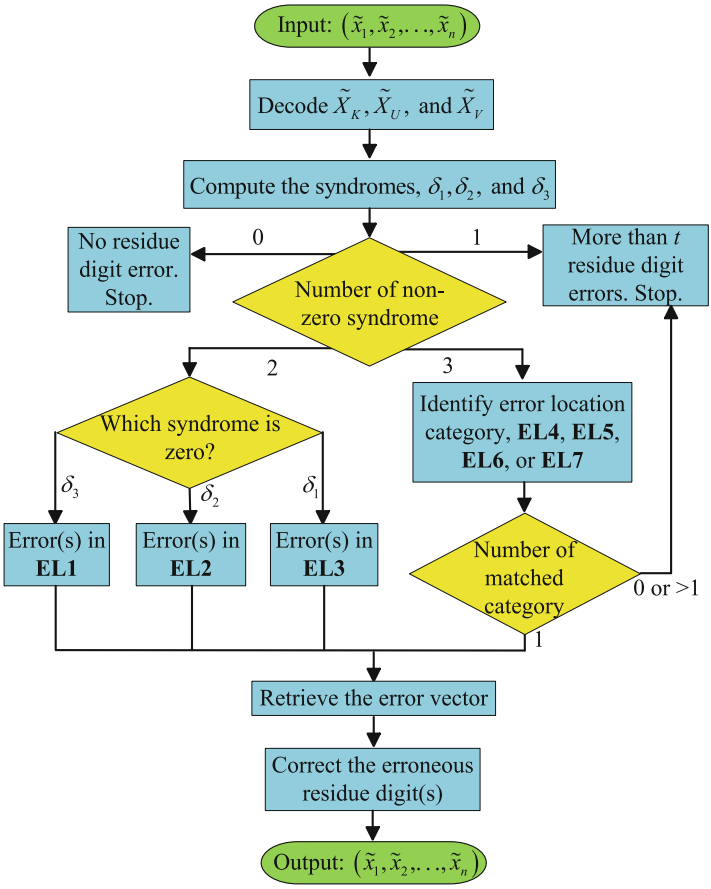

Fig. 4.12 The procedure for multiple residue error detection and correction in [13]

needed to retrieve the error vectors from three distinct error look-up tables. Upon retrieving the error vectors, the residue digit error(s) can be corrected by deducting the error vector from the received residue digits. Compared to the aforementioned method which uses the converted magnitude of residue representation as the address to a residue error look-up table, this syndrome-based method requires shorter computation time and less complex hardware as the bit-width of each syndrome is much smaller than that of the converted magnitude. The multiple residue error detection and correction procedure of [13] is illustrated by the flowchart depicted in Fig. 4.12.

Table 4.4 compares the two CRT-based approaches of [7], which are representative of a large majority of existing algorithms and the syndrome-based approaches of [12] and [13] for the detection and location of multiple residue digit errors, and the domains associated with their corrected outputs. [7] #1 and [7] #2 represent

the original and the extended schemes proposed in [7], respectively. The residue error detection approach determines the time consumed to identify the existence of residue digit error(s). If no error is detected, subsequent procedure of locating the residue digit errors is not required. Although only a single residue-to-binary computation is required for CRT approach, the computation of each syndrome involves smaller modulo operations and can be performed in parallel. Therefore, syndrome approach is more efficient than CRT approach in detecting the residue digit error(s) for the same chosen moduli set. In comparison to residue digit error detection and correction, locating the erroneous residue digits is usually more time consuming. The efficiency of this process dictates the overall delay of an algorithm. The algorithm of [12] solves the error location problem by using syndrome sets. The number of syndrome sets is determined by the size of the chosen moduli set. Similarly, the latency of the modulus projection approach adopted by [7] #1 also grows exponentially with the size of the chosen moduli set. Its maximum number of iterative computations is ${}^{n}C_{t}$. The time consumed by the improved modulus projection and MLD approach adopted by [7] #2 has less dependency on the size of moduli set. The maximum number of iterations has been reduced to $\lceil {}^{n}C_{t}/{}^{r}C_{t} \rceil$. The main drawback of this solution is the binary-to-residue conversion in each iterative computation, which requires expensive hardware implementation except only for a limited number of special moduli sets. On the other hand, the numbers of syndrome computations and look-up tables required by [13] are fixed regardless of the size of the moduli set. The locations of the erroneous residue digits and the corresponding error vector can be retrieved simultaneously from six look-up tables using the three computed syndromes. The complexity of this approach mainly lies in the sizes of the six look-up tables. Considering the uniqueness of error correction code in RRNS is the capability of detecting and correcting arithmetic operation errors in addition to transmission errors, it is beneficial to have the corrected output produced directly in the form of residue representation without the need for the reverse conversion. Among the algorithms in Table 4.4, only the corrected output of [12] is in the form of integer. If subsequent arithmetic operations are needed after the residue digit error correction, additional hardware will be incurred for the conversion into residue representation, which can be non-trivial for arbitrary moduli [14].

Table 4.4 Comparison of multiple residue error detection and correction algorithms

Algorithm	Approach		Corrected output domain
	Detect error	Locate error	
[12]	Syndrome	Syndrome sets	Residue
[7] #1	CRT	Modulus projection	Integer
[7] #2	CRT	Improved modulus projection + MLD	Integer and residue
[13]	Syndrome	Syndrome check	Residue

4.6 Applications of RRNS Codes in Error Detection and Correction

Error detection and correction capability of RRNS has been utilized in several fault-tolerant applications reported in literature [15–25]. In [15], the multicarrier modulation scheme is adaptively coded in RRNS with the aim to counter the frequency-selective fading effects caused by dispersive wide-band channels. The selected moduli sets consist of either three or six of information moduli with up to six redundant moduli. Each modulus has a word length of 8 bits and the chosen moduli set is {229, 233, 239, 241, 247, 251, 253, 255, 256}. To form an RRNS (9, 3) code which can be used to represent 24-bit data per code word and is capable of correcting up to three residue errors, {229, 233, 239} are chosen to be the information moduli while {241, 247, 251, 253, 255, 256} constitute the redundant moduli. According to the investigation on bit error rate (BER), the RRNS (9, 3) code exhibits the strongest error properties with a code rate of 0.33 in comparison to the counterparts such as RRNS (5, 3), (9, 3), (8, 6), (10, 6), and (12, 6) codes. The uniqueness of the scheme is the adjustable system performance in accordance to the targeted BER and bit per symbol (BPS) performance. For the targeted BER of 10^{-4}, the scheme outperforms all other convolution constituent code-based schemes in terms of BPS throughput when the signal-to-noise ratio (SNR) is above 15 dB.

Another application adopting RRNS-based error correction code reported in literature is hybrid memory [16]. Compared to conventional CMOS memory cells, hybrid memory, which is constructed using non-CMOS and CMOS nano-devices, can offer much larger data storage capacity on the same chip area. Nonetheless, hybrid memory faces high defect rate of 10 % owing to the non-mature manufacturing process and is highly susceptible to transient faults. The best solution to increasing the tolerance of hybrid memory towards faults is to incorporate error correction into it but the fault-tolerant hybrid memory system must possess concurrent error correction capability without being introduced with too much overhead. RRNS error correction code is considered as one of the best candidates due to its ability to correct cluster faults and inherited properties such as parallelism and modularity. However, the incorporation of RRNS error correction code requires large area overhead and long computation time. In [16], a modified version of RRNS error correction code, Six-Moduli RRNS (6 M-RRNS) is introduced. The chosen moduli include $\{2^n + 1, 2^n, 2^{n-1} - 1, 2^{n-2} - 1, 2^{n-3} - 1, 2^{n-4} + 1\}$ where $2^n + 1$ and 2^n are the information moduli and $2^{n-1} - 1$, $2^{n-2} - 1$, $2^{n-3} - 1$, and $2^{n-4} + 1$ are the redundant moduli. Unlike the conventional RRNS code, the redundant moduli are smaller than the information moduli. Decoding the code word in 6 M-RRNS using modulus projection approach will cause ambiguity in identifying the erroneous residue digits. It is suggested in [16] that the ambiguity problem can be solved by employing similar MLD technique proposed in [7]. To benchmark the performance of 6 M-RRNS code against the conventional RRNS (C-RRNS) and Reed–Solomon (RS) codes, the encoder and decoder of the error correction codes are implemented and synthesized using 90 nm CMOS technology library. The experimental results

show that the 6 M-RRNS code requires shorter code word lengths for 16-bit, 32-bit, and 64-bit memory in comparison to the other two codes. In other words, the 6 M-RRNS scheme can store more data for the same memory size.

Another unique application of RRNS is proposed in [17]. The technique utilizes the modularity nature of residue code to reduce the number of dropped messages in Ad Hoc networks caused by malicious nodes, buffer overflows, nodes movement, and collision. The suggested technique incorporates (n, k) RRNS code into a modified version of Ad hoc On-demand Multipath Distance Vector (AOMDV) routing protocol where a message is encoded into n number of residue digits and sent via multiple routes to the destinations. To enhance the reliability of the system, the RRNS code consists of ($r = n - k$) number of redundant residue digits. When there is d number of dropped or corrupted residue digits, the message arrives at the destination is recoverable as long as d is less than or equal to r. Figure 4.13 shows an example of the message delivery via six different routes. The message, X is encoded into six residue digits, (x_1, x_2, x_3, x_4, x_5, x_6) where x_4, x_5, and x_6 are the redundant digits. After encoding, the residue digits are transmitted to six nodes before being decoded at the destination. The message X can be successfully recovered if the number of dropped or corrupted residue digits is less than three. The

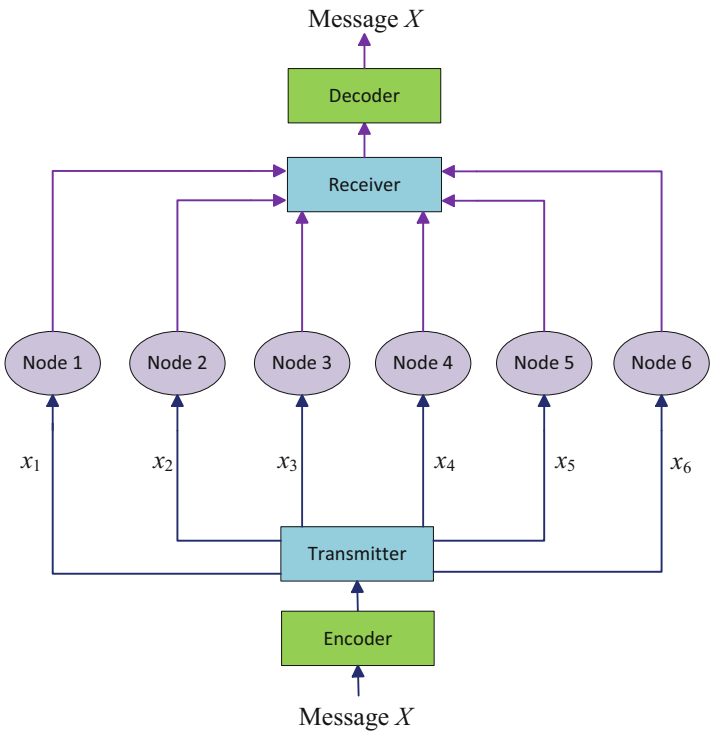

Fig. 4.13 Message delivery via six routes

performance of the router with the incorporation of RRNS code was benchmarked by counting the number of messages successfully delivered via nodes which are randomly distributed and moved to random waypoints. The experimental results in [17] show that the technique outperforms the conventional AOMDV in terms of message delivery rate via three, six, and nine routes.

4.7 Summary

Compared to classical fault-tolerant techniques, residue error detection and correction algorithm in RRNS possesses unique capability of correcting arithmetic operation errors in addition to transmission errors. Over the years, a great deal of research effort has been expended to explore RRNS for error detection and correction. These residue error detection and correction algorithms can be largely categorized according to their capability of correcting single or multiple residue digit errors. The error correction capability of an algorithm is determined by the number of redundant moduli. With the addition of $2t$ number of redundant moduli into the existing moduli set, an RRNS is capable of correcting up to t number of residue digit errors. The algorithms designed for correcting single residue digit error are generally simpler as only two redundant moduli are involved. Numerous approaches have been reported in literature but they can be generally classified into syndrome computation, range detection, and modulus projection. Among these approaches, syndrome computation has a greater advantage in terms of hardware implementation. This is mainly due to the avoidance of iterative computations in the location of erroneous residue digits. On the other hand, detecting and correcting multiple residue errors usually require longer computation time and more complex hardware, owing to the considerable efforts in iteratively checking all the possible combinations of different residue digit locations and magnitudes. Among the algorithms reported in the literature, the recent algorithm proposed in [13] is most distinctive. As only a fixed number of computations are required for locating erroneous residue digits, regardless of the size of the chosen moduli set, it is the more suitable algorithm for hardware implementation compared with the others.

References

1. H. Iwai, Roadmap for 22 nm and beyond. Microelectron. Eng. **86**(7–9), 1520–1528 (2009)
2. Y. Akasaka et al., in *Process integration, devices, and structures* (2011 ed.) [Online]. Available http://www.itrs.net/Links/2011ITRS/Home2011.htm
3. V.P. Nelson, Fault-tolerant computing: fundamental concepts. Computer **23**(7), 19–25 (1990)
4. F. Barsi, P. Maestrini, Error correcting properties of redundant residue number systems. IEEE Trans. Comput. **C-22**, 307–315 (1973)
5. G.A. Orton, L.E. Peppard, S.E. Tavares, New fault tolerant techniques for residue number systems. IEEE Trans. Comput. **41**(11), 1453–1464 (1992)

6. M. Etzel, W.K. Jenkins, Redundant residue number systems for error detection and correction in digital filters. IEEE Trans. Acoust., Speech, Signal Process. **28**(5), 538–545 (1980)
7. V.T. Goh, M.U. Siddiqi, Multiple error detection and correction based on redundant residue number systems. IEEE Trans. Commun. **56**(3), 325–330 (2008)
8. R.W. Watson, C.W. Hastings, Self-checked computation using residue arithmetic. Proc. IEEE **54**, 1920–1931 (1966)
9. N.S. Szabó, R.I. Tanaka, *Residue arithmetic and its applications to computer technology* (McGraw-Hill, New York, 1967), pp. 27–32
10. H. Krishna, K.Y. Lin, J.D. Sun, A coding theory approach to error control in redundant residue number systems. Part I: Theory and single error correction. IEEE Trans. Circuits Syst. **39**, 8–17 (1992)
11. T.F. Tay, C.H. Chang, A new algorithm for single residue digit error correction in Redundant Residue Number System, in *2014 Int. Symp. on Circuits and Syst.*, Melbourne, Australia, 2014, pp. 1748–1751
12. S.S.-S. Yau, Y.-C. Liu, Error correction in redundant residue number systems. IEEE Trans. Comput. **C-22**, 5–11 (1973)
13. T.F. Tay, C.H. Chang, A non-iterative multiple residue digit error detection and correction algorithm in RRNS. IEEE Trans. Compt. **65**(2), 396–408 (2015)
14. J.Y.S. Low, C.H. Chang, A new approach to the design of efficient residue generators for arbitrary moduli. IEEE Trans. Circuits Syst. Regul. Pap. **60**(9), 2366–2374 (2013)
15. T. Keller, T.-H. Liew, L. Hanzo, Adaptive redundant residue number system coded multicarrier modulation. IEEE J. Sel. Areas Commun. **18**(11), 2292–2301 (2000)
16. N.Z. Haron, S. Hamdioui, Redundant residue number system code for fault-tolerant hybrid memories. ACM J. Emerg. Technol. Comput. Syst. **7**(1), 4 (2011)
17. J. Alves Jr., L.F.L. Nascimento, L.C.P. Albini, Using the redundant residue number system to increase routing dependability on mobile ad hoc networks. Cyber Journals: J. of Selected Areas in Telecommunications **2**(1), 67–73 (2011)
18. L.-L. Yang, L. Hanzo, A residue number system based parallel communication scheme using orthogonal signaling—part I: system outline. IEEE Trans. Veh. Technol. **51**(6), 1534–1546 (2002)
19. H.T. How, T.H. Liew, E.-L. Kuan, L.-L. Yang, A redundant residue number system coded burst-by-burst adaptive join-detection based CDMA speech transceiver. IEEE Trans. Veh. Technol. **55**(1), 387–396 (2006)
20. P.E. Beckmann, B.R. Musicus, Fast fault-tolerant digital convolution using a polynomial residue number system. IEEE Trans. Signal Process. **41**, 2300–2313 (1993)
21. L. Hanzo, T. Liew, B. Yeap, Redundant residue number system codes, in *Turbo Coding, Turbo Equalisation and Space-Time Coding for Transmission over Fading Channels*, 1st edn. (Wiley-IEEE Press, Chichester, 2002), pp. 257–316
22. T.H. Liew, L.-L. Yang, L. Hanzo, Systematic redundant residue number system codes: analytical upper bound and iterative decoding performance over AWGN and Rayleigh channels. IEEE Trans. Commun. **54**(6), 1006–1016 (2006)
23. S. Zhang, L.-L. Yang, Y. Zhang, Redundant residue number system assisted multicarrier direct-sequence code-division dynamic multiple access for cognitive radios. IEEE Trans. Veh. Technol. **61**(3), 1234–1250 (2012)
24. S. Avik, N. Balasubramaniam, Performance of systematic RRNS based space-time block codes with probability-aware adaptive demapping. IEEE Trans. Wirel. Commun. **12**(5), 2458–2469 (2013)
25. V. Yatskiv, N. Yatskiv, J. Su, A. Sachenko, Z. Hu, The use of modified correction code based on residue number system in WSN, in *IEEE Int. Conf. Intelligent Data Acquisition and Advanced Computing Syst. (IDAACS) 2013*, Berlin, Germany, 2013, pp. 513–516

Chapter 5
Decimal Floating Point Number System

Hossam A.H. Fahmy

5.1 Need for Decimal Floating Point Formats

Computers have many limitations when dealing with numbers. Programmers know that integer data types are not suitable for any calculations resulting in fractions. For such calculations, they may use the floating point numbers either the "single precision" floating point numbers (stored in 32 bits, called **float** in several languages) or the "double precision" numbers (stored in 64 bits, called **double** in several languages). The IEEE 754-2008 standard for floating point arithmetic [1] calls these two precisions binary32 and binary64, respectively, to clarify their storage width and the fact that they use a binary base. The original standard in 1985 had the title "Standard for Binary Floating-Point Arithmetic" [2] but the 2008 revision [1] dropped the word binary from the title because the standard now describes both binary and decimal floating point arithmetic.

To understand why decimal is needed, consider the C code of Listing 5.1 where lines 5–7 define some variables and line 9 initializes x=0.1 and y=0.3. One would expect then that the values calculated on lines 11, 13, and 14 to be twox=0.2, threex =0.3, z1=0, and z2=0.

Lines 16–18 print the value of z1 with different format specifiers while line 19 prints the value of 3.0∗x−y when given directly as an argument to the printing function instead of first saving it in a variable. Line 21 prints z2 while line 23 prints the values of z1/z2 and (3.0∗x−y)/(2.0∗x−y+x) which one may expect to be both 0/0 giving an undefined quantity. The reality is that once we compile and run this code, line 23 prints "2 or 1"!

H.A.H. Fahmy (✉)
Electronics and Communications Engineering Department, Cairo University, Giza, Egypt

Electrical and Computer Engineering Department, Sultan Qaboos University, Muscat, Oman
e-mail: hfahmy@alumni.stanford.edu

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_5

Listing 5.1 A simple test: is $0.3 = 3 \times 0.1$?

```
#include <stdio.h>

int main (void)
   {
   double x, y;
   double twox, threex;
   double z1,z2;

   x=0.1;    y=0.3;

   twox    = 2.0*x;    threex = 3.0*x;

   z1= threex - y ;
   z2= twox   - y + x;

   printf("\n 3x-y  = %e",z1);
   printf("\n 3x-y  = %f",z1);
   printf("\n 3x-y  = %40.40f",z1);
   printf("\n 3x-y  = %e\n",3.0*x-y);

   printf("\n 2x-y+x= %e\n",z2);

   printf("\n (3x-y)/(2x-y+x)= %e or %e\n",z1/z2, (3.0*x-y)/(2.0*
       x-y+x));

   printf("\n (4/3-1)*3-1  = %e\n\n",(4.0/3.0 -1)*3.0-1.0);
   }
```

Results of the simple test from listing 5.1

```
3x-y   = 5.551115e-17
3x-y   = 0.000000
3x-y   = 0.0000000000000000555111512312578270211816
3x-y   = 2.775558e-17

2x-y+x= 2.775558e-17

(3x-y)/(2x-y+x)= 2.000000e+00 or 1.000000e+00

(4/3-1)*3-1   = -2.220446e-16
```

The bizarre results are easy to explain once we remember that the **double** data type defines a binary floating point number. In decimal notation, the fraction $(1/10)_{10} = (0.1)_{10}$ but in binary it is an infinitely recurring number $(0.000110011001100\ldots)_2$ which the computer rounds into the finite representation available in **double**. Similarly, y does not hold the exact value 0.3 but a rounded version. Then, each operation performed on these approximate values produces a result which is rounded as well before being saved in the required precision. The order of operations and their numbers causes different rounding decisions and hence a different final value.

Hence, for a computer *strictly* using binary64, if x=0.1 and y=0.30 then 3*x−y is $1.000\ldots00 \times 2^{-54} \approx 5.551 \times 10^{-17}$ while 2*x−y+x is $1.000\ldots00 \times 2^{-55} \approx 2.776 \times 10^{-17}$. Lines 16, 18, and 21 of the results provide these values. The emphasis on the word *strictly* in the previous sentence is important because it explains the value resulting from line 19. Certain processors and compilers store intermediate values in internal registers with wider precision which may cause a different rounding approximation. In this case, the wider precision caused the result to be 2^{-55} and not 2^{-54}. In light of these explanations, the results of line 23 appear now very logical!

Line 17 prints zero although the value of the variable z1 is not zero because the format specifier %f displays six digits after the fractional point unless the user specifically asks for a different number of digits as we have done in line 18. With this default behavior of %f we see on the screen a rounded version of the stored value in z1. In this specific case, the second rounding error causes the displayed value to correspond to the true mathematical value which we expect. In general, the effect of multiple rounding errors does not give the true mathematical result!

If all the above calculations were done in decimal, they would have produced the mathematical results we expect. The approximation errors might seem so small to the point of being negligible. However, line 23 shows that in some cases the calculations involving small approximation errors may get amplified and lead to unexpected results. With the large number of operations in a real application these errors can be significant. In fact, in a benchmark study on telecommunication billing [3], the errors resulting from the use of binary floating point numbers instead of decimal floating point numbers in the calculation of a company's bills amounted to a loss of about 5 million dollars per year. When it comes to financial calculations the use of decimal is a must [4].

Line 25 of the code shows us that although binary is not perfect decimal is not perfect either. The division by three results in a number that cannot be accurately represented in a finite precision binary number nor in a finite precision decimal number. Neither binary nor decimal can accurately represent transcendental numbers such as π or e. Finite precision computers will always have their limitations but at least decimal is more suited to many human calculations that binary is.

The suitability of decimal to human calculations was the reason that most machines used decimal representations in the early days of electronic computers. However, it was noted quickly [5] that binary is more suitable for the machine itself in many tasks such as addressing memories or to produce faster (approximate) results using a smaller number of logic gates. The time, area, and power savings due to the use of binary numbers outweighed the accuracy losses in computer hardware produced in the last 30 years of the twentieth century.

Nevertheless, decimal representations were needed for financial applications and were provided via software layers on top of the binary hardware. By the early years of the twenty-first century [4] the need for decimal hardware resurfaced due to the vast difference between the speed and energy consumption of a software versus a hardware solution [6]. Furthermore, the different software implementations were not all compatible. The revision of the IEEE standard responded to these changes

and included decimal floating point side by side with binary floating point. A system conforming to the standard may support either binary or decimal or both in a purely software manner or in a mixture of software and hardware.

Section 5.2 explains the representation of decimal digits and how the standard represents decimal floating point formats. Then Sects. 5.3–5.7 explain the basic operations on these formats. Section 5.8 illustrates some efforts on the verification of correct functionality for implementations while Sect. 5.9 describes potential embedded applications for decimal floating point arithmetic and briefly mentions the energy consumption.

5.2 Representation of Decimal Digits

To represent a single decimal digit using four bits we assign a different weight to each bit position then the value of the decimal digit equals the sum of the weights with a corresponding bit equal to one. The conventional Binary Coded Decimal (BCD) has the weights 8 4 2 1 for the four bits. Other codes use different weights as presented in Table 5.1. Some bit patterns are invalid for certain codes, in those cases, they are marked with an *i* in the table.

We see that, with the exception of 8 4 2 1, all the presented codes have redundant representations: the same number is represented by multiple bit patterns. However, this form of redundancy is different from what is described elsewhere in the current volume! Here the bits take only one of two values either 0 or 1 and do not exceed any "radix," the redundancy comes from the positional weights.

Table 5.1 Some decimal coding schemes

Pattern	8421	5421	4221	5211	6331	5221	5321	4421	2421	4311
1111	*i*	*i*	9	9	*i*	*i*	*i*	*i*	9	9
1110	*i*	*i*	8	8	*i*	9	*i*	*i*	8	8
1101	*i*	*i*	7	8	*i*	8	9	9	7	8
1100	*i*	9	6	7	9	7	8	8	6	7
1011	*i*	8	7	7	*i*	8	8	7	5	6
1010	*i*	7	6	6	9	7	7	6	4	5
1001	9	6	5	6	7	6	6	5	3	5
1000	8	5	4	5	6	5	5	4	2	4
0111	7	7	5	4	7	5	6	7	7	5
0110	6	6	4	3	6	4	5	6	6	4
0101	5	5	3	3	4	3	4	5	5	4
0100	4	4	2	2	3	2	3	4	4	3
0011	3	3	3	2	4	3	3	3	3	2
0010	2	2	2	1	3	2	2	2	2	1
0001	1	1	1	1	1	1	1	1	1	1
0000	0	0	0	0	0	0	0	0	0	0

Some combination of choices lead to incomplete codes such as the 6 3 3 1 code where there is no way to represent 2 nor 5.

Among the codes of Table 5.1, the 4 2 2 1, 5 2 1 1, 2 4 2 1, and 4 3 1 1 use all the possible sixteen combinations and do not have any invalid bit combinations. The 4 2 2 1, 5 2 1 1, and 2 4 2 1 have another interesting feature that does not exist in 4 3 1 1: the nines complement of a digit is equivalent to the ones complement of its binary representation. Designers use the properties of these various coding schemes (and others) to their advantage in many ways when building binary circuits for decimal numbers as we will see. In fact, most digital circuits are binary. Multi-valued logic is a mature field from the theoretical point of view. However, the design of circuits implementing multi-valued logic is a much harder task that does not scale easily to large systems. Hence, multi-valued logic has a very small market share. When designers need to implement a system with radix $\beta \neq 2$, they usually resort to codes similar to the ones we presented.

To illustrate their use, we briefly present the addition of two numbers using BCD digits. We assume in this problem that the input digits to the adder are each 4 bits with the normal BCD coding, i.e. the digit has the same value as the corresponding conventional binary encoding.

It is important to remember that any addition result exceeding the value of nine in a digit position must produce a carry to the next higher digit position. In a regular four bits binary adder, a carry is produced when the sum exceeds sixteen. A regular binary adder produces the primary sum bits p_3, p_2, p_1, p_0, and the carry c_4

$$\begin{array}{cccccc} & & a_3 & a_2 & a_1 & a_0 \\ & + & b_3 & b_2 & b_1 & b_0 \\ \hline & c_4 & p_3 & p_2 & p_1 & p_0 \end{array}$$

For example,

$$\begin{array}{rr} 5 & 0101 \\ +\,3 & +\,0011 \\ \hline 0\;\;8 & 0\;\;1000 \end{array}$$

For a BCD adder, we must indicate if the sum exceeds nine and produce the correct results in BCD. The sum exceeds nine when there is a carry out of the 4 bit binary adder or if the bits of the resulting digit are of the form: 101x or 11xx as in

$$\begin{array}{rr} 8 & 1000 \\ +\,9 & +\,1001 \\ \hline 1\;\;7 & 1\;\;0001 \end{array} \rightarrow 1\,0111 \quad \text{and} \quad \begin{array}{rr} 5 & 0101 \\ +\,6 & +\,0110 \\ \hline 1\;\;1 & 0\;\;1011 \end{array} \rightarrow 1\,0001$$

then the decimal carry out signal is

$$c_{\text{out}} = c_4 + p_3(p_2 + p_1)$$

If a carry is produced, we must correct the value of the resulting digit. This correction compensates for the six values that are not used in the case of BCD but that exist in the case of binary. Hence, we add 0110 to the primary sum.

Another way of looking at this correction would be to subtract ten from the primary sum since we are now generating a carry. The subtraction of ten is done by the addition of its two's complement, i.e. by adding 10110 to the *whole* digit including c_4. We then take the least significant 4 bits and any carry produced is conveyed to the next higher digit position. Whichever way, we correct the least significant bits by adding 0110.

This simple addition example shows why conventional BCD circuits use more logic gates and are slower than binary. Furthermore, conventional BCD uses 4 bits which can represent 16 different values to represent only 10; it is not an economical encoding for memory storage. The standard committee members and the designers who proposed circuit implementations for its operations attempted to circumvent all these hurdles.

To represent a floating point number with radix $\beta = 10$ such as -5.681×10^{53} the standard specifies three parts to denote its value. The sign of the number is encoded in a single bit (s) which is 1 for negative numbers and 0 for positive numbers. The second part is the exponent $e = 53$ while the number 5.681 constitutes the third part $m = 5.681$ called the significand. The original number is thus given by $(-1)^s \times \beta^e \times m$ where m has up to p significant digits starting from digit d_0 directly to the left of the fractional point and going down to digit d_{p-1} located $p - 1$ positions to the right of the fractional point. Note that, mathematically, $(-1)^s \times \beta^e \times m = (-1)^s \times \beta^q \times c$ where

$$m = d_0.d_{-1}d{-}2 \ldots d_{p-1},$$
$$c = d_0 d_{-1} d{-}2 \ldots d_{p-1}, \text{and}$$
$$q = e - (p - 1).$$

The view of the significand as an integer c could simplify the discussion of decimal floating point encoding as we will see shortly. The standard defines the decimal interchange floating point formats as:

	Sign	Combination	Trailing significand
	$\pm$	exponent and 4 MSB	$t = 10J$ bits
64 bits:	1 bit	13 bits, bias = 398	50 bits, 15 + 1 digits
128 bits:	1 bit	17 bits, bias = 6176	110 bits, 33 + 1 digits

with two basic formats decimal64 which has 64 bits and decimal128 which has 128 bits. Shorter (decimal32 with 32 bits) and wider (to allow for extended precisions) formats are also defined but the basic formats are decimal64 and decimal128. For finite numbers, the combination field encodes the exponent ($q +$ bias) and four significand bits. When the trailing significand field is combined with the leading significand bits from the combination field we get the complete significand. The combination field may also encode an infinite value ($\pm\infty$) or a Not-a-Number (NaN) value.

The standard [1] gives all the necessary details of the encodings. In our discussion here, we will concentrate more on finite numbers and the operations related to them.

The decimal formats in the standard also provide a feature which does not exist in binary formats: leading zeros. To explain it let us consider the question of doing a measurement in a lab and finding the mass of a body, *is* 0.050 kg = 0.05 kg? From a pure mathematical point of view the two numbers are equal but from a measurement point of view we assume that the 0.050 kg represents a measurement accurate to the nearest gram while 0.05 kg is for a measurement accurate to the nearest 10 g.

If both measurements are normalized to 5×10^{-2} and stored in a format with 16 digits as $(5.000000000000000 \times 10^{-2})$, they are

- indistinguishable and
- give the incorrect impression of a much higher accuracy (0.0500000000 00000 kg).

To maintain the distinction, we should store

$$\begin{array}{ll} 0.000000000000050 \times 10^{12} & \text{First measurement} \\ 0.000000000000005 \times 10^{13} & \text{Second measurement} \end{array}$$

with all those leading zeros. For the standard, both representations are members of the same *cohort*.

Hence, the integer c representing the significand may have leading zeros as its most significant digits. Given that c is an integer, we may represent it in normal binary notation for integers not in BCD to allow the use of normal binary hardware units. The design of significand addition and multiplication circuits is probably easier this way, however, the detection of digit boundaries for rounding or for exponent adjustment is much more difficult. Let us start to explain the operations to understand this.

5.3 Decimal Addition

5.3.1 Floating Point Addition

The following decimal addition uses a simplified system with four digits only to illustrate the ideas.

$$\begin{array}{rl} 1.324 & \times 10^5 \\ +\ 1.576 & \times 10^3 \end{array} \quad \left\{ \begin{array}{rll} & 1.324 & \times 10^5 \\ + & 0.01576 & \times 10^5 \\ \hline & 1.33976 & \times 10^5 \\ \approx & 1.340 & \times 10^5 \end{array} \right.$$

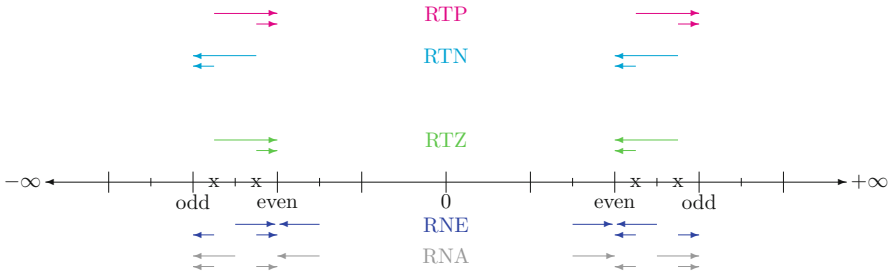

Fig. 5.1 Rounding directions. The *cross* marks indicate locations of precise results. The *big tick* marks indicate representable floating point numbers. The *short tick* marks represent tie cases

We start by calculating the difference between the exponents and then shift the smaller number to the right a number of places equal to that difference (i.e., divide it by $10^{\text{difference}}$, in this case 10^2). The digits shifted out, which are marked in gray, will be eventually rounded off. The value of the rounded digits is greater than half the digit in the least significant place so the rounding to nearest causes the result to become 1.340×10^5. A left shift of the larger number could also align the exponents of the two numbers, however the digits shifted out on the left are the most significant digits of the result. Surely, these digits are much more valuable than digits shifted at the least significant side to the right. Hence, it is the smaller number that is shifted for exponent alignment.

Figure 5.1 shows the five rounding directions of the standard.

1. Round To Positive (RTP): the result is the floating point number closest and no less than the infinitely precise result.
2. Round To Negative (RTN): the result is the floating point number closest and no greater than the infinitely precise result.
3. Round To Zero (RTZ): the result is the floating point number closest and no greater in magnitude than the infinitely precise result.
4. Round to Nearest with ties to Even (RNE): the result is the floating point number closest to the infinitely precise result. In case two floating point numbers are equally near, the one with an even least significant digit is chosen.
5. Round to Nearest with ties Away from zero (RNA): the result is the floating point number closest to the infinitely precise result. In case two floating point numbers are equally near, the one larger in magnitude is chosen.

The steps mentioned so far are similar to what exists in a binary floating point addition. Decimal floating point numbers may also have leading zeros as in

$$\begin{array}{rll} & 1.324 & \times 10^3 \\ + & 0.076 & \times 10^4 \end{array} \left\{ \begin{array}{rll} & 1.324 & \times 10^3 \\ + & 0.760 & \times 10^3 \\ \hline & 2.084 & \times 10^3 \end{array} \right.$$

where, due to the leading zeros, the smaller number here is not the one with the smaller exponent! In effect, the difference of exponents is less than the number of leading zeros in the number with the larger exponent. The 0.076 is thus shifted one digit to the left (multiplied by 10) to equate the exponents without losing any significant digits and the final result is 2.084×10^3. On the other hand, a shift of 1.324 one digit to the right (division by 10) yields the less accurate result 0.208×10^4.

What happens when the exponent difference is more than the number of leading zeros in the number with a larger exponent?

$$\begin{array}{rll} & 1.324 & \times 10^3 \\ + & 0.076 & \times 10^6 \end{array} \left\{ \begin{array}{rll} & 0.1324 & \times 10^4 \\ + & 7.600 & \times 10^4 \\ \hline & 7.7324 & \times 10^4 \\ \approx & 7.732 & \times 10^4 \end{array} \right.$$

Here, both numbers may be shifted! The leading zeros in decimal floating point numbers cause a slightly harder decimal alignment step when compared to alignment in binary floating point addition.

It is important to note that the above calculations are equivalent to

$$\begin{array}{rll} & 1324 & \times 10^2 \\ + & 1576 & \times 10^0 \end{array} \left\{ \begin{array}{rll} & 1324 & \times 10^2 \\ + & 001576 & \times 10^2 \\ \hline & 133976 & \times 10^2 \\ \approx & 1340 & \times 10^2 \end{array} \right.$$

$$\begin{array}{rll} & 1324 & \times 10^0 \\ + & 0076 & \times 10^1 \end{array} \left\{ \begin{array}{rll} & 1324 & \times 10^0 \\ + & 0760 & \times 10^0 \\ \hline & 2084 & \times 10^0 \end{array} \right.$$

$$\begin{array}{rll} & 1324 & \times 10^0 \\ + & 0076 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 0132.4 & \times 10^1 \\ + & 7600 & \times 10^1 \\ \hline & 7732.4 & \times 10^1 \\ \approx & 7732 & \times 10^1 \end{array} \right.$$

where the notation of $(-1)^s \times c \times 10^q$ (c is an integer) appears instead of $(-1)^s \times m \times 10^e$ (m has a single digit to the left of the fractional point). From now, we will in general just use the notation of $(-1)^s \times c \times 10^q$ similar to the way the standard actually encodes the numbers.

The following example

$$\begin{array}{rll} & 9853 & \times 10^7 \\ + & 1466 & \times 10^6 \end{array} \left\{ \begin{array}{rll} & 9853 & \times 10^7 \\ + & 01466 & \times 10^7 \\ \hline & 99996 & \times 10^7 \\ \approx & 1000 & \times 10^8 \end{array} \right.$$

shows that rounding may sometimes cause a carry out of the most significant digit. In such cases, the result must be normalized by a one digit right shift (division by 10) and the exponent increased by one.

Floating point numbers may be positive or negative. Thus, in

$$\begin{array}{rrl} & 1324 & \times 10^3 \\ + & -1321 & \times 10^3 \end{array} \left\{ \begin{array}{rrl} & 1324 & \times 10^3 \\ + & 8679 & \times 10^3 \\ \hline & 0003 & \times 10^3 \end{array} \right.$$

the effective operation is a subtraction conducted with the ten's complement. For such an operation where the result has leading zeros the standard specifies that it will be shifted left (i.e. multiplied by powers of ten) only to accommodate any digits which may be lost due to rounding. In the example at hand no such digits exist. However, in

$$\begin{array}{rrl} & +1004 & \times 10^3 \\ - & +3217 & \times 10^2 \end{array} \left\{ \begin{array}{rrl} & 1004 & \times 10^3 \\ + & 96783 & \times 10^3 \\ \hline & 06823 & \times 10^3 \\ = & 6823 & \times 10^2 \end{array} \right.$$

the result is shifted one digit to the left (multiplied by 10) to save the digit 3 from being rounded away.

The above examples illustrate some of the basic ideas. Interested readers should consult the standard and explore more intricate cases such as when the exponent is near the maximum or minimum values such that it is not possible to increment or decrement it further.

To summarize the basic ideas, we need to perform the following steps for the significand of the result:

1. Find the exponent difference and the leading zeros to decide the necessary alignment.
2. Align the numbers by shifting (multiplying or dividing by powers of ten).
3. Add or subtract the two numbers depending on the *effective* operation.
4. In the case of an effective subtraction and when the exponents are equal, the "significand" of the result may become negative. If so, complement it.
5. In the case of an effective subtraction, find the location of the leading non-zero digit and shift the result to the *left* at most up to this location *if needed*.
6. In the case of an addition with a carry overflow, normalize the result by a shift to the *right*.
7. Use the digits shifted to the right and the least significant digit as well as the sign of the number and the rounding direction to decide on the appropriate rounding action then add this rounding digit.
8. If an overflow due to rounding occurs, renormalize the result.

What about the exponent and sign? The exponent of the result is that of the "larger" number adjusted according to alignment and normalization. Depending on the rounding direction, the case of overflow yields either the maximum number with the maximum exponent or an indication of $\pm\infty$.

The sign is that of the operand assumed to be the largest number and if a complementation occurs the sign is flipped. It is important to note that the rounding direction affects the sign in case of a result exactly equal to zero: RTP $(x - x) = +0$ while RTN $(x - x) = -0$ but RTP $((-0) + (-0)) = -0$.

What are the main hardware pieces? This algorithm has a number of big parts:

- adders: used for exponent difference, significand addition, complementation, rounding;
- big shifters: used for alignment (right) and leading digit normalization (left); and
- multiplexers: used for short normalization shifts, making choices.

In addition to these pieces, some logic blocks calculate the rounding decision, decide the effective operation, invert the smaller operand for an effective subtraction, as well as other minor details.

Most of the ideas just mentioned apply equally to binary floating point addition but what distinguishes decimal is that the significand may have leading zeros and may be represented either as an integer in binary or as decimal digits in an encoding similar to BCD. When the binary encoding is used, designers refer to it as Binary Integer Decimal (BID) when the decimal encoding is used it is called Densely Packed Decimal (DPD). Let us explore these encodings.

1. BID

   The combination field determines the most significant bits of the significand. Those and the trailing significand are treated as one integer in unsigned binary notation.

   - If there is no need for exponent adjustments, then an operation on two such significands is very easy. It may use the existing binary hardware.
   - When exponent adjustments are necessary, the adjustment does not lead to a simple shift of the significand. We must multiply (or divide) by $10^{\text{exp_diff}}$. A number such as 76 is represented as 1001100 in binary while 760 is 1011111000 in binary. The multiplication by ten is a simple shift in decimal but it is a more elaborate operation in binary because the decimal digits are not isolated.
   - Since the "digit boundaries" are not directly available, rounding is not straightforward.

2. DPD

   Each ten bits, called "declet," of the trailing significand encodes three decimal digits. The conversion from a declet to three separate BCD digits and the conversion back is detailed in the standard and is quite easy to implement in few logic gates.

The accessibility of the digit boundaries simplifies

- the alignment of operands via simple shifts by any number of digits,
- the rounding of the result at the exact decimal digit boundary required by the standard,
  and
- the conversion from or to character strings for inspection by humans.

Ten bits can encode 1024 different combinations while three decimal digits encode 1000 different combinations only. The ratio of "wasted" possibilities is 24/1000 which is much smaller than if four bits (16 combinations) are used to encode one decimal digit (10 combinations) where the ratio there is 6/10. The higher encoding density is the reason this encoding is called Densely Packed Decimal.

5.3.2 Specific Designs

Over the first decade of the twenty-first century, many designs were proposed for decimal floating point operations [7]. We will focus our attention here on designs using the conventional paradigm of CMOS digital gates although some alternative technologies [8, 9] were also considered for decimal circuits at different points in time. Commercial hardware designs for decimal floating point additions were pioneered by IBM [10–12], followed by SilMinds [6, 13–15], then Fujitsu [16]. All of these commercial implementations use DPD. Intel [17] introduced a software library using the BID encoding while IBM [18] introduced a software library using DPD. Support of decimal floating point arithmetic in programming languages started with the C language [19] and Fortran followed.

The decimal floating point addition designs for DPD encoding use BCD not binary adders and must handle the leading zeros correctly. Beyond that, they are similar to binary floating point designs. Some early academic research on the implementation of decimal floating point addition was published [20] before the revised standard was even finalized. The first designs were concerned mostly with the provision of correct functionality. Other optimizations such as the merger of rounding with addition [21] or leading zero anticipation [22] or the inclusion of other related operations with addition appeared in later designs [23, 24]. Redundant adders [25] provide a possibility to integrate both binary and decimal operations in one hardware unit which can even detect and correct errors during the computation [26].

To align the significands for addition in numbers encoded in BID [27], we use a multiplier to multiply by $10^{\text{exp_diff}}$. Rounding of BID encoded numbers poses a similar challenge [28, 29]. DPD is easier to add and round but requires "decimal" not binary adders. Another approach for BID addition [30] uses special converters

from and to BID [31, 32] to convert the BID encoding into DPD/BCD, then add using a decimal adder, and finally convert back to BID at the output.

For the multiplication operation, BID significands are simply "binary integers" which simplifies the designs. However, the rounding is still more difficult than DPD as in the case of addition. On the other hand, DPD significand multiplication is harder than BID significand multiplication. Let us explore multiplication in more detail and see why.

5.4 Decimal Multiplication

5.4.1 Floating Point Multiplication

In floating point multiplication there is no operand alignment. In the normal cases, we multiply the significands, add the exponents, and produce a sign bit for the result equal to the *XOR* of the two operand signs. This simplified algorithm needs refinement to handle exceptional conditions which include cases when one of the inputs is special ($\pm\infty$ or NaN for example) or when the exponent of the result exceeds the maximum or minimum bounds.

The step of multiplying the significands is not that simple for DPD encoded numbers [33, 34]:

- The direct use of the multiplier digits in BCD-8421 leads to complex multiples of the multiplicand (3X, 6X, 7X, 9X) which are not easy to obtain via simple shifts of the multiplicand or with a few levels of logic gates.
- A "modified Booth" recoding leads to a digit set $\{-5, \ldots, 5\}$ with multiples 0, X, 2X, 3X, 4X, 5X, and their negatives. Again, the 3X multiple is not easy to obtain.
- Other recodings were proposed such as recoding each digit Y_i of the multiplier into two digits Y_i^U and Y_i^L using, for example
  - $Y_i = 4Y_i^U + Y_i^L$ with $Y_i^U \in \{0, 1, 2\}$ and $Y_i^L \in \{-2, -1, 0, 1, 2\}$ requiring 0, X, 2X, and their negatives as well as 4X and 8X

    or
  - $Y_i = 5Y_i^U + Y_i^L$ with $Y_i^U \in \{0, 1\}$ and $Y_i^L \in \{-2, -1, 0, 1, 2\}$ requiring 0, X, 2X, and their negatives as well as 5X.

Several properties of the decimal codes presented in Table 5.1 help in the generation of the multiplicand multiples [33]. For example, if we have a multi-digit number X where each digit is encoded in 5 2 1 1 and we form another number Y by shifting X one bit position to the left then interpret Y as being encoded in 4 2 2 1 then $Y = 2X$. Y is obviously one digit larger than X. We assume that zero is shifted into the least significant bit and that three extra zeros are padded to fill the most significant digit of Y.

The proof is quite simple. With n digits, $X = \sum_{i=0}^{n-1} x_i 10^i$ where the single digits $x_i = 5x_{i_3} + 2x_{i_2} + x_{i_1} + x_{i_0}$. Similarly, $Y = \sum_{i=0}^{n} y_i 10^i$ where the single digits $y_i = 4y_{i_3} + 2y_{i_2} + 2y_{i_1} + y_{i_0}$. Given that Y is a shifted version of X then $y_i = 4x_{i_2} + 2x_{i_1} + 2x_{i_0} + x_{(i-1)_3} = 2(2x_{i_2} + x_{i_1} + x_{i_0}) + x_{(i-1)_3}$. The full number

$$\begin{aligned} Y &= \sum_{i=0}^{n} \left(2(2x_{i_2} + x_{i_1} + x_{i_0}) + x_{(i-1)_3}\right) 10^i \\ &= \sum_{i=0}^{n} (2(2x_{i_2} + x_{i_1} + x_{i_0}))\, 10^i + 2\left(5x_{(i-1)_3}\right) 10^{i-1} \\ &= \sum_{i=0}^{n} (2(5x_{i_3} + 2x_{i_2} + x_{i_1} + x_{i_0}))\, 10^i \\ &= 2X. \end{aligned}$$

Similarly, a left shift by three bits of digits encoded in 8421 generates 5X with digits encoded in 5421.

Binary multipliers use carry save adders extensively to speed up the partial products summation step in multiplication. However, decimal digits represented in BCD-8421 need corrections when added and hence there is no direct equivalent of carry save adders in BCD-8421. Fortunately, the other decimal codes allow designers to build efficient carry save adders for decimal [33] as in the following example where three digits A, B, and C encoded in 4221 are added using normal binary full adders to produce the sum bits of S and the carry bits in H. The carry bits have a mathematical weight that is twice those of the bits in S. For binary multipliers a simple left shift of the wires by one bit location suffices to achieve the doubling. In decimal, we recode the 4221 to 5211 then shift left by one bit location to achieve a similar effect.

	Decimal Value	*BCD* 4221	
A :	3	0101	Binary
B : +	7	1101	full adders
C : +	8	1110	
S :	4	0110	
$H4221$:	7	1101	
			Recoding *BCD*4221 to *BCD*5211
$H5211$:	7	1100	
			Left Shift
$W = 2 \times H$:	14	1100-	W in *BCD*4221

Result $= S + 2H = S + W = 18$.

5.4.2 Specific Designs

As mentioned earlier, the significand multiplication for BID encoded numbers is a normal binary multiplication. The complications in BID multipliers arise in the rounding where the same problems described in floating point addition exist due to the inability to clearly distinguish the digit boundaries [35, 36].

DPD designs started first by presenting ideas for carry-save addition for decimal fixed point multipliers [37] then iterative high-frequency multipliers [38] and multioperand addition [39]. That early work combined with partial product generation [40] formed the base for the complete floating point multipliers [41, 42] and the innovative ideas [33] relying on the various decimal codes yielding fully parallel floating point multipliers [13, 34].

Due to the large area taken by multipliers, there were attempts to share hardware between binary and decimal and produce a combined unit. That track of work also started by fixed point combined units [43, 44] and then floating point combined units followed [45]. Other researchers focused on improving the speed of parallel multiplication [46] including the use of redundant representations [47, 48].

5.5 Decimal Fused Multiply Add (FMA)

5.5.1 Floating Point FMA

The fused multiply add operation is the calculation of a multiplication and an addition as in $ab + c$ with only one rounding at the end of the whole operation and not as two separate operations with two separate rounding steps. The presence of only one rounding provides a more accurate result. Software may invoke the FMA operation in many calculations such as in the computation of scalar products, matrix multiplications, or polynomial evaluation where there is often an iteration on an instruction such as ($\text{sum} = \text{sum} + a_i b_i$). Making this instruction a single operation that is both *faster* and more *accurate* is beneficial.

The FMA is usually faster than two separate operations because the additional input (after proper alignment) is just another bit vector that can be summed at the end of the partial product reduction tree of the multiplier. The FMA unit can perform regular addition as $a \times 1.0 + c$ and regular multiplication as $ab + 0.0$. Furthermore, if there is no hardware support for the division and square root, then the presence of the FMA instruction speeds the software implementations of those two operations. Extended precision software can also benefit from the presence of FMA to get the "lower" part of a multiplication using: $H = ab + 0.0$, $L = ab - H$. Due to its importance, the FMA became a required operation in the IEEE 754-2008 revised standard.

However, the inclusion of FMA support in processors raises some issues.

- The instruction format must support three inputs and a separate destination. If not, then an instruction such as $c = c + ab$ might be appropriate for most applications.
- The architecture must supply the FMA unit with three inputs which leads to increased wiring and additional register file ports.
- The use of FMA as an adder suffers from the presence of the multiplication hardware, the increased wiring, and the extra control lines which may slow down the normal addition. Similarly for multiplication. Hence, there was a proposal to provide for a more flexible unit that allows faster operations by skipping the unneeded parts [49].

In the FMA, the exponent of the multiplication product ab might be smaller than the exponent of the addend c. Fast implementations do not wait till that product is ready to shift it to the right but rather the addend is left shifted to align it. Hence, the data path in the FMA is even wider than in addition or in multiplication.

5.5.2 Specific Designs

The first decimal FMA proposal was a combined binary/decimal FMA [50]. However, according to its designer [50], that design fails in several cases and passes only 30 test cases which is a tiny number of test vectors.

The first verified [51] decimal FMA was presented in 2010 [14, 52]. Another slightly faster but also verified design [53] followed in 2011. Then designers started to introduce other features such as fast leading zero anticipation [54] or fully IEEE-compliant combined binary/decimal FMA [55].

5.6 Decimal Division

5.6.1 Floating Point Division

The division is defined as

$$a = bq + R$$

where a is the *dividend*, b is the *divisor*, q is the *quotient*, and R is the *remainder*. Similar to multiplication, no operand alignment is needed. In normal cases, we divide the significands, subtract the exponents, and produce a sign bit for the result equal to the *XOR* of the two operand signs. However, the division of the significands is more complicated than addition, subtraction, or multiplication. Furthermore,

special inputs (such as $\pm\infty$, NaN, or division by zero) as well as anomalous conditions such as when the exponent of the result exceeds the bounds must be detected and handled correctly.

Decimal division uses the same approaches used in binary division [56]. The subtractive methods (such as restoring, non-restoring, and SRT) are digit recurrence methods which converge linearly to the result. The multiplicative methods (such as Newton-Raphson and series expansion) iterate on a function which converges quadratically to the result.

A simplified subtractive division in the general case of base β performs three steps in iteration i to produce the quotient digit q_i starting from the most signficant digit and going down using the partial remainder R_{i+1} from the previous iteration:

1. compare $\beta \times R_{i+1}$ with the divisor b and its multiples to find q_i,
2. generate the correct multiple $q_i \times b$, and
3. subtract to get $R_i = \beta \times R_{i+1} - q_i \times b$, decrement i, and continue the iterations.

If we make each of these steps simpler, we get a faster division.

Advanced techniques such as SRT rely on higher radix and use redundancy to simplify comparisons, simplify the generation of $q_i b$, and make the subtraction faster.

On the other hand, the conventional schemes of multiplicative division double the precision each iteration (quadratic convergence) [57]. More elaborate techniques [58] allow even faster conversion. Subtractive methods have simpler operations than multiplicative methods in each iteration but require a number of iterations proportional to the precision of the operands. Hence, multiplicative methods would be faster for large operand precisions such as decimal128.

5.6.2 Specific Designs

Similar to other operations, the first design suggestions appeared before the publication of the revised standard [59]. That early proposal used Newton-Raphson technique and was presented in an expanded and detailed manner later [60]. The proposal for radix-10 digit recurrence divison [61] included the most recent developments in the binary digit recurrence algorithms. Similarly, a decimal SRT division [62] was also proposed. A more recent design using functional iteration is based on an internal redundant FMA [63] and uses a special rounding [64] unit.

5.7 Decimal Square Root and Other Functions

Similar to division, the square root and many elementary functions can be calculated via digit recurrence methods [56]. Multiplicative methods used for the binary square root [65] can also be migrated to decimal.

Decimal square root designs based on Newton-Raphson iterations were proposed [66] early on before the publication of the revised standard then verified [67] designs [68] appeared later.

Some researchers focused on other functions such as log [69, 70], antilog [71], and power [72, 73]. Others presented a larger set of elementary functions calculations [74]. Because transcendental functions yield approximate values and there are no currently known algorithms to produce correctly rounded results for all these functions, it is acceptable to convert the decimal numbers to binary, calculate the transcendental functions using the currently available binary implementations, then convert the result back to decimal [75]. The well-known CORDIC algorithm to calculate many functions was also migrated to decimal [76, 77].

5.8 Verification

As is clear from the discussion thus far, designs for floating point operations include many more details and special cases beyond the designs for integer operations. Both formal verification methods [78, 79] and simulation based verification [80] have been used in different attempts to verify floating point hardware.

For decimal hardware specifically, the FPGen framework to generate targeted test vectors was used for various decimal floating point operations in commercial designs from IBM [81]. The freely available test vectors from Cairo University [82] were successful also in detecting errors in various designs [51, 67, 83] as well as in software libraries.

5.9 Potential Embedded Systems Applications

Current commercial implementations of decimal floating point arithmetic are in servers [10–12, 16] mainly targeting the financial sector. In the future, with the wider adoption of the ideas inherent in smart grids and the Internet-of-Things where many devices may communicate and conduct some financial transactions on behalf of their owners we might see more dedicated hardware for decimal in such embedded systems. Such services may obviously be rendered via a software layer on top of regular binary processors. However, dedicated decimal hardware consumes less energy [6].

The use of hardware for decimal operations instead of software leads to

- a much shorter time to finish the operation (a factor of may be 100–1 or more) and
- no energy consumed in overheads such as fetching and decoding of software instructions.

However, the additional circuits consume static power when idle and burn energy. In general, the use of hardware, for decimal or for other functionalities, is faster and more energy efficient than software. Software, on the other hand, provides the flexibility to change the implementation without changing the circuits. Commercial processor makers prioritize the amount of chip area dedicated to specific functions based on the use frequency and the expected gains from accelerating the related applications.

5.10 Conclusion

This chapter touched on some aspects of decimal floating point arithmetic as defined in the IEEE 754-2008 standard [1]. The need for decimal is illustrated by the simple program code which leads to surprising results from a human perspective. Given that decimal allows the presence of leading zeros and may be encoded in either DPD or BID the migration of ideas from binary floating point designs must handle these intricacies. The number of decimal arithmetic designs is still growing and we may see in the future a wider adoption of combined units where both binary and decimal are integrated to save area. The verification of correct functionality of such combined units will build on the previous work done to verify each system separately.

References

1. IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic* (IEEE, New York, 2008)
2. IEEE Task P754, *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic* (IEEE, New York, 1985)
3. M.F. Cowlishaw, The 'telco' benchmark. World-Wide Web document, IBM Hursley Laboratory, Hursley (2002). http://www.speleotrove.com/decimal/telco.html
4. M.F. Cowlishaw, Decimal floating-point: algorism for computers, in *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: Proceedings: Santiago de Compostela, June 15–18, 2003*, ed. by J.C. Bajard, M. Schulte (IEEE Computer Society Press, Silver Spring, MD, 2003), pp. 104–111
5. W. Buchholz, Fingers or fists? (the choice of decimal or binary representation). Commun. Assoc. Comput. Mach. **2**(12), 3–11 (1959)
6. H.A.H. Fahmy, R. Raafat, A.M. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, Energy and delay improvement via decimal floating point units, in *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, Portland, OR (2009), pp. 221–224
7. L.-K. Wang, M.A. Erle, C. Tsen, E.M. Schwarz, M.J. Schulte, A survey of hardware designs for decimal arithmetic. IBM J. Res. Dev. **54**, 8:1–8:15 (2010)
8. H.A.H. Fahmy, K. Ismail, Analysis of a single-electron decimal adder. Appl. Phys. Lett. **70**, 2613–2615 (1997)
9. M. Gladshtein, Delay-based processing-in-wire for design of QCA serial decimal arithmetic units. ACM J. Emerg. Technol. Comput. Syst. **10**, 13:1–13:18 (2014)

10. L. Eisen, J.W. Ward III, H.-W. Tast, N. Mäding, J. Leenstra, S.M. Mueller, C. Jacobi, J. Preiss, E.M. Schwarz, S.R. Carlough, IBM POWER6 accelerators: VMX and DFU. IBM J. Res. Dev. **51**, 663–683 (2007)
11. E.M. Schwarz, J.S. Kapernick, M.F. Cowlishaw, Decimal floating-point support on the IBM System z10 processor. IBM J. Res. Dev. **53**, 4:1–4:10 (2009)
12. S. Carlough, A. Collura, S. Mueller, M. Kroener, The IBM zEnterprise-196 decimal floating-point accelerator, in Schwarz and Oklobdzija [84], pp. 139–146
13. R. Raafat, A. Mohamed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, H.A.H. Fahmy, A decimal fully parallel and pipelined floating point multiplier, in *Forty-Second Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2008)
14. R. Samy, H.A.H. Fahmy, R. Raafat, A. Mohamed, T. ElDeeb, Y. Farouk, A decimal floating-point fused-multiply-add unit, in *Fifty-Third MidWest Symposium on Circuits and Systems, (MWSCAS)*, Seattle, Washington (2010)
15. H.A.H. Fahmy, T. ElDeeb, M. Hassan, Y. Farouk, R. Eissa, Decimal floating point for future processors, in *The 22nd International Conference on Microelectronics, ICM 2010* (2010)
16. T. Yoshida, T. Maruyama, Y. Akizuki, R. Kan, N. Kiyota, K. Ikenishi, S. Itou, T. Watahiki, H. Okano, Sparc64 x: Fujitsu's new-generation 16-core processor for unix servers. IEEE Micro **33**, 16–24 (2013)
17. M. Cornea, J. Harrison, C. Anderson, P. Tang, E. Schneider, E. Gvozdev, A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. IEEE Trans. Comput. **58**, 148–162 (2009)
18. M. Cowlishaw, *The decNumber C Library* (IBM Corporation, San Jose, CA, 2006). Version 3.37
19. International Organization for Standardization, ISO/IEC TR 24732:2009 Information technology — Programming languages, their environments and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic. Technical Report, International Organization for Standardization, Geneva, 2009
20. J. Thompson, N. Karra, M.J. Schulte, A 64-bit decimal floating-point adder, in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 19–20 February 2004 (IEEE Computer Society Press, Silver Spring, MD, 2004), pp. 297–298
21. Á. Vázquez, E. Antelo, A high-performance significand BCD adder with IEEE 754-2008 decimal rounding, in Bruguera et al. [85], pp. 135–144
22. M.H. Amin, A.M. ElTantawy, H.A.H. Fahmy, A. Khedr, Efficient decimal leading zero anticipator designs, in *Forty-Fifth Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2011)
23. L.-K. Wang, M.J. Schulte, A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator, in Bruguera et al. [85], pp. 125–134
24. L.-K. Wang, M.J. Schulte, J.D. Thompson, N. Jairam, Hardware designs for decimal floating-point addition and related operations. IEEE Trans. Comput. **58**, 322–335 (2009)
25. K. Yehia, H.A.H. Fahmy, M. Hassan, A redundant decimal floating-point adder, in *Forty-Fourth Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2010)
26. S.Y. Elsayed, H.A.H. Fahmy, M.S. Khairy, Residue codes for error correction in a combined decimal/binary redundant floating point adder, in *Forty-Sixth Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2012)
27. C. Tsen, S. Gonzalez-Navarro, M. Schulte, Hardware design of a binary integer decimal-based floating-point adder, in *25th International Conference on Computer Design, 2007 (ICCD 2007)*, 7–10 October 2007, Resort at Squaw Creek, Lake Tahoe, CA, ed. by IEEE (IEEE Computer Society Press, Silver Spring, MD, 2007), pp. 288–295
28. C. Tsen, M. Schulte, S. Gonzalez-Navarro, Hardware design of a binary integer decimal-based IEEE P754 rounding unit, in *ASAP 07: Conference Proceedings: IEEE 18th International Conference on Application-Specific Systems, Architectures, and Processors: Montréal: July 8–11, 2007*, ed. by IEEE (IEEE Computer Society Press, Silver Spring, MD, 2007), pp. 115–121
29. C. Tsen, S. Gonzalez-Navarro, M.J. Schulte, K. Compton, Hardware designs for binary integer decimal-based rounding. IEEE Trans. Comput. **60**, 614–627 (2011)

30. A.A.M. Ayoub, Binary integer decimal-based floating point adder. Master's thesis, Electronics and Electrical Communications Department, Cairo University, Giza (2014). Available at http://eece.cu.edu.eg/~hfahmy/thesis/2014_03_dpdbid.pdf
31. A.A. Ayoub, H.A.H. Fahmy, BID to BCD/DPD converters. US Patent number 9134958 (2015)
32. A.A. Ayoub, H.A.H. Fahmy, T. Eldeeb, DPD/BCD to BID converters. US Patent number 9143159 (2015)
33. A. Vazquez, E. Antelo, P. Montuschi, A new family of high-performance parallel decimal multipliers, in Kornerup and Muller [86], pp. 195–204
34. A. Vazquez, E. Antelo, P. Montuschi, Improved design of high-performance parallel decimal multipliers. IEEE Trans. Comput. **59**, 679–693 (2010)
35. S. Gonzalez-Navarro, C. Tsen, M. Schulte, Binary integer decimal-based multiplier for decimal floating-point arithmetic, in *Conference Record of the Forty-first Asilomar Conference on Signals, Systems and Computers: November 4–7, 2007, Pacific Grove, CA. ACSSC 2007*, ed. by M.B. Matthews (IEEE Computer Society Press, Silver Spring, MD, 2008), pp. 353–357
36. S. Gonzalez-Navarro, C. Tsen, M.J. Schulte, Binary integer decimal-based floating-point multiplication. IEEE Trans. Comput. **62**, 1460–1466 (2013)
37. M.A. Erle, M.J. Schulte, Decimal multiplication via carry-save addition, in *IEEE International Conference on Application-Specific Systems, Architectures and Processors: Proceedings: ASAP 2003*, 24–26 June, 2003, The Hague, ed. by E.F. Deprettere (IEEE Computer Society Press, Silver Spring, MD, 2003), pp. 348–358. IEEE Computer Society Order Number PR01992
38. R.D. Kenney, M.J. Schulte, M.A. Erle, A high-frequency decimal multiplier, in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings*, 11–13 October 2004, ed. by IEEE (IEEE Computer Society Press, Silver Spring, MD, 2004), pp. 26–29
39. R.D. Kenney, M.J. Schulte, M.A. Erle, Multioperand decimal addition, in *Emerging Trends in VLSI Systems Design: Proceedings: IEEE Computer Society Annual Symposium on VLSI*, 19–20 February 2004, Lafayette, LA [ISVLSI 2004], ed. by A. Smailagic, M.A. Bayoumi (IEEE Computer Society Press, Silver Spring, MD, 2004), pp. 251–253. IEEE Computer Society order number P2097
40. M. Erle, E. Schwarz, M. Schulte, Decimal multiplication with efficient partial product generation, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ARITH-17 2005*, June 27–29, 2005, Cape Cod, MA, ed. by P. Montuschi, E.E.M. Schwarz (IEEE Computer Society Press, Silver Spring, MD, 2005), pp. 21–28
41. M.A. Erle, M.J. Schulte, B.J. Hickmann, Decimal floating-point multiplication via carry-save addition, in Kornerup and Muller [86], pp. 46–55
42. M.A. Erle, B.J. Hickmann, M.J. Schulte, Decimal floating-point multiplication. IEEE Trans. Comput. **58**, 902–916 (2009)
43. L. Dadda, Multioperand parallel decimal adder: a mixed binary and BCD approach. IEEE Trans. Comput. **56**, 1320–1328 (2007)
44. M. Mahmoud, H.A.H. Fahmy, A parallel combined binary/decimal fixed-point multiplier with binary partial products reduction tree, in *The 21st International Conference on Computer Theory and Applications (ICCTA)*, Alexandria (2011)
45. C. Tsen, S. Gonzalez-Navarro, M.J. Schulte, B. Hickmann, K. Compton, A combined decimal and binary floating-point multiplier, in *20th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2009)*, Boston, MA, 7–9 July 2009, ed. by IEEE (IEEE Computer Society Press, Silver Spring, MD, 2009), pp. 8–15
46. G. Jaberipur, A. Kaivani, Improving the speed of parallel decimal multiplication. IEEE Trans. Comput. **58**, 1539–1552 (2009)
47. L. Han, S.-B. Ko, High-speed parallel decimal multiplication with redundant internal encodings. IEEE Trans. Comput. **62**, 956–968 (2013). See comment [48]
48. S. Gorgin, G. Jaberipur, Comment on High-speed parallel decimal multiplication with redundant internal encodings. IEEE Trans. Comput. **64**, 293–294 (2015). See [47]

49. D.R. Lutz, Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines, in Schwarz and Oklobdzija [84], pp. 123–128
50. P.K. Monsson, Combined binary and decimal floating-point unit. Master's thesis, Technical University of Denmark (2008)
51. A. Sayed-Ahmed, H.A.H. Fahmy, R. Samy, Verification of decimal floating-point fused-multiply-add operation, in *The Ninth ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, Sharm El-Sheikh (2011)
52. R. Samy, H.A.H. Fahmy, T. Eldeeb, R. Raafat, Y. Farouk, M. Elkhouly, A. Mohamed, Decimal floating-point fused multiply-add unit (2014). US Patent number 8694572
53. A.M. ElTantawy, Decimal floating point arithmetic unit based on a fused multiply add module. Master's thesis, Electronics and Electrical Communications Department, Cairo University, Giza (2011). Available at http://eece.cu.edu.eg/~hfahmy/thesis/2011_08_dfma.pdf
54. A. Akkas, M.J. Schulte, A decimal floating-point fused multiply-add unit with a novel decimal leading-zero anticipator, in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)* (2011), pp. 43–50
55. A.A.A. Wahba, Fully IEEE-compliant binary/decimal unit based on a binary/decimal fma. Master's thesis, Electronics and Electrical Communications Department, Cairo University, Giza (2014). Available at http://eece.cu.edu.eg/~hfahmy/thesis/2014_06_bd_fma.pdf
56. M.D. Ercegovac, T. Lang, *Digital Arithmetic* (Morgan Kaufmann, Los Altos, CA, 2004)
57. M.J. Flynn, On division by functional iteration. IEEE Trans. Comput. **C-19**, 702–706 (1970). Reprinted in [87]
58. P. Rabinowitz, Multiple-precision division. Commun. Assoc. Comput. Mach. **4**, 98–98 (1961)
59. L.-K. Wang, M.J. Schulte, Decimal floating-point division using Newton–Raphson iteration, in *Proceedings of 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 27–29 September 2004 (IEEE Computer Society Press, Silver Spring, MD, 2004), pp. 84–95
60. L.-K. Wang, M.J. Schulte, A decimal floating-point divider using Newton–Raphson iteration. J. VLSI Sig. Proc. **49**, 3–18 (2007)
61. T. Lang, A. Nannarelli, A radix-10 digit-recurrence division unit: algorithm and architecture. IEEE Trans. Comput. **56**, 727–739 (2007)
62. H. Nikmehr, B. Phillips, C.-C. Lim, Fast decimal floating-point division. IEEE Trans. Very Large Scale Integr. VLSI Syst. **14**, 951–961 (2006)
63. A. Mohamed, R. Raafat, H.A.H. Fahmy, T. Eldeeb, Y. Farouk, R. Samy, M. Elkhouly, Parallel redundant decimal fused-multiply-add circuit (2014). US Patent number 8805917
64. A. Mohamed, H.A.H. Fahmy, R. Raafat, Y. Farouk, M. Elkhouly, R. Samy, T. Eldeeb, Rounding unit for decimal floating-point division (2014). US Patent number 8751555
65. C.V. Ramamoorthy, J.R. Goodman, K.H. Kim, Some properties of iterative square-rooting methods using high-speed multiplication. IEEE Trans. Comput. **C-21**, 837–847 (1972)
66. L.-K. Wang, M.J. Schulte, Decimal floating-point square root using Newton–Raphson iteration, in *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors: ASAP 2005* , 23–25 July 2005, Samos, ed. by S. Vassiliadis, N.J. Dimopoulos, S.V. Rajopadhye (IEEE Computer Society Press, Silver Spring, MD, 2005), pp. 309–315
67. A.S. Ahmed, H.A.H. Fahmy, U. Kuehne, Verification of the decimal floating-point square root operation, in *19th IEEE European Test Symposium (ETS)*, Paderborn (2014)
68. R. Raafat, A. Mohamed, H.A.H. Fahmy, Y. Farouk, M. Elkhouly, T. Eldeeb, R. Samy, Decimal floating-point square-root unit using Newton-Raphson iterations (2014). US Patent number 8812575
69. D. Chen, Y. Zhang, Y. Choi, M.H. Lee, S.-B. Ko, A 32-bit decimal floating-point logarithmic converter, in Bruguera et al. [85], pp. 195–203
70. D. Chen, L. Han, Y. Choi, S. Ko, Improved decimal floating-point logarithmic converter based on selection by rounding. IEEE Trans. Comput. **61**, 607–621 (2012)

71. D. Chen, Y. Zhang, D. Teng, K. Wahid, M.H. Lee, S.-B. Ko, A new decimal antilogarithmic converter, in *2009 IEEE International Symposium on Circuits and Systems: Circuits and Systems for Human Centric Smart Living Technologies, Conference Program*, Taipei International Convention Center, Taipei, May 24–May 27, 2009, ed. by IEEE (IEEE Computer Society Press, Silver Spring, MD, 2009), pp. 445–448
72. M. Hassan, T. ElDeeb, H.A.H. Fahmy, Algorithm and architecture for on-line decimal powering computation, in *Forty-Fourth Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2010)
73. W.S. Sayed, H.A.H. Fahmy, What are the correct results for the special values of the operands of the power operation? Trans. Math. Softw. **42**(2), 14:1–14:17 (2016) doi:10.1145/2809783
74. T. ElDeeb, H.A.H. Fahmy, M.Y. Hassan, Decimal elementary functions computation. (2014). US Patent number 8788560
75. J. Harrison, Decimal transcendentals via binary, in Bruguera et al. [85], pp. 187–194
76. Á. Vázquez, J. Villalba, E. Antelo, Computation of decimal transcendental functions using the CORDIC algorithm, in Bruguera et al. [85], pp. 179–186
77. A. Kaivani, G. Jaberipur, Decimal CORDIC rotation based on selection by rounding: algorithm and architecture. Comput. J. **54**, 1798–1809 (2011)
78. C. Kern, M.R. Greenstreet, Formal verification in hardware design: a survey. ACM Trans. Des. Autom. Electron. Syst. **4**, 123–193 (1999)
79. D.M. Russinoff, A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7^{TM} processor. LMS J. Comput. Math. **1**, 148–200 (1998)
80. E. Guralnik, M. Aharoni, A.J. Birnbaum, A. Koyfman, Simulation-based verification of floating-point division. IEEE Trans. Comput. **60**, 176–188 (2011)
81. M. Aharoni, R. Maharik, A. Ziv, Solving constraints on the intermediate result of decimal floating-point operations, in Kornerup and Muller [86], pp. 38–45
82. C. University, Arithmetic operations debugging and verification (2012). Available at http://eece.cu.edu.eg/~hossam/arith_debug/
83. A. Sayed-Ahmed, H.A.H. Fahmy, M. Hassan, Three engines to solve verification constraints of decimal floating-point operations, in *Forty-Fourth Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA (2010)
84. E. Schwarz, V.G. Oklobdzija (eds.), *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, July 25–27, 2011, Tübingen (IEEE Computer Society Press, Silver Spring, MD, 2011)
85. J.D. Bruguera, M. Cornea, D. DasSarma, J. Harrison (eds.), *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, June 8–10, 2009, Portland, OR (IEEE Computer Society Press, Silver Spring, MD, 2009)
86. P. Kornerup, J.-M. Muller (eds.), *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, June 25–27, 2007, Montpellier (IEEE Computer Society Press, Silver Spring, MD, 2007)
87. E.E. Swartzlander Jr., *Computer Arithmetic*. IEEE Computer Society Press Tutorial, vol. 1 (IEEE Computer Society Press, Silver Spring, MD, 1990)

Chapter 6
Design and Evaluation of Booth-Encoded Multipliers in Redundant Binary Representation

Yajuan He, Jiaxing Yang, and Chip-Hong Chang

6.1 Introduction

The state-of-the-art digital signal processing applications play an important role in making the complex real-time algorithms for speech, audio, image processing, video, control and communication systems economically feasible [1–4]. Multiplication is one of the most commonly used arithmetic operators in these application-specific datapaths. Comparing with many other arithmetic operations, multiplication is time consuming and power hungry. The critical paths dominated by digital multipliers often impose speed limits on the entire design. Therefore, there have been an immense volume of publications and endless research interest in the design of energy efficient digital multipliers at different design abstraction levels [5–8].

Most digital multiplier designs are based on the two's complement arithmetic, which is referred to as normal binary (NB) arithmetic as opposed to the redundant binary (RB) arithmetic studied in this chapter. Fast NB multipliers use modified Booth encoders, and 3-to-2 counters or 4-to-2 compressors in a tree structure for parallel computation [9–11]. In the last three decades, most speed improvements in this architecture have been achieved via extreme circuit optimization and the use of advanced fabrication technology. The gain resulting from architectural innovation is almost stagnant. It is conjecture that new insight into energy efficient multiplier design is likely to be derived from an alternative arithmetic with a different

Y. He (✉) • J. Yang
School of Microelectronics and Solid Electronics, University of Electronic Science and Technology of China, Chengdu, Sichuan 610054, China
e-mail: yjhe@uestc.edu.cn; jxyang@std.uestc.edu.cn

C.-H. Chang
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798, Singapore
e-mail: echchang@ntu.edu.sg

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_6

architectural optimization outlook. In view of this, alternative RB number based architectures with the merit of carry propagation free accumulation are sought to speed up digital multiplication [12–19]. The idea is to apply a simple signed digit representation as an internal format for the addition of multiple operands. The redundancy is exploited to speed up the addition of partial products, which is a crucial stage of the digital multiplier architecture. Furthermore, the use of redundant binary adders (RBAs) makes a more regular interconnection network and modular partial product summing tree structure. Recently, an RB multiply-and-accumulate (MAC) unit has also shown to exhibit high-performance and energy efficiency for error-resilient applications due to its elimination of carry propagation chain [20]. Advocators are optimistic that the inherent carry-free addition and structural regularity of RB multiplier architecture offer significant room for both power and latency reduction.

Unfortunately, there are some fallacies in certain success acclamations in the RB multiplier realm. The most critical misconception is the existence of carry-free conversion from an RB representation to an NB representation. As a matter of fact, for most RB multipliers, the critical path includes the RB-to-NB conversion. The exploitation of RB arithmetic for efficient implementation of digital multiplier is therefore not well-elucidated without a proper evaluation of the design trade-off among its constituent building blocks. As the trajectory of an RB multiplier in the area-time space is a strong function of the ways the partial products are generated and how they are encoded, a structural approach such as that used in analyzing the performance of one-bit CMOS full adder cells [21] can provide a good insight into the trade-off and limitation of RB arithmetic, and eradicate some fallacious use of RB arithmetic on digital multiplier design. There are many potential compositions of the fabrics that made up different RB multiplier circuits. These fabrics are characterized by the radix and type of Booth encoders and decoders, as well as the coding format used for the RB partial product representation, addition, and conversion. A multitude of architectural translations exist for each building block but not all of them are compatible. What has been lacking at present is the understanding of the extent of influences on different VLSI performance factors of one module to its concomitant module upon their integration. The aim of this chapter is to provide a systematic analysis of different circuit modules for each building block that have potential to form a high-performance RB multiplier. The advantage of this anatomy is that it facilitates speciation of RB multipliers from judicious topological combinations of different circuit modules.

The remainder of this chapter is organized as follows. Sect. 6.2 presents the background information pertaining to the Booth multiplier design with RB representation. The base architecture of the RB multiplier focused in this research is outlined along with a comprehensive study of each of its building blocks. The feasibility of adapting different RB coding schemes to a generalized reverse conversion algorithm is discussed in Sect. 6.3. In Sect. 6.4, twenty-one $N \times N$-bit RB multiplier architectures are constructed with different configurations of partial product reduction and RB coding methods for design space exploration. The performances of these designs are analyzed and discussed. Finally, the concluding statements from these analyses are provided in Sect. 6.5.

6.2 Booth Multiplier Design in Redundant Binary Representation

6.2.1 Redundant Binary Arithmetic and Carry-Free Adding Rule

An RB number is a subset of a more generalized set of numbers known as signed digit number representation, which was first introduced by Avizienis [22] in 1961 for fast parallel arithmetic. It did not receive much attention then until the early 1980s when Takagi [23] applied this unconventional arithmetic to fast multiplication and Edamatsu et al. [2] implemented it in VLSI circuit.

In RB representation, the decimal value of an n-digit RB number, $D = (d_{n-1}d_{n-2}\ldots d_1d_0)$, is given by $D = \sum_{i=0}^{n-1} 2^i \times d_i$, where $d_i \in \{1,0,-1\}$. To implement RB arithmetic with standard logic elements, an RB number needs to be encoded into an NB bit stream. According to the different mapping methods, there are three representative coding formats in RB number representation as illustrated in Table 6.1. Table 6.1a shows the sign-magnitude coding for an RB digit: the bit on the right, d_i^a, represents the magnitude of the signed digit, which is either "0" or "1," whereas the bit on the left, d_i^s, indicates its sign, which is either "+" or "−". This coding format is denoted as a dibit (d_i^s, d_i^a). The signed digit d_i can be expressed as:

$$d_i = (-1)^{d_i^s} \times d_i^a \tag{6.1}$$

Another representation for an RB number is the positive–negative coding as shown in Table 6.1b. The value of the digit d_i is equal to the subtraction of the right dibit d_i^- from the left dibit d_i^+, as indicated in (6.2).

$$d_i = d_i^+ - d_i^- \tag{6.2}$$

Table 6.1 RB representations of a signed digit D with different coding methods

(a) Sign-magnitude coding [14]		(b) Positive–negative coding [12]		(c) Positive–negative-complement coding [13]	
Coding (d_i^s, d_i^a)	D	Coding (d_i^+, d_i^-)	D	Coding (d_i^+, d_i^-)	D
(0, 0)	0	(0, 0)	0	(0, 0)	$\overline{1}$
(0, 1)	1	(0, 1)	$\overline{1}$	(0, 1)	0
(1, 0)	$\overline{0} \equiv 0$	(1, 0)	1	(1, 0)	0
(1, 1)	$\overline{1}$	(1, 1)	0	(1, 1)	1

Table 6.2 Carry-free adding rule

a_i	b_i	$a_{i\text{-}1}b_{i\text{-}1}$	$h_{i\text{-}1}$	c_i	s_i
0	0	Don't care	Any	0	0
$\overline{1}$	1			0	0
1	$\overline{1}$			0	0
0	$\overline{1}$	Both are non-negative	0	0	$\overline{1}$
$\overline{1}$	0	Otherwise	1	$\overline{1}$	1
0	1	Both are non-negative	0	1	$\overline{1}$
1	0	Otherwise	1	0	1
1	1	Don't care	Any	1	0
$\overline{1}$	$\overline{1}$			$\overline{1}$	0

Table 6.1c shows the third possible representation, the positive–negative-complement coding, of an RB digit. In this representation, the relationship between the values of the digit d_i and its dibit, d_i^+ and d_i^-, is given by (6.3).

$$d_i = d_i^+ - \overline{d_i^-} \tag{6.3}$$

Due to the redundancy in the binary encoding of a signed digit, the addition of two RB numbers could be made free of carry propagation. Thus, a set of carry-free adding rules are developed to guarantee that the carry-out signal of an RB full adder is independent of the actual carry-in signal from the lower order digit.

Suppose a_i and b_i are two input operands to an RB full adder, there are altogether nine possible combinations of them as shown in Table 6.2. The adder generates an intermediate sum s_i and an intermediate carry c_i before it outputs the final sum z_i. The carry-free adding rules for the addition of two RB numbers can be summarized as follows. Consider the i-th RB full adder that adds the i-th digits, a_i and b_i, from two RB numbers. It receives h_{i-1} from the $(i-1)$-th RB full adder, which is "0" if both inputs to the $(i-1)$-th RB full adder are non-negative and "1" otherwise. This information is used advantageously by the RB full adder to generate an intermediate sum digit s_i and an intermediate carry digit c_i to avoid the propagation of carry. To eliminate the propagation of the possible carry-in of "1," an intermediate sum of "$\overline{1}$" is generated and a carry-out of "1" is created to compensate for the required sum of "1". The final sum z_i is obtained by adding the current immediate sum s_i and the immediate carry c_{i-1} from the $(i-1)$-th RB full adder. As the carry c_i is independent of c_{i-1}, the addition is carry-free. Therefore, all RBAs are to be designed with this adding rule in order to make the addition free of carry propagation.

6.2.2 Booth Algorithms in Redundant Binary Multiplier

In Booth multiplier, one of the two operands of the multiplication is signed digit encoded. The operand that is Booth encoded is called the multiplier and the other operand is called the multiplicand. The Booth algorithm represents a simple

Table 6.3 Booth-3 encoding

Normal binary bits $b_{3i+2}b_{3i+1}b_{3i}b_{3i-1}$	Multiple	Normal binary bits $b_{3i+2}b_{3i+1}b_{3i}b_{3i-1}$	Multiple
000(0)	$+0$	100(0)	$-4M$
000(1)	$+M$	100(1)	$-3M$[a]
001(0)	$+M$	101(0)	$-3M$[a]
001(1)	$+2M$	101(1)	$-2M$
010(0)	$+2M$	110(0)	$-2M$
010(1)	$+3M$[a]	110(1)	$-M$
011(0)	$+3M$[a]	111(0)	$-M$
011(1)	$+4M$	111(1)	-0

[a]Hard multiples

and efficient way to reduce the number of summands required to produce the multiplication result. In radix-r Booth-k encoding ($r = 2^k$), a signed digit d_i is generated from k adjacent multiplier bits $b_{ki+k-1}b_{ki+k-2}b_{ki+1}b_{ki}$ and a borrow bit b_{ki-1} as follows:

$$d_i = -2^{k-1}b_{ki+k-1} + \sum_{j=0}^{k-2} 2^j b_{ki+j} + b_{ki-1} \text{ for } i = 0, 1, \ldots, \left\lceil \frac{N}{k} \right\rceil - 1 \tag{6.4}$$

where k is a positive integer, $\lceil \alpha \rceil$ denotes the smallest integer value larger than or equal to α, N is the word length of the NB number B, and $b_{-1} = 0$. Table 6.3 shows the Booth-3 encoded digits and their corresponding binary bits with the overlapping bit in bracket. A multiple is defined as the product of the Booth-encoded digit d_i and the multiplicand M.

As the radix value, $r = 2^k$, of Booth-k (for positive integer k) encoded multiplier increases, the number of partial products decreases reciprocally with k to $1/k$. Intuitively, it is tempting to select the radix of Booth encoding algorithm as high as possible to reduce as many partial products as possible for the fastest multiplier. However, a close examination reveals that the number of multiples increases commensurately with the radix to $2k + 1$. Besides, the number of hard multiples, which are not the power-of-two factors of the multiplicand, also increases. For example, in Booth-3 encoding, there are two hard multiples, $\pm 3M$, out of a total of 9 distinct multiples. These hard multiples cannot be obtained by simple shifting and/or complementation operations on the multiplicand. Additional carry propagate adders (CPAs) are required to generate them. These CPAs increase the latency of the multiplier because the generation of partial products will not be accomplished until all these hard multiples are produced. Therefore, the advantage of Booth-3 and higher radix Booth encodings has been somewhat compromised due to the long delay and complex decoding logic required for the generation of hard multiples.

To speed up the generation of hard multiples in high-radix Booth encoding, a partially redundant biased Booth encoding algorithm was proposed in [24]. Figure 6.1 depicts the generation and negation of the hard multiple 3*M*. It is generated in a partially redundant form by using a series of small length adders (4-bit). The carry bits of each small length adder are not propagated but brought forward to the partial product summing tree. However, when the 3*M* multiple is negated, both the sum and carry vectors need to be complemented and a "1" is added at their LSB positions. Therefore, the long strings of zeros between carries become strings of ones in the negative multiple. A properly selected biasing constant is introduced to revert the strings of ones back to strings of zeros. The "1"s can be combined with the carry and sum bits to form a single compensation vector.

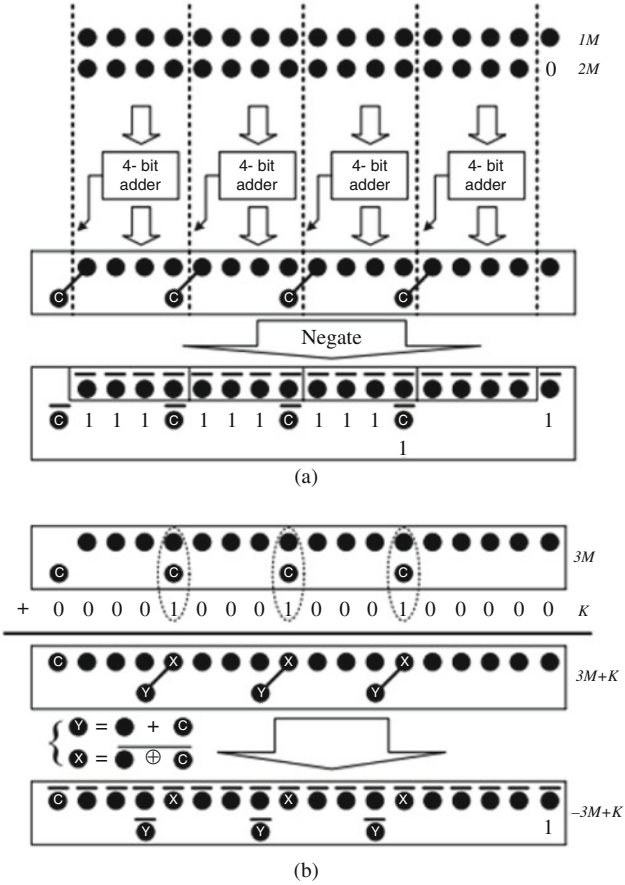

Fig. 6.1 3M hard multiple generation and negation in partially redundant biased Booth encoding scheme (**a**) Negating a number in partially redundant form (**b**) Transforming the simple redundant form with a biasing constant

Table 6.4 RB Booth-4 encoding

Normal binary $b_{4i+3}b_{4i+2}b_{4i+1}b_{4i}(b_{4i-1})$	Multiple		Normal binary $b_{4i+3}b_{4i+2}b_{4i+1}b_{4i}(b_{4i-1})$	Multiple	
	$+M$	$-M$		$+M$	$-M$
0 0 0 0 (0)	0	0	1 0 0 0 (0)	0	$8M$
0 0 0 0 (1)	M	0	1 0 0 0 (1)	M	$8M$
0 0 0 1 (0)	M	0	1 0 0 1 (0)	M	$8M$
0 0 0 1 (1)	$2M$	0	1 0 0 1 (1)	$2M$	$8M$
0 0 1 0 (0)	$2M$	0	1 0 1 0 (0)	$2M$	$8M$
0 0 1 0 (1)	$4M$	M	1 0 1 0 (1)	0	$5M$[a]
0 0 1 1 (0)	$4M$	M	1 0 1 1 (0)	0	$5M$[a]
0 0 1 1 (1)	$4M$	0	1 0 1 1 (1)	0	$4M$
0 1 0 0 (0)	$4M$	0	1 1 0 0 (0)	0	$4M$
0 1 0 0 (1)	$5M$[a]	0	1 1 0 0 (1)	M	$4M$
0 1 0 1 (0)	$5M$[a]	0	1 1 0 1 (0)	M	$4M$
0 1 0 1 (1)	$8M$	$2M$	1 1 0 1 (1)	0	$2M$
0 1 1 0 (0)	$8M$	$2M$	1 1 1 0 (0)	0	$2M$
0 1 1 0 (1)	$8M$	M	1 1 1 0 (1)	0	M
0 1 1 1 (0)	$8M$	M	1 1 1 1 (0)	0	M
0 1 1 1 (1)	$8M$	0	1 1 1 1 (1)	0	0

[a]Hard multiples

The biasing constant of each such partial product introduces an extra compensation vector to the partial product summing tree.

The problem of generating hard multiples in high-radix Booth encoding was also addressed by Besli and Deshmukh [25]. They noticed that some multiples can be obtained by subtracting one simple multiple from another, where a simple multiple refers to one that can be expressed as a power-of-two factor of the multiplicand. The partial products generated in this manner are in congruence with the format of the positive–negative RB coding. This idea has led to a different Booth encoding logic, called the RB Booth encoding (RBBE). Table 6.4 shows the RB Booth-4 encoding, where the original hard multiples $\pm 3M$ are replaced by $\pm(4M - M)$. Among the four hard multiples in the original Booth-4 encoding, $3M$, $6M$, and $7M$ are easily obtained by the subtraction of two simple multiples. The only exception is the hard multiple $5M$ (marked by "a" in Table 6.4), which cannot be generated in this manner. Therefore, additional hardware is necessary to generate the $5M$ multiple. A simple RBA is suggested in [25] to add $4X$ and X, as shown in Fig. 6.2. Fortunately, this RB addition is carry-free and it does not lie in the critical path of the RB Booth encoder and RB partial product generator circuit.

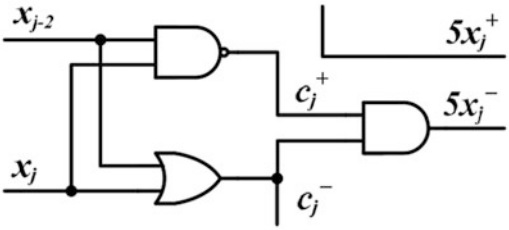

Fig. 6.2 RBA for 5M hard multiple generation

6.2.3 Redundant Binary Coding Interface Components

The use of RB number for digital multiplication is anomalous, or at least incompatible with the data transfer format through standard peripheral interfaces. The two input operands are, by default, assumed to be in two's complement form. Since the partial products generated are NB numbers, to accumulate the partial products through an RBA, they should be converted to RB partial products using an NB-to-RB converter.

In the partial products array of an RB multiplication, suppose $A = (a_{n-1}a_{n-2} \ldots a_0)_2$ and $B = (b_{n-1}b_{n-2} \ldots b_0)_2$ are two n-bit NB partial products. Their summation can be combined into a single n-digit RB number R by:

$$R = A + B = A - (-B) \tag{6.5}$$

Substituting $-B = \overline{B} + 1$ into (6.5) gives:

$$R = A - \left(\overline{B} + 1\right) = A - \overline{B} - 1 = \left[-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a^i\right] - \left[-2^{n-1}\overline{b_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{b_i}\right] - 1 = -2^{n-1}\left(a_{n-1} - \overline{b_{n-1}}\right) + \sum_{i=0}^{n-2} 2^i \left(a_i - \overline{b_i}\right) - 1 \tag{6.6}$$

As an example, consider the positive–negative-complement coding. According to (6.3), an RB digit r can be encoded by $r = r^+ - \overline{r^-}$, where $r^+, r^- \in (0,1)$ and $r \in (0,1,\overline{1})$. Therefore, the terms $a_i - \overline{b_i}$ in (6.6) can be encoded as $r_i = (a_i, b_i)$. To eliminate the hardware required for sign extension, the most significant digit (MSD) can be simply negated to $-(a_{n-1}, b_{n-1})$. From (6.3), it is noted that

$$-\left(r^+, r^-\right) = -r^+ + \overline{r^-} = \left(\overline{r^-}, \overline{r^+}\right) \tag{6.7}$$

This means that the positive–negative-complement coding is symmetric, i.e., r^+ and r^- are commutative and $\left(\overline{r^-}, \overline{r^+}\right) \equiv \left(\overline{r^+}, \overline{r^-}\right)$. Therefore, R can be encoded as follows:

$$R = \left(\overline{a_{n-1}}, \overline{b_{n-1}}\right)(a_{n-2}, b_{n-2}) \ldots (a_1, b_1)(a_0, b_0) + (0, 0) \tag{6.8}$$

This method of generating an RB partial product from two adjacent NB partial products is adopted in the RB multiplier of [12, 13]. It is simple and straightforward, as it involves only the re-wirings and change of the most significant bit (MSB). Thus, the time required by this conversion is independent of the operand length. Equation (6.8) also implies that every RB partial product row so composed requires one correction constant, $(0, 0) = \overline{1}$ to be added by an RBA at its LSB position. All the correction constants generated from the RB partial products, together with those constants from the negative multiples, can be accumulated to form a new RB partial product, called the RB correction vector.

On the other hand, to communicate the result to standard peripheral devices, the final product expressed in RB format also needs to be converted back to the NB number through an RB-to-NB converter. Let F be the n-digit final partial product, then $F = (F^+, F^-)$, where F^+ and F^- are the two binary numbers in two's complement form decomposed from the RB number F. If Z represents the final result in NB representation, the reverse conversion from F to Z can be derived from the basic RB number definition of (6.3):

$$Z = F^+ - \overline{F^-} = F^+ + F^- + 1 \tag{6.9}$$

This implies that the reverse converter can be implemented in a straightforward way by a two operand CPA with a carry-in of "1" fed to the least significant bit (LSB). Let (f_i^+, f_i^-) denote a digit of the final RB partial product (F^+, F^-), and c_{i-1} denote the carry-in from the next lower order digit, then the sum output z_i of the partial products and the carry-out signal c_i can be derived from (6.9) as follows:

$$\begin{cases} z_i = f_i^+ \oplus f_i^- \oplus c_{i-1} \\ c_i = f_i^+ \cdot f_i^- + f_i^+ \cdot c_{i-1} + f_i^- \cdot c_{i-1}, \qquad c_{-1} = 1 \end{cases} \tag{6.10}$$

where $i = 0, 1, \ldots, N-1$ and N is the number of digits of the final RB partial product to be converted to NB number. c_{-1} represents the first carry-in to the least significant digit (LSD) of the RB number.

In RB multiplier, the binary pair representing each RB digit can never become "1" simultaneously. This is because (1, 0) has been converted to (0, 1) before the RBA tree stage to eliminate the inconsistent representations of "0" in order to simplify the design of RBA_PNC cell [13]. Therefore, the inherent redundancy of this coding format gives rise to the following simplifications for the generation of the carry-generate bit g_i, the carry propagate bit p_i, and the half-sum bit d_i.

$$\begin{cases} g_i = f_i^+ \times f_i^- = f_i^+ \\ p_i = f_i^+ + f_i^- = f_i^- \\ d_i = f_i^+ \oplus f_i^- \end{cases} \tag{6.11}$$

With (6.11), the carry-out signal in (6.10) can be rewritten as:

$$c_i = f_i^+ + f_i^- \times c_{i-1}, \ c_{-1} = 1 \tag{6.12}$$

Therefore, fast RB-to-NB conversion problem can be traced back to the origin of fast CPA logic [26]. The main overhead of RB multiplication process lies in the conversion of final partial product summation result from the RB form back to its NB representation.

6.2.4 Review of Existing RB Multipliers

There are at least two properties of RB arithmetic that make it a viable and potential substitute for the conventional NB multiplier: (1) the RBA can be configured to add any RB numbers free of carry propagation; (2) communications among RBAs within and across different layers of RBA tree are simpler than those of the full and half adders of Carry Save Adder (CSA) tree of NB multiplier. The use of RBA tree for the accumulation of partial products makes a highly modular and regular cell structure that can be easily laid out on silicon.

Notwithstanding the carry-free addition property and regularity of RBAs, it has not led to as many new designs in RB multiplier as in NB multiplier. Over the last three decades, the number of RB multiplier architectures can be reviewed in three broad categories.

In the early 1980s, Takagi [23] proposed a high-speed multiplication algorithm, which used RB representation internally. Based on this algorithm, the RB multiplier architecture [27] was developed and a 16-bit RB multiplier was implemented on an LSI chip using a standard n-E/D MOS process [28]. It was the first silicon proven RB multiplier that demonstrated the speed competitiveness and performance repeatability in the digital multipliers of its time [14]. Later, enhanced performance RBA cell was developed based on the sign-magnitude coding format [29]. Similar RB multiplier designs were then implemented with CMOS process in [2, 29, 30] to obtain faster CMOS multipliers with a reduced number of transistors and good layout regularity.

The noteworthy progress of RB arithmetic had aroused the interest of computer architects and researchers to make further advancement in this field. In 1990s, a remarkably high-speed RB multiplier architecture was proposed by Makino et al. [31]. It was designed based on the positive–negative RB coding and possessed two distinctive features. One was the new design of RBA and the other was the RB-to-NB converter, which was implemented efficiently with carry-select method. This design was detailed further in [12, 32]. A number of RB multipliers have been proposed thereafter based on the same architectural concept [5, 6, 25]. Among them, an ingenious development came from Lee et al. [6]. This group of researchers proposed a radix-64 Booth encoding algorithm to emphatically improve the reduction rate of partial products. They defined 9 fundamental multiplying coefficients {0, 1, 2,

3, 4, 8, 16, 24, 32} so that any of the 65 multiplying coefficients of Booth-6 encoder can be represented by an RB number made up of two fundamental multiplying coefficients. The idea was to reduce the number of multiplying coefficients needed to improve the reduction rate of partial products. In the mean time, Besli and Deshmukh [25] proposed an RB Booth encoding algorithm to directly generate the partial products in RB format. This was perhaps the first successful resolution of hard multiple problem in high-radix Booth encoding algorithms by means of RB representation.

The third category of RB multiplier architectures was represented by the design described in [13]. A novel RBA cell was proposed with positive–negative-complement coding. More importantly, this work presented a method for RB-to-NB conversion using the so-called equivalent bit conversion algorithm. It claimed to eliminate the need for carry propagation in the final conversion stage by taking the full advantage of RB multiplication process. Unfortunately, the myth of carry-free reverse conversion had been shattered by a flaw in the truth table used in this algorithm. A carry chain in the conversion stage was erroneously neglected. The errors had been detected by several researchers [33, 34] and it was proven later that carry propagation is ineluctable in any multiplication process [34]. In [35], a direct-conversion scheme was also proposed without any carry propagation to minimize this critical path delay for parallel architectures. Despite the constant latency (i.e., it is independent of the word size) of this converter, carry propagation had been re-introduced into the revised addition rule. Therefore, the declaration was misleading as the original carry-free addition property had been completely abolished in the RBA tree of this multiplier.

From those unsuccessful attempts, we can conclude affirmatively that the parallel transformation from any RB representation to NB number without incurring some degree of carry propagation is impossible. Some analysis is required to optimize the trade-off between carry propagation and conversion efficiency. In RB multiplier, maintaining the carry-free addition property in the RBA tree is preferred to improving the reverse conversion efficiency at the expense of jeopardizing this property. With the carry-free RBA tree, the carry propagation is bound to be imposed on the final RB-to-NB conversion stage. The key point is that the carry-propagation delay occurs only once, at the very end, rather than in each addition step [36].

The Booth encoder and partial product generation (BEPPG) stage is yet another crucial stage in the trichotomy of RB multiplier architecture. Since negation in two's complement arithmetic requires carry propagation addition, negative partial product is more efficiently generated by a bit inversion of the multiplicand followed by an insertion of a "1" at its LSB position in the partial product summing tree. Therefore, one additional partial product row is generated to complete the two's complement negation of partial products for negative multiples. For example, Booth-2 encoding generates five instead of four partial products for an 8×8-bit multiplication. The additional delay required to add an extra partial product row critically slows down short multiplier due to the relatively fewer number of adder stages in its partial product summing tree. This is the case especially for the power-of-two operand lengths, which are the most commonly encountered word lengths for application-

specific datapaths and general computing benchmarks [37–39]. Therefore, new RB multiplier architecture with Booth encoders and partial product generators that eradicates the overhead of negation, especially for power-of-two operand lengths, is another perspective that has not been investigated adequately.

Owing to the absence of hard multiple, Booth-2 encoding is attractive in digital multiplier design. In [6], Booth-6 (radix-64) encoding was claimed to be optimal for RB multiplier design. The claim was substantiated by the performance ascendancy of their proposed RB multiplier over other RB multipliers that used Booth encoders of different radices. However, the comparisons were made based on the published experimental results targeted on different process technologies. From their experimental results, it is evident that the critical delay of Booth encoding and partial product generation of their scheme contributed to almost 41 % of the total delay time, which was much higher than the 26 % reported in [12]. Furthermore, a closed examination also reveals that the proposed Booth-6 encoder circuit, designed and optimized at transistor level, is actually Booth-3 encoder in disguise.

All in all, we observe a lack of systematic analysis of the fabrics that compose RB multiplier circuits. With better understanding of their performance trade-offs, particularly in terms of energy dissipation and delay, it will be less tedious to tailor the RB multiplier design to different application-specific constraints. Legitimate amalgamation of existing and newly proposed modules will be analyzed in the next section. The consolidation of these results will facilitate fruitful exploration of RB multiplier architectures with more desirable performance characteristics.

6.3 Architectural Exploration on Redundant Binary Booth Multipliers

For ease of exposition, we partition an RB multiplier into three major building blocks. They are the Booth encoder and partial product generator (BEPPG), which generates the RB partial products (RBPP) according to the selected multiples; the RB partial product summing tree, which compresses multiple RB partial products to a single RB number; and the RB-to-NB converter, which outputs the result in NB format. The anatomy of RB multiplier for functional analysis is shown in Fig. 6.3.

6.3.1 Taxonomy of Booth Encoders and Partial Product Generators

Being the front end circuits of RB multiplier design, the Booth encoder and partial product generator contribute critically to the performance and cost of the multiplier as a whole. How efficient the RB partial products are generated affects the area–delay–power trade-off of subsequent summation network. A Booth encoder can

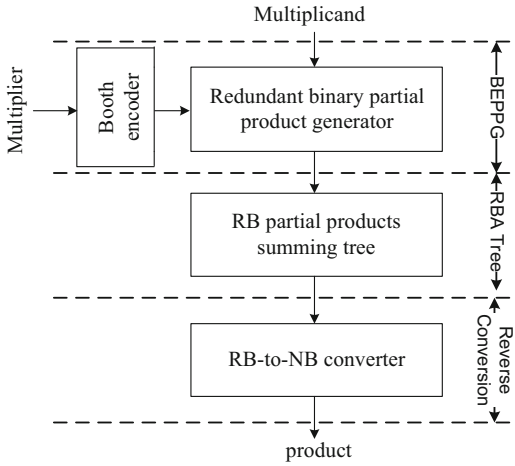

Fig. 6.3 Trichotomy of RB Booth multiplier architecture

be deemed as a digit-set converter as each slice of it converts a string of binary bits to a signed digit. The choice of a good digit-set converter for a given operand length is prerogative in that once it is fixed, the RB multiplier design loses a great deal of mobility on the speed-size optimization space. This subsection focuses on various configurations of the BEPPG modules based on the existing Booth encoding algorithms, which have been discussed in Sect. 6.2.2.

For the convenience of analysis, we make a dichotomy of Booth encoding algorithms according to the way in which the partial products are generated. Those Booth encoding methods that generate the partial products in NB format are classified as the normal binary Booth encoding (NBBE) and those others that generate the RB partial products directly are classified as redundant binary Booth encoding (RBBE). The partial product generator (PPG) is also known as the Booth decoder. Since the Booth encoder and decoder are normally dovetailed as a single entity, for brevity, the abbreviations of NBBE and RBBE are also used for the combined Booth encoder and partial product generator with no ambiguity.

6.3.1.1 Normal Binary Booth-*k* Encoding

In NB Booth-k algorithm (k is a positive integer), a Booth-encoded digit is generated from $k+1$ consecutive bits of an NB number. As illustrated in (6.4), the digit-set conversion process entails no carry propagation when $k \leq 2$. This is referred to as the simple Booth encoding, as opposed to the high-radix Booth encoding. In simple NBBE, NBBE-1 is obsoleted as it has zero partial product reduction rate. Therefore, the only useful one left is NBBE-2, which is widely used in high-speed digital multipliers to halve the number of NB partial products. To minimize the delay time and eliminate the glitches associated with the Booth multiplier, a modified NBBE was proposed in [40, 41]. Compared with NBBE-2, the modified NBBE-2

(MNBBE-2) saves one gate delay in the path of Booth encoder with the penalty of an increased number of gates used in the partial product generator.

With the radix value increases to $k > 2$, hard multiples emerge and mandate carry propagation additions, which complicate the realization of high-radix Booth encoders and their partial product generators. Although the number of partial products in the summation network can be proportionally reduced by increasing the radix of NBBE, there is a limit over which the advantage of high partial product reduction rate is offset by the cost of generating the hard multiples and the decoding logic.

Based on our classification, the partially redundant biased Booth encoding (PRBBE) algorithm reviewed in Sect. 6.2.2 falls under the high-radix NBBE category. This is because the partial products generated from PRBBE are in NB format.

6.3.1.2 Redundant Binary Booth-*k* Encoding

In RBBE scheme, most of the multiples can be expressed as a difference of two simple power-of-two multiples. The partial products that are generated conform to the format of positive–negative RB coding. This encoding method has eliminated the correction vector in the RB summing tree due to the two's complement arithmetic and RB coding. A representative of RBBE is that of [25]. However, as only one Booth-encoded digit is consumed for one RB partial product, half of the binary bits representing the RB partial product generated from the simple multiple in RBBE are filled with "0"s, which is rather inefficient.

A derivative of RBBE scheme is the new Covalent Redundant Binary Booth Encoding (CRBBE) presented in [42]. It binds two adjacent modified Booth encoders to compose an RB partial product. It shares the same advantages of RBBE for the ease of generating hard multiples and the avoidance of error compensation vector, which are the two problems associated with NBBE.

6.3.2 One-Digit BEPPG Module

To avoid superfluous simulation data from obscuring a meaningful analysis, we omit the less competitive parametric modules and focus our evaluation on those more representative and heterogeneous rivals. In consideration of the severity of hard multiple problem, it is reasonable to stop at radix-16 ($k \leq 4$) for high-radix NBBE and RBBE. For PRBBE, we consider the most appealing PRBBE-3 for the analysis based on the recommendation of [24]. This leaves us with altogether seven competitive Booth encoder and partial product generator (BEPPG) modules proposed in recent RB multiplier designs. The gate-level implementations of one slice of these BEPPG modules are illustrated in Fig. 6.4. It includes five NBBE

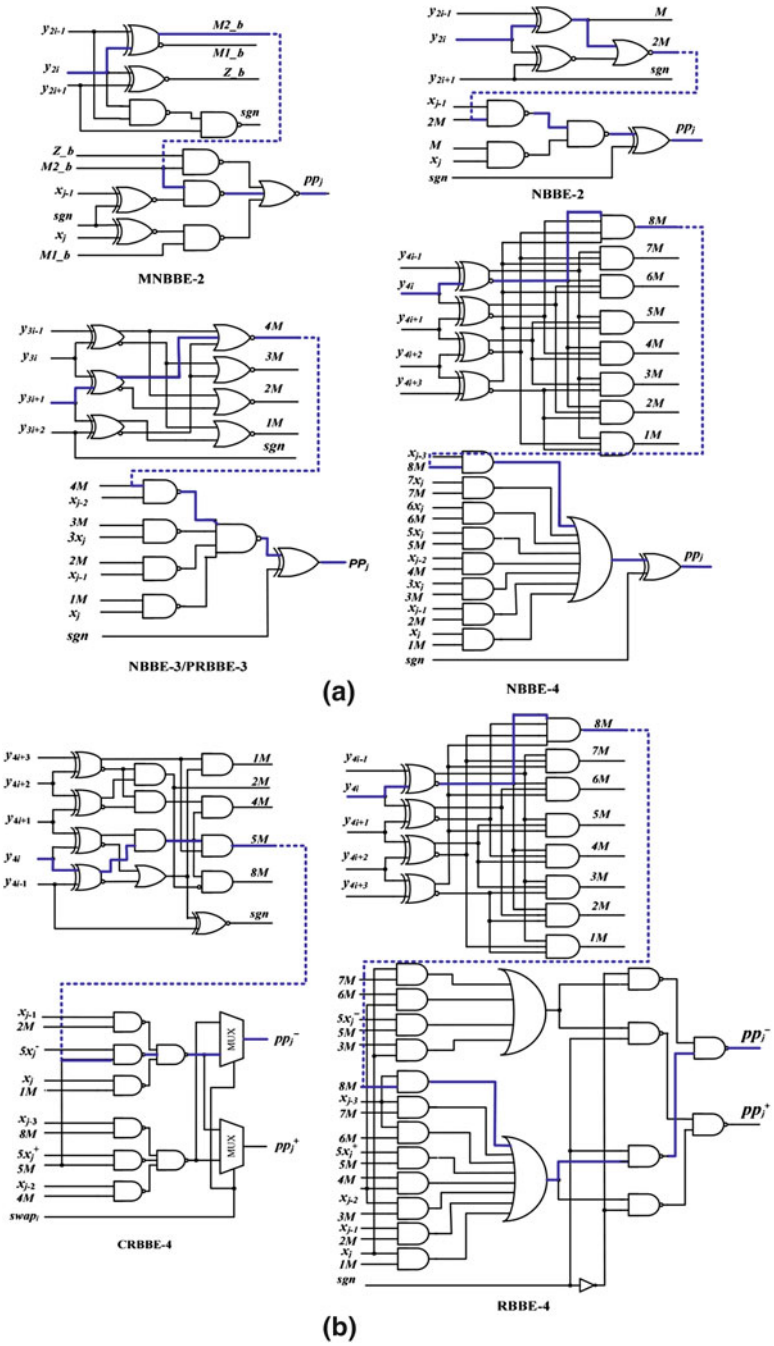

Fig. 6.4 Circuit implementation of BEPPG modules (**a**) Normal binary Booth encoding. (**b**) Redundant binary Booth encoding

designs in Fig. 6.4a and two RBBE designs in Fig. 6.4b. For each BEPPG slice in Fig. 6.4, a potential critical path is highlighted.

Apart from the difference in the generation of multiples, PRBBE-3 has exactly the same Booth encoder and partial product generator as NBBE-3 [24]. Therefore, it can be demonstrated with the same schematic as NBBE-3 in Fig. 6.4a. Similarly, as far as only the encoder logic is concerned, NBBE-4 is equivalent to RBBE-4. They are differentiated by the partial product generators. The CRBBE-4 circuit is implemented as shown in Fig. 6.4, by abutting two Booth-2 encoders with an auxiliary polarization circuit.

An abridged characterization of the area–time requirement to generate one digit of RB partial product is performed for each type of BEPPG modules. It should be noted that since the partial product generated by each slice of NBBE module is in NB form, two slices of NBBE based modules are required to generate one digit of RB partial product. The delay of each module is evaluated on the critical path and expressed in terms of the FO4 delay in a CMOS 0.18-μm process model, and the number of unit gates (a unit gate is equivalent to a two-input NAND gate) of the Booth encoder and the partial product generator are separately accounted for the area complexity. The characterization is shown in Table 6.5.

From Table 6.5, MNBBE-2 has the shortest delay time and NBBE-2 is the most compact design to generate one digit of RB partial product. For the same partial product reduction rate, RBBE-4 and CRBBE-4 are slower and more complex comparing with the above two modules. As this evaluation is made at digit level regardless of the type of RB partial product generated, the delay and complexity of the CPA required to generate the hard multiple have not been apportioned. Therefore, although NBBE-3 and PRBBE-3 generate the hard multiples differently, they exhibit the same performance in Table 6.5. Furthermore, high-radix Booth encoding modules, NBBE-3 and NBBE-4, are obviously inferior to the simple Booth encoding module in standalone comparison. However, due to the different partial product reduction rate, the landscape of RB multiplier employing these BEPPG modules might change as the length of the operand varies.

Table 6.5 Delay and unit gate number of one-digit BEPPG module

		No. of unit gate	
BEPPG	Delay (FO4)	BE	PPG
NBBE-2	6.208	14	12
MNBBE-2	4.952	18	22
NBBE-3	7.168	34	20
NBBE-4	8.456	66	36
PRBBE-3	7.168	34	20
RBBE-4	9.002	33	28
CRBBE-4	7.212	26	16

6.3.3 Qualitative Analysis of BEPPG on N × N-Bit RB Multipliers

Let N be the operand length of the RB multiplier, with Booth-k encoding, the number of Booth encoders, n_{BE}, and partial product generators, n_{PPG}, can be calculated by (6.13) and (6.14), respectively.

$$n_{\mathrm{BE}} = \left\lceil \frac{N}{k} \right\rceil \tag{6.13}$$

$$n_{\mathrm{PPG}} = (N + k - 1) \left\lceil \frac{N}{k} \right\rceil \tag{6.14}$$

Therefore, the total number of RB partial products in the summation network can be derived as in (6.15) and (6.16), for NBBE-k and RBBE-k, respectively.

$$n_{\text{PP-NBBE}} = \left\lceil \frac{N}{2k} \right\rceil + 1 \tag{6.15}$$

$$n_{\text{PP-RBBE}} = \left\lceil \frac{N}{k} \right\rceil \tag{6.16}$$

From (6.13), (6.14), (6.15), and (6.16), the number of Booth encoders and partial product generators is the same for the same radix of NBBE and RBBE algorithms, but the number of RB partial products generated from NBBE is around half of that generated from RBBE. Therefore, NBBE-k has approximately the same reduction rate of RB partial products as RBBE-$2k$. On the other hand, by comparing (6.15) and (6.16), it is noted that the correction vector needed by NBBE for RB coding and partial product negation has been eliminated in RBBE. If the bit length of the multiplier is exactly $2^{n+1} \times k$, the extra vector required by the NBBE multiplier will not only consume additional hardware and power consumption for its accumulation, but also extra delay in generating the final product.

Table 6.6 summarizes the characteristics of seven $N \times N$-bit multipliers employing different BEPPG modules of Fig. 6.4. It lists the quantity of various forms of resources, including the number of Booth encoders, the number of partial product generators, the number of RB partial products, and the number of stages of RBA summing tree. It also indicates whether or not the correction vector and the carry propagation adder are required for each RB multiplier.

From Table 6.6, it can be seen that the RB multiplier architecture employing NBBE-4 has the largest partial product reduction rate. Due to the least number of RB partial products, it may also have the smallest number of stages in the RBA summing tree. However, this advantage is offset by the requirement of CPAs for the

Table 6.6 Characteristics of $N \times N$-bit RB multiplier architectures with different BEPPGs

Multiplier	No. of BE	No. of PPG	No. of RBPPs	No. of RBA stages	CPA incurred	Correction vector incurred
NBBE-2	$\lceil \frac{N}{2} \rceil$	$(N+1) \times \lceil \frac{N}{2} \rceil$	$\lceil \frac{N}{4} \rceil + 1$	$\left\lceil \log_2^{(\lceil \frac{N}{4} \rceil + 1)} \right\rceil$	N	Y
MNBBE-2	$\lceil \frac{N}{2} \rceil$	$(N+1) \times \lceil \frac{N}{2} \rceil$	$\lceil \frac{N}{4} \rceil + 1$	$\left\lceil \log_2^{(\lceil \frac{N}{4} \rceil + 1)} \right\rceil$	N	Y
NBBE-3	$\lceil \frac{N}{3} \rceil$	$(N+2) \times \lceil \frac{N}{3} \rceil$	$\lceil \frac{N}{6} \rceil + 1$	$\left\lceil \log_2^{(\lceil \frac{N}{6} \rceil + 1)} \right\rceil$	Y	Y
NBBE-4	$\lceil \frac{N}{4} \rceil$	$(N+3) \times \lceil \frac{N}{4} \rceil$	$\lceil \frac{N}{8} \rceil + 1$	$\left\lceil \log_2^{(\lceil \frac{N}{8} \rceil + 1)} \right\rceil$	Y	Y
PRBBE-3	$\lceil \frac{N}{3} \rceil$	$(N+2) \times \lceil \frac{N}{3} \rceil$	$\left\lceil \frac{N+1}{6} \right\rceil + 1$	$\left\lceil \log_2^{(\lceil \frac{N+1}{6} \rceil + 1)} \right\rceil$	Y	Y
RBBE-4	$\lceil \frac{N}{4} \rceil$	$(N+3) \times \lceil \frac{N}{4} \rceil$	$\lceil \frac{N}{4} \rceil$	$\left\lceil \log_2^{\lceil \frac{N}{4} \rceil} \right\rceil$	N	N
CRBBE-4	$\lceil \frac{N}{4} \rceil$	$(N+3) \times \lceil \frac{N}{4} \rceil$	$\lceil \frac{N}{4} \rceil$	$\left\lceil \log_2^{\lceil \frac{N}{4} \rceil} \right\rceil$	N	N

generation of hard multiples. NBBE-3 and PRBBE-3 also face the same problem although only one hard multiple is needed to be generated. The RB multiplier architectures employing NBBE-2 and RBBE-4 have exactly the same feature as those with MNBBE-2 and CRBBE-4, respectively.

6.3.4 One-Digit RB Adder Cells

The RB partial product summing tree is the cornerstone of RB multiplier. The key component that appears abundantly in RB summing tree is the RB full adder cell. Three efficient representative RBA cells designed based on sign-magnitude, positive–negative, and positive–negative-complement coding in [12–14], respectively, are considered. To make the abbreviation meaningful, the RBA cells designed with the above coding methods are abbreviated as RBA_SM, RBA_PN, and RBA_PNC, respectively. Although the logic implementation of RB full adder may vary with the coding format used, they are essentially derived from the same adding rule. The underlying difference is the choice of appropriate intermediate control signals for the purpose of simplifying and optimizing the circuit in the selected coding format. Therefore, these RBA cells are evaluated here to manifest the effect of coding on the performance of RBA. Their gate-level circuit implementations are illustrated in Fig. 6.5. In addition, the corresponding half adder cells are also developed to simplify the design of RB partial product summing tree. These RB half adders are useful for the summation of an RB variable and an RB constant in some corner cells of the RB summing tree. The respective RB half adders are prefixed with RBHA to differentiate them from the RB full adders (RBFA).

Figure 6.5a shows the gate-level implementation of an RBA with sign-magnitude coding. Two intermediate signals, u_i and v_i, are introduced to make the realization of a simple circuit configuration compliant to the carry-free addition rule possible. It adds the two RB numbers expressed in the sign values, $a\ s\ i$ and $b\ s\ i$, and the absolute values $a\ a\ i$ and $b\ a\ i$ to produce the sum bits, $z\ s\ i$ and $z\ a\ i$. It is noted that, to simplify the circuit design, the input coding of (1,0) is refrained from feeding into the RBA directly, and 0, 1, and $\bar{1}$ are completely specified by (0,0), (0,1), and (1,1), respectively.

The RBFA cell designed based on the positive–negative coding is shown in Fig. 6.5b. It uses the intermediate signals, α_i and β_i to prevent continuous carry propagation by eliminating the collision of the sum and carry from the lower digit. Similarly, the inconsistent representation of zero, i.e., (1, 1) needs to be removed from the input before the operands are fed into the RBA cell.

Figure 6.5c shows the gate-level implementation of an RBA cell with positive–negative-complement coding. Contrary to the previous two coding methods, there is no intermediate control signals required. c_j is the output carry signal, which can be calculated directly from the input signals so that the chain of carry propagation is

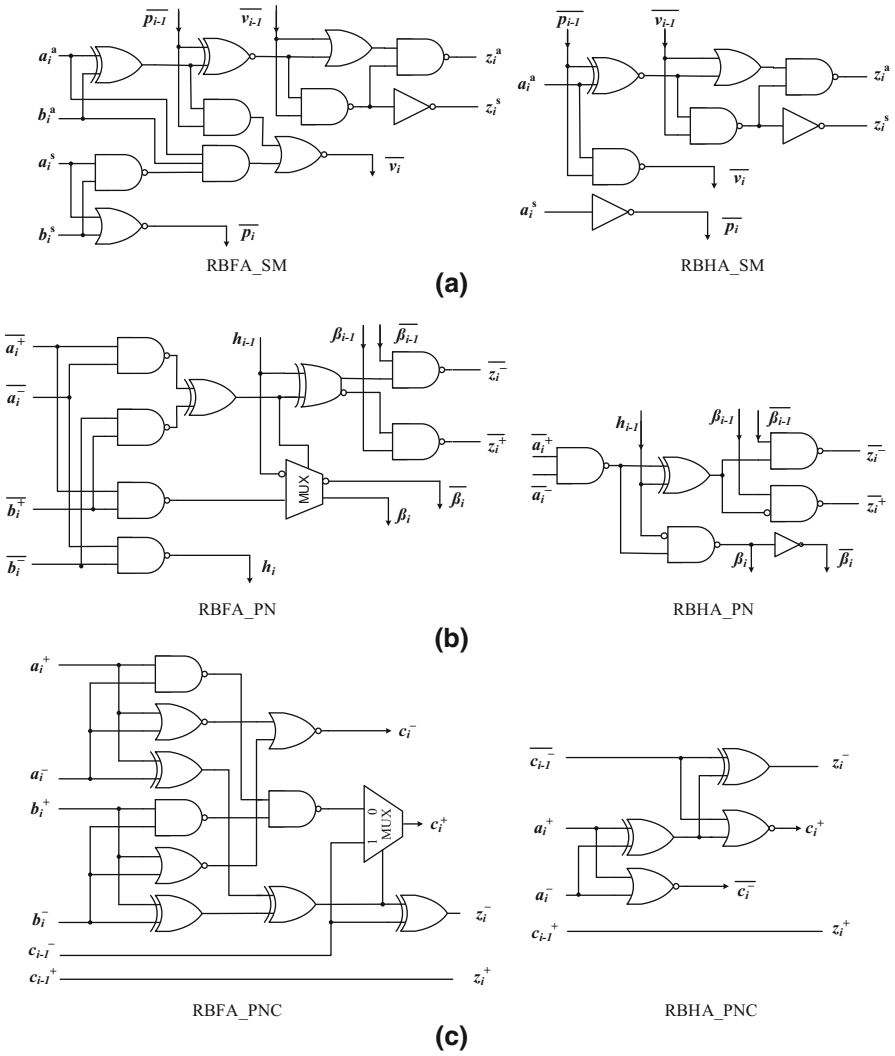

Fig. 6.5 Circuit implementation of RBA cells. (**a**) Sign-magnitude coding. (**b**) Positive–negative coding. (**c**) Positive–negative-complement coding

limited to only one adder. Furthermore, due to the symmetry of positive–negative-complement coding observed from Table 6.1, there is no preprocessing circuit required for each RB digit to avoid the inconsistent representations of "0" prior to its input into the RBA cell.

The FO4 delay with CMOS 0.18-μm process model and the number of unit gates of RB full and half adders for SM, PN, and PNC codings are summarized in Table 6.7.

Table 6.7 FO4 delay and complexity of redundant binary full and half adders

RBA cells	Delay (FO4)	No. of unit gate
RBFA_SM	3.824	15
RBHA_SM	2.924	7
RBFA_PN	3.456	17
RBHA_PN	2.822	8
RBFA_PNC	3.740	21
RBHA_PNC	2.606	8

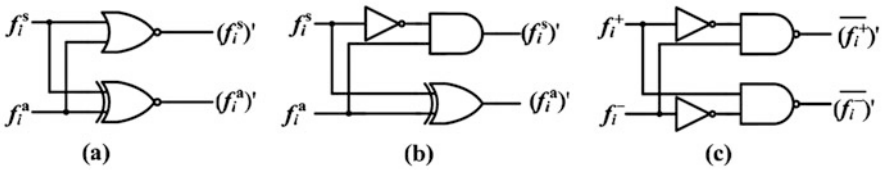

Fig. 6.6 Three different anterior converters used in RB multiplier design (**a**) SM coding for NBBE (**b**) SM coding for RBBE (**c**) PN coding for both NBBE and RBBE

Since the use of RBHA and RBFA is mutually exclusive and the critical path in the RB summing tree is dominated by the number of RBFAs, the delay of RBHA is of less significant in this comparison. This result indicates that RBA_PN is the fastest adder among the three coding schemes with moderate adder complexity. SM coding leads to the least complexity RB full and half adder cells, but these adder cells are also the slowest. RBA_PNC has an intermediate speed but uses the most number of gates.

6.3.5 Converters for RBA Interface

To fuse the heterogeneous fabrics designed with different coding formats into an RB multiplier, some simple converters are needed before and/or after the RBA summing tree. Although Booth encoding itself can be seen as a digit-set converter, its purpose is to reduce the number of stages required in the RB summing tree. Not all Booth encoding schemes discussed in Sect. 6.3.2 prepare the partial products in a form ready for consumption by the RBA. Some simple converters are required to convert the NB partial products to RB partial products prior to the RBA summing tree stage. Figure 6.6 shows three different one-digit anterior converters. Provision has also been made to eliminate the ambiguity of dual representations of "0" in these converters. Figure 6.6a illustrates the one-digit converter used in the conversion of NB partial products to RB partial products in order to add them with the RBA_SM summing tree for the NBBE based RB multipliers. Since the RB partial products generated directly from the RBBE algorithm assume the PN coding format, the anterior converter is needed to adapt them to the RBAs designed for other coding methods. Figure 6.6b depicts such a converter used to prepare the RB partial

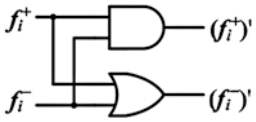

Fig. 6.7 Posterior converter used in RB-to-NB conversion for PNC coding

products generated by RBBE for the reduction by the RBA_SM summing tree. Figure 6.6c shows another converter used to adapt the RB partial products to RBA_PN summing tree for both NBBE and RBBE based RB multipliers. It should be noted that NB-to-RB converters are also necessary for RBA_PNC addition. However, each of them can be reduced to a simple inverter as indicated in Sect. 6.2.3 and absorbed into the receiving RBA cell.

Another kind of converter needed for the fusion of heterogeneous coding formats appears in the final stage of RB multiplier. This simple converter circuit is referred to as a posterior converter, which is used to adapt an RB-to-NB reverse converter circuit to the RB input of any coding format. The reverse conversion algorithm can be unified for all three coding methods as discussed in Sect. 6.2.3. The unanimity of carry generation using the same logical structure has already been taken care by the above forward converters for the SM and PN coding schemes. The redundant mappings have been removed prior to the RBA summing tree stage to simplify the RBA cell design. For the case of PNC, due to the coding symmetry, there is no need to eradicate the dual representations of zero before the RBA summing tree. The resolution of redundant mapping can be deferred until the RB-to-NB converter stage. Figure 6.7 illustrates the coherent converter, which is used only for the PNC coding in order to unify the reverse conversion algorithm.

6.4 Performance Evaluation

6.4.1 Configurations of RB Booth Multipliers

Based on the characteristics of the fabrics presented in Sect. 6.3, different configurations of RB Booth multipliers are delineated according to the types of RBA cells and converters in Table 6.8. The logic equations of one-digit anterior and posterior converters in NBBE and RBBE based multiplier architectures are also listed in the table where applicable. Any efficient parallel adder architectures can be employed to realize the RB-to-NB converter. The RB-to-NB converters implemented for the three RB coding schemes, SM, PN, and PNC, are abbreviated as CONV_SM, CONV_PN, and CONV_PNC, respectively.

If PNC coding is used for the RB multiplier design, the anterior converters can be saved, but posterior converters are introduced to remove the representation redundancy. From the logic equations, the delays of the anterior and posterior converters are about the same. The anterior converters for RBA_SM are slightly slower due to the longer XOR or XNOR gate delay. All converters have constant

Table 6.8 Configurations of RB multiplier with different code converters

RB multiplier architectures	Anterior converter	RBA cell	Posterior converters	RB-to-NB converter
NBBE	$\begin{cases} f_i^{s'} = \overline{f_i^s + f_i^a} \\ f_i^{a'} = f_i^s \odot f_i^a \end{cases}$	RBA_SM	N. A.	CONV_SM
RBBE	$\begin{cases} f_i^{s'} = \overline{f_i^s} \times f_i^a \\ f_i^{a'} = f_i^s \odot f_i^a \end{cases}$			
NBBE and RBBE	$\begin{cases} f_i^{+'} = \overline{f_i^+} \times f_i^- \\ f_i^{-'} = f_i^+ \times \overline{f_i^-} \end{cases}$	RBA_PN	N. A.	CONV_PN
NBBE and RBBE	N. A.	RBA_PNC	$\begin{cases} f_i^{+'} = f_i^+ \times f_i^- \\ f_i^{-'} = f_i^+ + f_i^- \end{cases}$	CONV_PNC

gate-delay time independent of the word length. The difference comes from the number of converter circuits required. A posterior converter is required for each digit in only the final sum of the RBA summing tree whereas an anterior converter is needed for every digit of the RB partial products before the RBA summing tree. The anterior converter circuits certainly outnumber the posterior converter circuits. From this point of view, PNC coding seems to be more efficient than the other two codings. Furthermore, for RB-to-NB converters, the delays of these three converters in CMOS implementation are comparable, while CONV_SM is slightly simpler.

The coding efficiency can be qualitatively analyzed in each stage as discussed. However, when different BEPPG modules are amalgamated with RBA summing tree using different RB coding methods, the efficiency of the RB multiplier design due to different coding methods cannot be easily ascertained. There is a bewildering number of design options, considering the number of modules substitutable in each stage of the RB Booth multiplier architecture. Every module has some intriguing merits of its own. The configuration becomes more competitive under certain operand length when the modules augment each other. The findings are best corroborated by the synthesis results.

To date, no systematic analysis has been made in the literature for different RB multiplier topologies under a uniform simulation and comparison strategy. In this section, a variety of RB multiplier topologies derived from several Booth encoding methods and three main RB coding schemes are implemented, synthesized, and compared for speed, power consumption, and energy-delay products. Altogether 21 different RB multipliers are built from various designs of each module of Table 6.8. The RB-to-NB converters of all these multipliers are designed with the same hybrid carry-lookahead/carry-select conversion algorithm which has been proven to be more energy efficient [36]. The following convention is adopted for the nomenclature of RB multipliers. Each multiplier is denoted by a prefix of its BEPPG module name indicated in Sect. 6.3.2 and a postfix of the designated coding format. Among these 21 RB multiplier configurations, 15 designs are presented for the first time in [43].

6.4.2 Numerical Simulation Results

This subsection presents the simulation results of RB multipliers for six commonly used operand lengths from 8 to 64 bits to extrapolate the performance trajectory of each multiplier as it scales. Each design is structurally described at gate level using VHDL. The designs are functionally verified by ModelSim for randomly generated input patterns before they are synthesized and mapped to Artisan TSMC 0.18-μm standard cell library [44] using the Synopsys Design Compiler [45] with a nominal wire load model. The mean power dissipation for each RB multiplier is calculated with the Monte Carlo statistical model with more than 99.9 % confidence level that the error is bounded below 3 %.

Table 6.9 Comparison of areas of RB multipliers

		Area (μm^2)					
S/N	RB multiplier architectures	8b	16b	24b	32b	48b	64b
1	NBBE-2_ SM	18,939	51,825	97,174	164,320	359,613	629,400
2	MNBBE-2_ SM	20,065	54,996	107,508	182,261	400,570	703,345
3	NBBE-3_ SM	22,055	57,292	97,144	148,954	295,150	505,769
4	NBBE-4_ SM	22,026	59,099	99,009	155,450	319,171	550,086
5	PRBBE-3_ SM	22,705	59,644	102,981	164,645	344,313	584,378
6	RBBE-4_ SM	23,616	65,721	119,381	211,051	444,232	716,549
7	CRBBE-4_ SM	19,059	54,782	102,656	168,407	359,431	646,877
8	NBBE-2_PN	21,142	57,073	102,125	169,905	367,058	646,794
9	MNBBE-2_PN	22,076	60,574	112,995	186,667	408,234	755,796
10	NBBE-3_PN	23,105	58,932	104,335	153,411	300,463	536,930
11	NBBE-4_PN	24,158	63,919	106,939	162,929	320,321	580,345
12	PRBBE-3_PN	24,253	63,528	107,564	171,882	346,699	603,792
13	RBBE-4_PN	26,882	69,919	126,832	229,283	460,102	761,062
14	CRBBE-4_PN	21,295	59,034	110,154	177,137	376,428	669,862
15	NBBE-2_PNC	19,536	56,537	106,098	175,451	376,892	659,308
16	MNBBE-2_PNC	21,152	58,038	115,946	186,179	411,403	752,977
17	NBBE-3_PNC	22,513	60,038	101,508	158,051	315,466	555,377
18	NBBE-4_PNC	24,534	63,289	104,514	166,072	330,264	586,514
19	PRBBE-3_PNC	23,152	62,554	108,320	174,147	357,295	621,987
20	RBBE-4_PNC	24,988	69,011	130,045	242,481	471,647	769,544
21	CRBBE-4_ PNC	20,817	57,829	109,324	176,254	388,313	691,969

Since multiplication is often the speed-limiting elements in application, the synthesis tool is set for speed optimization. Table 6.9 shows the area results of the synthesis. It indicates the relative complexity of the RB multipliers in comparison when area optimality is traded for speed. Table 6.10 lists the worst-case delays of different sizes of RB multiplier configurations. The power consumption is also simulated based on the maximum input rate that each individual multiplier is able to function. Therefore, the energy per operation of each design can be obtained by multiplying the average power consumption with the worst-case delay. These results are summarized in Table 6.11. The area–time–energy trade-offs are illuminated in a three-dimensional scatter plot of Fig. 6.8, where the abscissas are the natural logarithm of energy dissipation in pJ and worse case delay in ns, and the ordinate is the natural logarithm of area in μm^2. Different shapes and symbols are used to denote different RB multiplier configurations. The shapes and symbols are colored in blue, red, and green to indicate the RB coding schemes of SM, PN, and PNC, respectively.

Table 6.10 Comparison of worst-case delays of RB multipliers

		Delay (ns)					
S/N	RB multiplier architectures	8b	16b	24b	32b	48b	64b
1	NBBE-2_ SM	1.906	2.581	3.011	3.405	3.856	4.427
2	MNBBE-2_ SM	1.766	2.401	2.795	3.194	3.679	4.209
3	NBBE-3_ SM	2.158	2.799	3.329	3.628	4.329	4.702
4	NBBE-4_ SM	2.433	3.192	3.750	4.109	4.756	5.156
5	PRBBE-3_ SM	1.985	2.629	3.156	3.487	4.078	4.582
6	RBBE-4_ SM	1.823	2.518	3.050	3.429	4.039	4.453
7	CRBBE-4_ SM	1.675	2.269	2.863	3.167	3.738	4.181
8	NBBE-2_PN	1.787	2.358	2.723	3.064	3.562	4.011
9	MNBBE-2_PN	1.647	2.219	2.591	2.923	3.392	3.870
10	NBBE-3_PN	2.131	2.708	3.154	3.433	4.002	4.391
11	NBBE-4_PN	2.303	3.019	3.469	3.872	4.418	4.819
12	PRBBE-3_PN	1.802	2.401	2.829	3.202	3.735	4.249
13	RBBE-4_PN	1.741	2.351	2.828	3.197	3.786	4.184
14	CRBBE-4_PN	1.629	2.193	2.651	2.892	3.454	3.812
15	NBBE-2_PNC	1.809	2.468	2.834	3.286	3.684	4.297
16	MNBBE-2_PNC	1.677	2.236	2.638	3.034	3.531	4.054
17	NBBE-3_ PNC	2.143	2.743	3.232	3.511	4.188	4.672
18	NBBE-4_PNC	2.314	3.057	3.576	4.025	4.595	4.976
19	PRBBE-3_PNC	1.885	2.502	2.925	3.301	3.915	4.419
20	RBBE-4_PNC	1.712	2.396	2.865	3.246	3.898	4.295
21	CRBBE-4_ PNC	1.588	2.159	2.652	2.969	3.579	3.938

6.4.3 Analyses and Discussions

The voluminous amount of data makes the analysis difficult due to the intricate correlation between different contributing factors. In this subsection, the results are discussed in three perspectives. First, the Booth encoder and decoder complexity of two different classes of Booth multipliers and the effect of extra correction vector as the size of the multiplier changes. Second, the adversity of hard multiples as the radix of Booth multiplier increases. Third, the impact of RB coding method on the overall performance of the multiplier. Since the coding efficiency analysis has been decoupled in the first two discussions, only the results of RB multipliers with PN coding are presented for the analysis in Sects. 6.4.3.1 and 6.4.3.2. The exceptions that deviate from the general extrapolation are singled out for separate discussion in these subsections.

Table 6.11 Comparison of energy dissipations of RB multipliers

		Energy dissipation (pJ)					
S/N	RB multiplier architectures	8b	16b	24b	32b	48b	64b
1	NBBE-2_ SM	4.789	14.82	29.98	53.78	122.05	223.42
2	MNBBE-2_SM	5.484	16.83	35.01	61.64	140.07	249.65
3	NBBE-3_ SM	5.427	15.94	30.19	50.29	106.16	196.62
4	NBBE-4_ SM	6.133	18.68	34.26	55.93	117.77	210.93
5	PRBBE-3_ SM	5.322	16.57	31.99	54.92	120.04	210.29
6	RBBE-4_ SM	5.684	18.99	35.04	62.96	142.33	252.54
7	CRBBE-4_ SM	5.093	16.03	32.93	56.57	130.18	228.24
8	NBBE-2_PN	4.931	16.04	31.95	57.66	127.11	233.41
9	MNBBE-2_PN	5.651	17.85	36.56	63.98	144.19	265.54
10	NBBE-3_PN	5.451	16.76	32.36	54.92	114.07	208.03
11	NBBE-4_PN	6.496	19.99	37.62	60.07	127.12	228.48
12	PRBBE-3_PN	5.552	17.43	33.75	58.06	124.15	224.97
13	RBBE-4_PN	5.661	18.84	36.49	66.09	147.93	264.06
14	CRBBE-4_PN	5.135	16.25	33.46	59.08	132.80	238.56
15	NBBE-2_PNC	4.916	15.52	31.51	55.73	123.95	229.78
16	MNBBE2_PNC	5.607	17.73	37.49	62.78	143.21	258.92
17	NBBE-3_PNC	5.480	16.54	31.82	53.51	111.13	205.19
18	NBBE-4_PNC	6.584	19.13	35.96	59.11	123.64	222.84
19	PRBBE-3_PNC	5.544	17.02	33.98	56.63	123.05	220.96
20	RBBE-4_PNC	5.646	19.07	37.11	68.03	145.78	259.87
21	CRBBE4_PNC	5.084	16.25	34.07	58.37	131.75	237.02

6.4.3.1 Normal Binary Booth Encoding vs. Redundant Binary Booth Encoding

As discussed in Sect. 6.3.2, Booth encoding is classified as NBBE and RBBE depending on the way their RB partial products are generated. For the same radix number, the partial product reduction rate of NBBE is double that of RBBE. To account for the effects due to the use of different types of Booth encoder and decoder, a reasonable and meaningful comparison shall be made based on the same RB partial product reduction rate. Therefore, two NBBE multipliers, namely NBBE-2 and MNBBE-2, and two RBBE multipliers, namely RBBE-4 and CRBBE-4, with the same reduction rate of 1/4 have been selected for this discussion.

From Table 6.10, it is found that CRBBE-4 multiplier is the fastest design for all the power-of-two operand lengths. For these operand lengths, CRBBE-4 multiplier executes on average 6.60 %, 1.21 %, and 7.90 % faster than NBBE-2, MNBBE-2, and RBBE-4 multipliers, respectively. Due to the existence of correction vector, the speed of NBBE multiplier is degraded by the additional stage in the partial product summation network. For 24-bit and 48-bit multipliers, MNBBE-2 multiplier executes on average 4.81 %, 9.39 %, and 2.03 % faster than NBBE-2, RBBE-4, and CRBBE-4 multipliers, respectively. This is because when the operand length is not

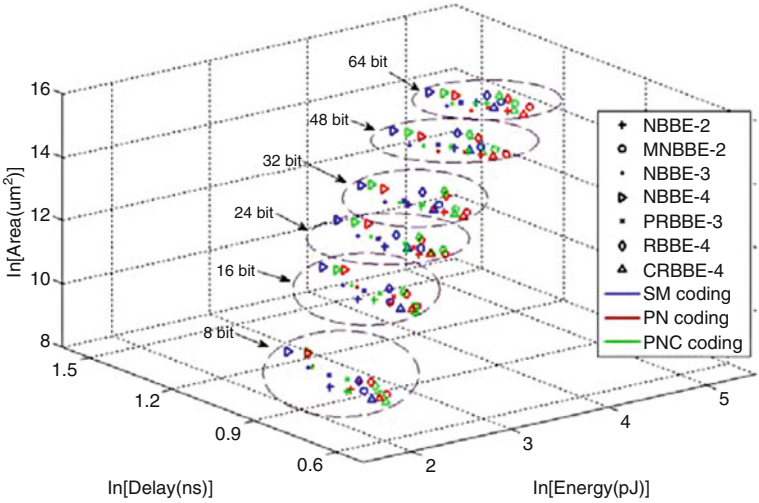

Fig. 6.8 Scatter plot of area vs. worst-case delay and energy dissipation in natural logarithmic scale

a power-of-two integer, the extra correction vector contributes little or no effect to the critical path delay.

From Table 6.11, it is evident that NBBE-2 multiplier always consumes the least energy. It saves about 11.55 %, 13.11 %, and 3.10 % energy comparing with MNBBE-2, RBBE-4, and CRBBE-4 multipliers, respectively. The breakdown of our power analysis results reveals that although MNBBE-2 multiplier consumes the least switching power, it consumes larger cell internal power, which can probably be imputed to its larger gate internal capacitance. CRBBE-4 multiplier is secondary in energy and its energy consumption approximates that of NBBE-2 multiplier. Despite having lower complexity of Booth encoder and partial product generator, the RBAs in the RB partial product summing tree of NBBE-2 multiplier outnumber those of CRBBE-4 multiplier, which accounts for the reduced advantage in energy dissipation. RBBE-4 multiplier presents lower speed and dissipates more energy than CRBBE-4 multiplier for all word lengths. This is primarily due to its less efficient encoder and much more complicated partial product generator.

If both speed and energy consumption are equally important, the combined effect of energy efficiency is best benchmarked using the energy-delay product (EDP) metric. Figure 6.9 shows the EDP of RB multipliers of these four multipliers. The EDP for each operand length is normalized so that the multiplier with the largest EDP has an EDP of one. The results show that CRBBE-4 multiplier is the most energy efficient for the power-of-two operand lengths, and NBBE-2 multiplier tops all multipliers for operand lengths that are not power-of-two integers.

Similar trends of delay, energy, and EDPs are also observed for the same four multiplier architectures with SM and PNC codings except that the extent of performance difference in each case varies.

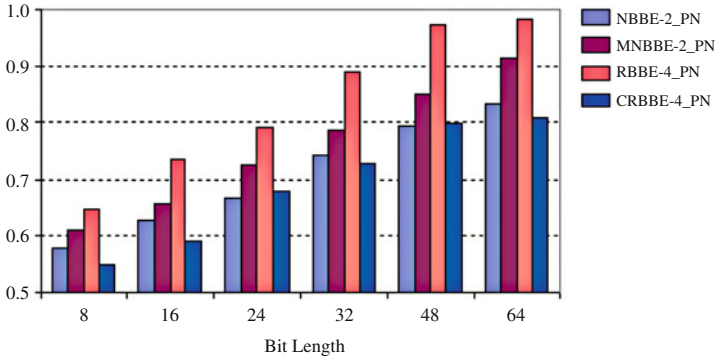

Fig. 6.9 Normalized EDP of NBBE and RBBE multipliers

6.4.3.2 High-Radix Booth Encoding Vs. Simple Booth Encoding

As indicated in Sect. 6.2.2, the existence of hard multiples is a major issue of high-radix Booth encoding schemes. To assess the significance of hard multiples, the high-radix Booth encoding schemes, NBBE-3, PRBBE-3, and NBBE-4, are compared with the simple Booth encoding NBBE-2.

From Table 6.10, it is conspicuous that high-radix Booth multipliers are slower in this group of RB multipliers. On average, NBBE-2 multiplier is faster than NBBE-3, NBBE-4, and PRBBE-3 multipliers by 12.19 %, 20.47 %, and 3.49 %, respectively. The delay time aggravates as the radix number increases in the high-radix Booth multipliers. This shows that the generation of hard multiples is indeed a major performance stumbling block of these multipliers.

As observed from Table 6.11, among the three high-radix Booth multipliers, NBBE-3 multiplier consumes the least energy in view of a better trade-off between the complexity of RBA summing tree and the number of CPAs required for their hard multiple generation. The energy saving of NBBE-2 multiplier is not prominent and it diminishes gradually as the operand length increases. It exhibits 9.54 %, 4.27 %, and 1.27 % lower energy dissipation than NBBE-3 for 8-bit, 16-bit, and 24-bit multipliers, respectively. When the word length increases to 32, 48, and 64 bits, it begins to consume 4.75 %, 10.26 %, and 10.87 %, respectively more energy than NBBE-3 multiplier. This can be explained as follows. Comparing with NBBE-2 multiplier, NBBE-3 multiplier has more complex Booth encoder and selector logic, as well as high overhead of hard multiple generation. When the size of the multiplier is small, excessive energy are dissipated in these logic circuits. As the word length of the multiplier increases, more RB partial products can be reduced by NBBE-3 and the energy reduction in the RBA summing tree offsets these logic overheads.

For the same rate of partial product reduction, it is interesting to note that with small length adders and additional compensation vector, PRBBE-3 multiplier outperforms NBBE-3 multiplier with a penalty of more energy dissipation. Figure 6.10a shows the normalized EDP of these four RB multipliers graphically.

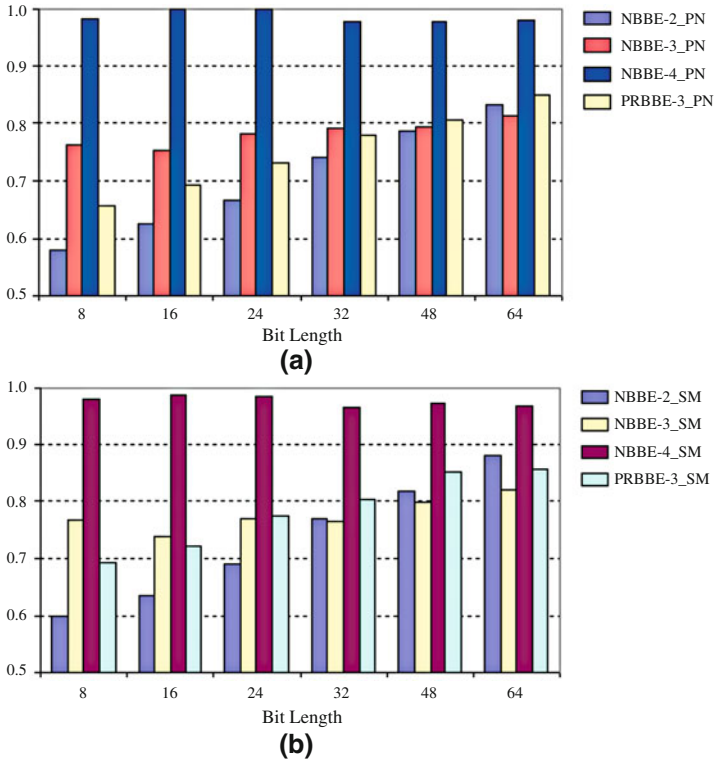

Fig. 6.10 Normalized EDP of high-radix and simple Booth multipliers. (**a**) PN coding. (**b**) SM coding

It indicates that NBBE-2 multiplier is the most energy efficient design from 8 bits to 48 bits but the efficiency decreases gradually and it loses out to NBBE-3 multiplier when the operand length increases to 64 bits.

These four multiplier architectures with PNC coding follow similar trends in delay, energy, and EDP comparisons. Therefore, the above analysis is also valid for PNC coding. However, the EDP rankings of the same four multipliers designed with SM coding have some subtle differences as illustrated in Fig. 6.10b. It is noted that NBBE-3 multiplier becomes advantageous from 32 bits onwards instead of 64 bits. The EDP gaps between NBBE-3 and PRBBE-3 shown in Fig. 6.10b also display a different trend from that shown in Fig. 6.10a. This has led to the following investigation on the coding efficiency.

6.4.3.3 Redundant Binary Coding Efficiency

It can be observed from Table 6.10 that most of the RB multipliers implemented with PN coding are faster than their counterparts implemented with the other two

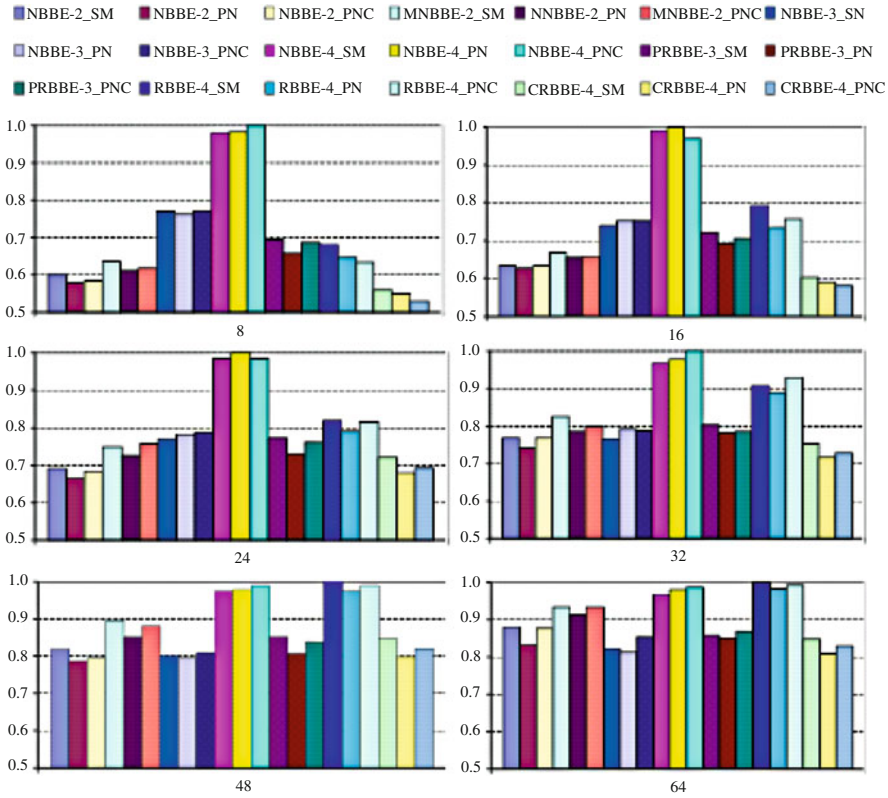

Fig. 6.11 Normalized EDP of all Booth multipliers. The sizes of multipliers from *top left* to *bottom right* are 8, 16, 24, 32, 48, and 64 bits

codings. So generally speaking, designs with positive–negative coding have higher speed. This is probably due to the different RBA cells. RBA PN is the fastest among the three RBA cells. On the other hand, as noted in many cases of Table 6.11, SM coding produces the multiplier designs with the least energy consumption. This is also consistent with the earlier qualitative analysis, which indicates that the RB multiplier implemented with RBA_SM and CONV_SM has the least logic complexity. To further investigate the RB coding efficiency, the normalized EDP values for all RB multipliers are consolidated in Fig. 6.11 with one chart for each word length from 8 bits to 64 bits.

From these results, the following conclusions can be made:

1. It is difficult to make a conclusive inference on coding efficiency, but some RB coding schemes are found to benefit certain Booth multiplier architectures. In most situations, NBBE-2, MNBBE-2, and PRBBE-3 multipliers perform better with PN coding; NBBE-3 and NBBE-4 multipliers are more efficient with

SM coding. CRBBE-4 and RBBE-4 multiplier is more energy efficient with PNC coding only when the operand length is small, and PN coding is more advantageous when the operand length becomes larger.

2. For power-of-two operand lengths, CRBBE-4_PN multiplier achieves the smallest EDP. This is because its speed advantage outweighs the somewhat higher energy dissipation in the RBA summing tree. However, this ascendancy in EDP becomes less prominent when the word length increases. This is possibly caused by its relatively complex Booth encoder and decoder logic in the partial product generation in comparison with that of NBBE multiplier.
3. For operand lengths that are not power-of-two integers, NBBE-2_PN and NBBE-3_SM multipliers outperform other 24-bit and 48-bit multipliers, respectively. This empirical result is also consistent with the qualitative analysis made pertaining to the high-radix and simple Booth encoding methods.

6.5 Conclusions

In this chapter, high-performance Booth multiplier based on RB representation has been investigated. The design considerations on several building modules and their logic circuits have been qualitatively discussed at a higher level of abstraction to highlight the potential performance trade-off before further empirical study. The unification of the reverse converter and the anterior and posterior interfacing logic make composition of RB multipliers from heterogeneously encoded modules possible. Upon ruling out incompatible and non-competitive architectural options, 21 different configurations (most of them are novel circuit configurations not explicitly reported in the literature) of $N \times N$-bit RB multiplier architectures have been constructed from combinations of various designs of each module. These RB multipliers have been implemented, simulated, analyzed, and compared for different operand lengths from $N = 8$ to 64. The investigation has been carried out impartially using a consistent synthesis setup and an appropriate figure of merit. To summarize, high-radix Booth multiplier is not suitable for speed-dominant design, but it remains an attractive choice for low power applications with large dynamic range. Covalent redundant binary Booth encoding is recommended for power-of-two operand lengths for its high-speed and low energy-delay product especially for digital multimedia applications where 8-bit and 16-bit multiplications are ubiquitous. We have also shown that the advantages of some topologies can be undermined by the types of RB coding format used. In general, sign-magnitude coding is more likely to produce lower power designs for the same Booth multiplier architecture, while positive–negative coding tends to yield higher speed designs.

References

1. H.-C. Chow, I.-C. Wey, A 3.3V 1 GHz high speed pipelined booth multiplier, in *Processing of the 2002 IEEE International Symposium Circuits and Systems (ISCAS)*, vol. 1, Arizona, USA, May 2002, pp. 457–460
2. H. Edamatsu, T. Taniguchi, T. Nishiyama, S. Kuninobu, A 33 MFLOPS floating point processor using redundant binary representation, in *1988 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, San Francisco, USA, Feb 1988, pp 152–153, 342–343
3. M. Tonomura, High-speed digital circuit of discrete cosine transform. IEICE Trans. Fundam. **E78-A**(8), 1342–1350 (1995)
4. Z. Yu, M.L. Yu, K. Azader, A.N. Willson Jr, A low power adaptive filter using dynamic reduced 2's-complement representation, in *Proceedings of the 2002 IEEE Custom Integrated Circuit Conference (CICC)*, Orlando, FL, USA, May 2002, pp. 141–144
5. N. Itoh, Y. Naemura, H. Makino, Y. Nakase, T. Yoshihara, Y. Horiba, A 600-MHz 54×54-bit multiplier with rectangular-styled Wallace tree. IEEE J. Solid-State Circuits **36**(2), 249–257 (2001)
6. S.-H. Lee, S.-J. Bae, H.-J. Park, A compact radix-64 54×54 CMOS redundant binary parallel multiplier. IEICE Trans. Electron. **E85-C**(6), 1342–1350 (2002)
7. J.-Y. Kang, J.-L. Gaudiot, A simple high-speed multiplier design. IEEE Trans. Comput. **55**(10), 1253–1258 (2006)
8. A. Vazquez, E. Antelo, J.D. Bruguera, Fast radix-10 multiplication using redundant BCD codes. IEEE Trans. Comput. **63**, 1902–1914 (2014)
9. J. Gu, C.H. Chang, K.S. Yeo, Algorithm and architecture of a high density, low power scalar product macrocell. IEEE Proc. Comput. Digit Technol. **151**(2), 161–172 (2004)
10. S.F. Hsiao, M.R. Jiang, J.S. Yeh, Design of high-speed low-power 3–2 counter and 4–2 compressor for fast multipliers. Electron. Lett. **34**(4), 341–343 (1998)
11. Z. Wang, G. Jullien, W.C. Miller, A new design technique for column compression multipliers. IEEE Trans. Comput. **44**(8), 962–970 (1995)
12. H. Makino, Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara, K. Mashiko, An 8.8-*ns* 54×54-bit multiplier with high speed redundant binary architecture. IEEE J. Solid-State Circuits **31**(6), 773–783 (1996)
13. Y. Kim, B.-S. Song, J. Grosspietsch, S.F. Gillig, A carry-free $54b \times 54b$ multiplier using equivalent bit conversion algorithm. IEEE J. Solid-State Circuits **36**(10), 1538–1545 (2001)
14. Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, N. Takagi, A high-speed multiplier using a redundant binary adder tree. IEEE J. Solid-State Circuits **22**(1), 28–34 (1987)
15. R. Kattamuri, S.K. Sahoo, Computation sharing multiplier using redundant binary arithmetic, in *Proceedings of the 2010 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Kuala Lumpur, Malaysia, Dec 2010, pp. 108–111
16. T.-B. Juang, C.-C. Wei, C.H. Chang, Area-saving technique for low-error redundant binary fixed-width multiplier implementation, in *Proceedings of the 2009 IEEE International Symposium on Integrated Circuits*, Singapore, Dec 2009, pp. 550–553
17. S.K. Reddy, S.K. Sahoo, S. Chakraborty, A high speed, high Radix 32-bit Redundant parallel multiplier, in *Proceedings 2011 IEEE International Conference on Emerging Trends in Electrical and Computer Technology*, Nagercoil, India, Mar 2011, pp. 917–921
18. S.K. Sahoo, A. Ashati, R. Sahoo, C. Shekhar, A high-speed radix-64 parallel multiplier using a novel hardware implementation approach for partial product generation based on redundant binary arithmetic, in *Proceedings of the 2008 IEEE International Conference on Emerging Trends in Engineering and Technology,* Nagpur, India, July 2008, pp. 474–479
19. E.K.L. Surendran, P.R. Antony, Implementation of fast multiplier using modified radix-4 Booth algorithm with redundant binary adder for low energy applications, in *Proceedings of the 2014 IEEE International Conference on Computational Systems and Communications*, Trivandrum, Kerala, India, Dec 2014, pp. 266–271

20. S. Dutt, A. Chauhan, R. Bhadoriya, S. Nandi, G. Trivedi, A high-performance energy-efficient hybrid redundant MAC for error-resilient applications, in *Proceedings of the 2015 International Conference VLSI Design*, Bangalore, India, Jan 2015, pp. 351–356
21. A.M. Shams, T.K. Darwish, M.A. Bayoumi, Performance analysis of low-power 1-bit CMOS full adder cells. IEEE Trans. VLSI Syst. **10**(1), 20–29 (2002)
22. A. Avizienis, Signed-digit number representations for fast parallel arithmetic. IRE Trans. Electron. Comput. **EC-10**, 389–400 (1961)
23. N. Takagi, A high-speed multiplier with a regular cellular array structure using redundant binary representation, Technical Report R82-14, Yajima Lab, Department of Information Science, Kyoto University, Kyoto, Japan, Jun 1982
24. G.W. Bewick, Fast multiplication: algorithms and implementation, Ph.D. dissertation, Stanford University, Feb 1994
25. N. Besli, R.G. Deshmukh, A novel redundant binary signed-digit (RBSD) Booth's encoding, in *Proceedings of the 2002 IEEE Southeast Conference,* Columbia, South Carolina, USA, Apr 2002, pp. 426–431
26. G.M. Blair, The equivalence of twos-complement addition and the con-version of redundant-binary to twos-complement numbers. IEEE Trans. Circuits Syst. I Regul. Pap. **45**, 669–671 (1998)
27. N. Takagi, H. Yasuura, S. Yajima, High-speed VLSI multiplication algorithm with a redundant binary addition tree. IEEE Trans. Comput. **C-34**(9), 789–796 (1985)
28. Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, N. Takagi, High speed multiplier LSI using a redundant binary adder tree, in *Proceedings of the 1984 IEEE International Conference Computer Design (ICCD'1984)*, New York, USA, Oct 1984
29. S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, N. Takagi, Design of high-speed MOS multiplier and divider using redundant binary representation, in *Proceedings of the 8th IEEE Symposium Computer Arithmetic*, Como, Italy, May 1987, pp. 80–86
30. S. Kuninobu, T. Nishiyama, T. Taniguchi, High speed MOS multiplier and divider using redundant binary representation and their implementation in a microprocessor. IEICE Trans. Electron. **E76-C**(3), 436–445 (1993)
31. H. Makino, Y. Nakase, H. Shinohara, An 8.8-ns 54×54-bit multiplier using new redundant binary architecture, in *Proceedings of the 1993 IEEE International Conference Design (ICCD'1993)*, Cambridge, Massachusetts, USA, Oct 1993, pp. 202–205
32. H. Makino, H. Suzuki, H. Morinaka, Y. Nakase, K. Mashiko, T. Sumi, A 286 MHz 64-b floating point multiplier with enhanced CG operation. IEEE J. Solid-State Circuits **31**(4), 504–513 (1996)
33. M.D. Ercegovac, T. Lang, Y. Kim, B.S. Song, J. Grosspietsch, S.F. Gillig, Comments on to 'A carry-free 54b×54b multiplier using equivalent bit conversion algorithm'. IEEE J. Solid-State Circuits **38**(1), 160–161 (2003)
34. W. Rulling, A remark on carry-free binary multiplication. IEEE J. Solid-State Circuits **38**(1), 159–160 (2003)
35. I. Choo R.G. Deshmukh, A novel conversion scheme from a redundant binary number to two's complement binary number for parallel architectures, in *Proceedings of the 2001 IEEE Southeast Conference*, Clemson, South Carolina, USA, Apr 2001, pp. 196–201
36. Y. He, C.H. Chang, A power-delay efficient hybrid carry-lookahead/carry-select based redundant binary to two's complement converter. IEEE Trans. Circuits Syst. I Regul. Pap. **55**(1), 336–346 (2008)
37. O.T.-C. Chen, L.-H. Chen, N.-W. Lin, C.-C. Chen, Application-specific data path for highly efficient computation of multistandard video codecs. IEEE Trans. Circuits Syst. Video Technol. **17**(1), 26–42 (2007)
38. S. Perri, P. Corsonello, G. Cocorullo, A 64-bit reconfigurable adder for low power media processing. Electron. Lett. **38**(9), 397–399 (2002)
39. N. Slingerland, A.J. Smith, Measuring the performance of multimedia instruction sets. IEEE Trans. Comput. **51**(11), 1317–1332 (2002)

40. R. Fried, Minimizing energy dissipation in high-speed multipliers, in *Proceedings of the 1997 IEEE International Symposium Low Power Electronics and Design*, Monterey, California, USA, Aug 1997, pp. 214–219
41. W.-C. Yeh, C.-W. Jen, High-speed Booth encoded parallel multiplier design. IEEE Trans. Comput. **49**(7), 692–701 (2000)
42. Y. He, C.H. Chang, A new redundant binary Booth encoding for fast 2n-bit multiplier design. IEEE Trans. Circuits Syst. I Regul. Pap. **56**(6), 1192–1201 (2009)
43. Y. He, Design and analysis of redundant binary Booth multipliers, Ph.D. dissertation, the Nanyang Technological University, Jun 2007
44. TSMC 0.18μm Process 1.8-Volt SAGE-X™ Standard Cell Library Databook, Artisan Components, Inc., Oct 2001
45. Design Compiler User Guide, Synopsys, Inc., Sept 2003

Chapter 7
Robust Analog Arithmetic Based on the Continuous Valued Number System

Babak Zamanlooy and Mitra Mirhassani

7.1 Introduction

To implement arithmetic architectures there are many approaches available, some with new logic families such as *Multiple-Valued-Logic* (MVL) [1, 2]. In this type of arithmetic systems, the signal levels in the digits are increased from a conventional two-state binary to multiple values. This feature is specially interesting for hardware implementation of neural networks, where a high number of interconnections are required between the network layers. Using a multiple-valued number system can reduce the number of interconnections, and hence reduce the area and power consumption of the circuit.

The *Continuous Valued Number System* (CVNS) falls within the MVL arithmetic category. The CVNS and the basic concept of analog-digits were first introduced in [3], fundamental methods of addition arithmetic and array multiplication were developed in [4, 5], and circuit design issues were discussed in [6] and [7].

The radix-2 CVNS arithmetic has been very effective and used in development of a series of arithmetic units [8–11]. This class of the CVNS can interface with the digital inputs directly, and the conversion process between the two number systems is very simple. The radix-2 CVNS is also used for implementation of mixed-signal multipliers [12]. The multiplier is useful for implementation of high density CVNS neural networks, where a high number of adders and multipliers are required. The CVNS multiplier is compact, area efficient, and its operations are verified under different environmental conditions.

B. Zamanlooy • M. Mirhassani (✉)
Analog and Mixed Signal Research Group, Electrical and Computer Engineering, University of Windsor, Windsor, ON, Canada
e-mail: mitramir@uwindsor.ca

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_7

In Sect. 7.2, the CVNS theory is briefly reviewed, describing the original CVNS digit formation, addition, and conversion methods. Section 7.3 presents the modified CVNS adder when inputs are in the binary format for radix-2 CVNS addition. Section 7.4 describes the alternative truncated addition method as opposed to the original idea of the CVNS addition. Complexity analysis of the CVNS adders is summarized in Sect. 7.5. The original CVNS multiplication algorithm is reviewed in Sect. 7.6, with the alternative method shown in Sect. 7.7. In the last section implementation of the multiplier is presented.

7.2 Continuous Valued Number System

Continuous Valued Number System is an analog-digit representation and arithmetic system. In this number system arithmetic operations are performed by applying digit-level modular reduction operations on continuous real values.

7.2.1 CVNS Analog-Digits

Any continuous, real value such as $|x| \leq M$ from a number system with radix B can be mapped to a set of the CVNS analog-digits in radix-β. A CVNS value, $(\!(x)\!)$, can be formed as an ensemble of analog-digits, $(\!(x)\!)_i$, and can be written as a vector as follows:

$$(\!(x)\!) \Rightarrow \{(\!(x)\!)_n, (\!(x)\!)_{n-1}, \ldots, (\!(x)\!)_0 | (\!(x)\!)_{-1}, \ldots, (\!(x)\!)_{-k}\} \tag{7.1}$$

where $-k \leq i \leq n$ represents the indices of the CVNS digits, and the bar (|) represents the radix point.

There are two general methods for obtaining the CVNS digits [3]. A series method, *Cascade Digit Generation*, which finds the digits one by one. In this method, the lower index digits are generated from the higher order digits in their set.

The second method is a parallel digit generation method called *Modular Reduction Digit Generation*, where correction all digits are obtained from the original value of x. Both methods of digit generation are described in the next section.

7.2.2 Cascade Digit Generation

It is assumed that the maximum representable ranges in both number systems are equal, which leads to:

$$B^{m+1} = \beta^{n+1} = M \tag{7.2}$$

where m and n are the maximum index values of the binary and the CVNS number systems, respectively, and B is the positional number system radix.

A real value such as x can be converted into a set of CVNS digits by applying the series method, obtaining the most significant digit as follows [3]:

$$(\!(x)\!)_n = \frac{x}{M} \cdot \beta \tag{7.3}$$

with an associated integer, $\text{int}((\!(x)\!)_n) = \lfloor (\!(x)\!)_n \rfloor$.

The rest of the CVNS digits are computed as follows:

$$(\!(x)\!)_{i-1} = \left((\!(x)\!)_i - \lfloor (\!(x)\!)_i \rfloor\right) \cdot \beta \tag{7.4}$$

where $-k \leq i < n$

An important property of the CVNS number system is that within a set, higher index digits have higher volume of information relating to the magnitude of the original value compared to the lower index digits. Moreover, lower index digits can be derived from the higher index digits. All the digits in the set share and overlap information, which can be used for digit error detection and correction. To clarify the concept of analog-digits, a numerical example is presented here.

Example. Radix-10 CVNS digits of an arbitrary value such as $x = 78.324 < 100$ are obtained using the cascade method. The value of $n = 1$ is obtained based on the condition in (7.2). The most significant digit is equal to:

$$(\!(x)\!)_1 = \frac{78.324}{100} \cdot 10 = 7.8324$$

The rest of the CVNS digits are obtained as follows:

$$\begin{aligned}
(\!(x)\!)_0 &= \left((\!(x)\!)_1 - \lfloor (\!(x)\!)_1 \rfloor\right) \cdot \beta = (7.8324 - 7) \cdot 10 = 8.324 \\
(\!(x)\!)_{-1} &= \left((\!(x)\!)_0 - \lfloor (\!(x)\!)_0 \rfloor\right) \cdot \beta = (8.324 - 8) \cdot 10 = 3.24 \\
(\!(x)\!)_{-2} &= \left((\!(x)\!)_{-1} - \lfloor (\!(x)\!)_{-1} \rfloor\right) \cdot \beta = (3.24 - 3) \cdot 10 = 2.4 \\
(\!(x)\!)_{-3} &= \left((\!(x)\!)_{-2} - \lfloor (\!(x)\!)_{-2} \rfloor\right) \cdot \beta = (2.4 - 2) \cdot 10 = 4
\end{aligned}$$

The second method is a parallel technique. While the series method requires less resources, any error generated in the higher order digits can be transferred to lower order digits. The parallel method requires more resources; however, each digit is generated from the original value, and errors do not spread in the digits set.

7.2.3 Modular Digit Generation

All of the analog-digits of a set can be obtained by applying modular reduction operation as follows [3]:

$$((x))_i = \left(\frac{x}{M} \cdot \beta^{n-i+1}\right) \bmod \beta \tag{7.5}$$

where mod is the modular reduction operation on any real value such that $(a) \bmod \beta = a - I \times \beta$, and $0 \leq (a) \bmod \beta < \beta$, and I is an integer.

Example. As an example, CVNS digits for $x = 78.324 < 100$ using the modular reduction method are obtained as follows:

$$((x))_1 = \left(\frac{78.324}{100} \cdot 10\right) \bmod 10 = 7.8324$$

$$((x))_0 = \left(\frac{78.324}{100} \cdot 100\right) \bmod 10 = 8.324$$

$$((x))_{-1} = \left(\frac{78.324}{100} \cdot 10^3\right) \bmod 10 = 3.24$$

$$((x))_{-2} = \left(\frac{78.324}{100} \cdot 10^4\right) \bmod 10 = 2.4$$

$$((x))_{-3} = \left(\frac{78.324}{100} \cdot 10^5\right) \bmod 10 = 4$$

The choice of which method to apply is determined by mathematical convenience or by the hardware implementation requirements.

7.2.4 CVNS Addition

In its most basic and complete format, CVNS addition is performed independently on each of the CVNS analog-digits. Considering two values $x, y < |M/2|$, the CVNS digits of the summation outcome, $((z))$, are obtained by column-wise addition of analog-digits as follows [3]:

$$((z))_i = \left(((x))_i + ((y))_i\right) \bmod \beta \tag{7.6}$$

The addition of the CVNS values is by summation of the columns without intercommunication. The modular reduction operation after the addition prevents the digits exceeding the radix value in case of addition overflow.

Table 7.1 CVNS addition between two arbitrary values

i	2	1	0	−1	−2
$((x))_i$	0.4834	4.834	8.34	3.4	4
$((y))_i$	0.3289	3.289	2.89	8.9	9
$((z))_i$	0.8123	8.123	1.23	2.3	3

To demonstrate the addition operation, an example is presented in Table 7.1. The CVNS addition of two arbitrary values $x = 48.34$ and $y = 32.89$ is performed with $\beta = 10$.

In the next section, radix-2 CVNS is reviewed. This class has the flexibility of interfacing with digital values directly.

7.3 Addition of Two Binary Operands in CVNS

If the radix of the CVNS value is chosen to be equal to 2 or 2^{ν} radix, the digit generation between binary and CVNS and addition can be simplified extensively.

7.3.1 Binary to CVNS Conversion

Binary values can be converted into radix-2 CVNS representation quite easily if the CVNS radix is chosen as follows:

$$\beta = 2^{\nu} \tag{7.7}$$

with integer $\nu \geq 1$.

Any quantized value which is the weighted sum of binary digits can be represented as follows:

$$x = \pm \sum_{i=0}^{m} x_i \cdot 2^i \tag{7.8}$$

where x_i represents the $m + 1$ binary digits.

By applying the given expressions for the CVNS digit generations, the highest index analog-digit of the corresponding CVNS number is equal to:

$$((x))_n = \sum_{i=0}^{m} x_i \cdot 2^{i+\nu-m-1} \tag{7.9}$$

The associated integer part of the MID is the partial sum of the digits with $2^{i+\nu-m-1} \geq 1$, which is satisfied for the index range $m+1-\nu \leq i \leq n$, and can be obtained as follows:

$$\lfloor (\!(x)\!)_n \rfloor = \sum_{i=1-\nu}^{0} x_{m+i} \cdot 2^{i+\nu-1} \tag{7.10}$$

The next CVNS digit and its associated integer are equal to:

$$(\!(x)\!)_{n-1} = \sum_{i=0}^{m-\nu} x_i \cdot 2^{i+2\nu-m-1} \tag{7.11}$$

$$\lfloor (\!(x)\!)_{n-1} \rfloor = \sum_{i=1-\nu}^{0} x_{m+i-\nu} \cdot 2^{i+\nu-1} \tag{7.12}$$

This process is used for obtaining all of the remaining CVNS digits. In general:

$$(\!(x)\!)_{n-j} = \sum_{i=0}^{m-j\nu} x_i \cdot 2^{i+(1+j)\nu-m-1} \tag{7.13}$$

The conversion process to the CVNS representation directly applies to binary digit values (in case of binary inputs, 0 or 1), which means less hardware and a faster conversion process. The required number of the CVNS digits under the conditions in (7.2) and (7.7) is:

$$n = \left\lceil (m+1)\frac{\log 2}{\log \beta} \right\rceil - 1 = \left\lceil \frac{m+1}{\nu} \right\rceil - 1 \tag{7.14}$$

7.3.2 Addition

The addition of two CVNS values can be easily performed by using (7.6). The digit wise addition of the two CVNS values is obtained by the independent addition of each of the corresponding CVNS digits as presented below:

$$(\!(z)\!)_{n-j} = \left((\!(x)\!)_{n-j} + (\!(y)\!)_{n-j}\right) \bmod \beta \tag{7.15}$$

$$= \left(\sum_{i=0}^{m-\nu j} (x_i + y_i) 2^{i+(1+j)\nu-m-1} \right) \bmod \beta \tag{7.16}$$

The analog-digit with the highest index has theoretically all the information about the number. However, the implementation circuitry is analog, therefore it is required to add all of the digits to preserve the accuracy of outcome.

There is an error correction mechanism in CVNS, which is able to detect and correct the digits [6, 8]. This multiple-digit CVNS correction procedure is referred to as the *Reverse Evolution* [3], which examines the digit values starting from the lower index analog-digits and is able to detect and remove multiple errors. However, the procedure introduces delays into the addition process and requires more hardware.

To keep the process of addition accurate while it is implemented by analog circuit, it is suggested to generate the CVNS digits by a partitioning method. To do this, input binary digits are partitioned into smaller groups. Therefore, analog circuits process limited number of digital input each time. This not only relaxes the target requirements of analog circuits, but also reduces the complexity of analog circuits. This method performs the addition in a truncated manner, which is fully explained in next section.

7.3.3 CVNS to Binary Conversion

The result of the CVNS addition is converted back into the target binary format. Although at the input of the CVNS adder, a group of binary digits are used to form CVNS digits, at the output each CVNS digit is only equivalent to ν binary digits.

If the CVNS radix is chosen higher than two ($\nu > 1$), each CVNS digit is translated into ν binary digits. For example, if the CVNS radix is chosen to be 4, then each CVNS digit is equivalent to 2 binary digits. Converting each of the CVNS digits back into binary format is as follows:

$$z_{(n-j)\nu-i+1} = \begin{cases} 1 & \left(\beta^i (\!(z)\!)_{n-j}\right) \bmod \beta > \beta/2 \\ 0 & \left(\beta^i (\!(z)\!)_{n-j}\right) \bmod \beta \leq \beta/2 \end{cases} \tag{7.17}$$

where $0 \leq j \leq n$ and $0 \leq i \leq \nu$.

7.4 Truncated Addition

In the CVNS digit ensemble, the digit with the highest index interfaces with all of the input digits circuitry, in order to acquire the full information. Lower index digits, however, require less data. To facilitate the implementation of such a system, the MID is truncated in the addition process. In general the CVNS digits are truncated as long as the approximation does not affect the outcome of the final (integer)

computation; hence, the previous need for an error correction process after each two additions is eliminated. This in turn reduces the delay and overhead circuitry for the CVNS addition.

To do this, conversion from binary to the CVNS is performed over a group of digits, and not on all of the digits at once. The input binary digits are divided into smaller groups, independent of the input word length. The group size is determined by the implementation requirements and the technology of implementation and is denoted by φ with $\varphi > 1$, and is limited by the maximum reliable resolution of the analog implementation environment.

Considering the limited resolution of the analog environment, summation term in (7.15) has to be split into two terms: the first term contains φ digits of each input, while the second term is truncated to provide a suitable approximation level for the addition. If the truncation method is applied, (7.15) is modified to [13]:

$$(\!(z)\!)_{n-j} = \left(\sum_{i=m-\nu j-(\varphi-1)}^{m-\nu j} (x_i + y_i) 2^{i+(1+\nu)j-m-1} + Tr_{n-j} \right) \bmod \beta \tag{7.18}$$

where Tr_{n-j} is called the truncation signal.

The Tr_{n-j} is generated by binary digits with indices from 0 to $m - \nu j - \varphi - 2$. If $\varphi \leq m - j\nu - (\varphi - 2)$, these remaining digits should also be grouped into smaller sizes. Two methods for processing these binary digits within the limited resolution of the analog environment are developed. These methods follow the same logic, but differ on how binary digits are partitioned and processed.

Equation (7.15) shows the relation between the binary and the CVNS digits with the maximum redundancy among the digits. Any CVNS digit that interfaces with more than φ binary digits can be split into several groups. This means that the lower index CVNS digits that do not contain more than one group of data are not altered by truncation. These are digits with indices from 0 up to $\lceil \frac{\varphi}{\nu} \rceil - 1$ which interface with φ or less binary digits. Truncation starts for digits with higher indices.

To generate the addition outcome, $(\!(z)\!)_{n-j}$, it is required to form $\left\lceil \frac{m+1-j\nu}{\varphi} \right\rceil$ groups of binary digits. Then the outcome of each of these groups is used to determine the truncation signal, Tr_{n-j}, for that digit. Depending on which truncation scheme is used, some of groups are redundant, and are shared by a few other CVNS digits. The total number of all of the groups, redundant and non-redundant, in the adder is:

$$\sum_{j=0}^{n} \left\lceil \frac{m+1-j\nu}{\varphi} \right\rceil \tag{7.19}$$

The above expression reflects the upper boundary of the required groups for full redundancy. However since some of the groups as will be shown are redundant, to save area, some groups are used for multiple CVNS digits.

Two main methods are proposed for generating groups, whose outcome changes the structural format of the CVNS adder based on the different connectivity require-

ments. These methods are called *Sliding Groups* and *Uniform Groups*, and differ from each other by how binary inputs digits are grouped within each CVNS digit.

7.4.1 Sliding Groups

This partitioning scheme is close to the partitioning scheme used in the previous CVNS adder design [8]. The difference between the two is that the data from the lower significant binary digits is not ignored in the proposed addition scheme. The starting point for generating groups is always from the most significant binary digit. Figure 7.1 depicts details of this method for a general case.

The gray block indicates a group of digits that are used in the addition based on Eq. (7.18). These blocks receive the information from the rest of the groups. For example, MID requires information of all of the input binary digits. To facilitate reliable addition between the MID of the input operands, all of the input digits with indices from 0 to $n - j$ are divided into smaller groups. By partitioning the binary input in groups of length φ, input data is processed in fixed lengths, and the maximum required resolution of each group is equivalent to φ bits. The outcomes of each group are two signals that are analogues to the carry-propagate and carry-generate signals of the fast binary adders.

Fig. 7.1 Binary input bits partitioning in the CVNS digits based on the sliding groups, showing both redundant and non-redundant groups, where $k = m - \left\lceil \frac{m+1-\varphi}{\varphi} \right\rceil \varphi$ and $n \geq j \geq 0$

By choosing φ carefully, information generated in the groups of the lower informed digits can be reused for the more informed digits, which in turn reduces the hardware redundancy. This requirements can be met when:

$$\varphi = j\nu \tag{7.20}$$

where j is an integer larger than 2.

To reduce the area and to optimally reuse the generated data, (7.20) provides the best results. The first repetition starts when $m-2\varphi+1 = m-j\nu-\varphi+1$, which can be simplified into $\varphi = j\nu$. In this case it could be ensured that data of some of the groups is reused. This condition is highly effective in reducing the gate redundancy. The results and general expressions in this section are for general case, and can be made simpler if this condition is used.

In this method, information of only CVNS digits with indices from $\left(n - \left\lceil \frac{\varphi}{\nu} \right\rceil + 1\right)$ to n can generate all of the required information, which means that all of the other digits can reuse the data generated within these digits. The group method reduces the digit-level redundancy among the digits which in turn results in reduced gate-level redundancy at the implementation level. However, full information of the input binary digits required for proper addition is kept the same as the original CVNS addition [3] as expressed by Eq. (7.15). The total number of redundant groups which can be reduced from the total groups given in (7.19) is:

$$\sum_{i=\left\lceil \frac{\varphi}{\nu} \right\rceil}^{n-\left\lceil \frac{\varphi}{\nu} \right\rceil} \left\lceil \frac{m-(n-i)\nu-\varphi+1}{\varphi} \right\rceil \tag{7.21}$$

Figure 7.1 can be used as a map to properly distribute the input binary information within the CVNS digits, and to decide on which groups to generate. Outcomes of each group are:

$$gt_{(p)(q)} = \begin{cases} 1 & \sum_{i=u_{ts}(j)-\varphi+1}^{u_{ts}(j)} f_{ts}(i) \geq 2 \\ 0 & \text{otherwise} \end{cases} \tag{7.22}$$

$$rt_{(p)(q)} = \begin{cases} 1 & \sum_{i=u_{ts}(j)-\varphi+1}^{u_{ts}(j)} f_{ts}(i) \geq 2-2^{-\varphi} \\ 0 & \text{otherwise} \end{cases} \tag{7.23}$$

where

$$f_{ts}(i) = (x_i + y_i)2^{i-m+(n+1-j)\nu+q\varphi-1} \tag{7.24}$$

$$u_{ts}(j) = m-(n-j)\nu-q\varphi \tag{7.25}$$

and p, q, and j are integers in the range of $1 \leq p \leq \lceil \frac{\varphi}{\nu} \rceil$, and $\left(m - \lceil \frac{\varphi}{\nu} \rceil + 1\right) \leq j \leq n$, and $1 \leq q \leq \left\lceil \frac{m-(n-j)\nu-\varphi+1}{\varphi} \right\rceil$ correspondingly, and p is dependent to j and found as follows:

$$n - j = t \left\lceil \frac{\varphi}{\nu} \right\rceil + o$$
$$p = \left\lceil \frac{\varphi}{\nu} \right\rceil - o \tag{7.26}$$

where integers t and o are the modules and reminder values, respectively.

The signals gt and rt, which are the outputs of the first layer of the adder, appear in the CVNS digits periodically, with the periodic length of $\lceil \frac{\varphi}{\nu} \rceil$. The second layer of the CVNS adder combines these information to generate the truncation signal, Tr_{n-j}. The truncation signal Tr_{n-j} used in (7.18) is obtained recursively from the groups and is equal to:

$$Tr_{n-j} = \beta^{\varphi/\nu-1}\Big(gt_{(p)(q)} + rt_{(p)(q)}.gt_{(p)(q-1)} + rt_{(p)(q)}.rt_{(p)(q-1)}.gt_{(p)(q-2)} + \cdots + rt_{(p)(q)}.rt_{(p)(q-1)} \ldots rt_{(p)(1)}.C_{in}\Big) \tag{7.27}$$

where $n \geq j \geq 0$, $1 \leq k \leq \left\lceil \frac{m-(n-j)\nu-\varphi+1}{\varphi} \right\rceil$ and C_{in} is the carry to the adder and p is found from (7.26).

There are at most $\left\lceil \frac{m+1}{\varphi} \right\rceil - 1$ addition terms in Eq. (7.27), while the last term requires $\left\lceil \frac{m}{\varphi} \right\rceil$ multiplications for computing each Tr_{n-j}. Due to the limitations in gate fan-in of the implementation medium it may be required that Tr_{n-j} to be computed in more than two stages. After forming the truncation signal, it is going to be the only signal per each CVNS digit that passed from the second layer to the last layer of the adder. The CVNS addition does not require to compute carry-propagate or carry-ready for individual bits.

At the last layer, the last required groups within each CVNS digit are computed and addition is performed based on Eq. (7.18).

To clarify this scheme, a numerical example is shown in Fig. 7.2, where groups for radix-4 addition of two 16-bit binary values of $1001, 0011, 1111, 0111$ and $0011, 0101, 1110, 1001$ are shown. Since the radix of the CVNS representation is higher than the radix of binary representation, 8 CVNS digits are required. Group size is considered equal to 4, and truncation signals for each group are computed in Table 7.2 where details of addition for each of CVNS digits are shown, and binary outcome of addition is obtained in the last two columns of the table.

An alternative method for partitioning the binary input digits is given next, which is more efficient in terms of required groups.

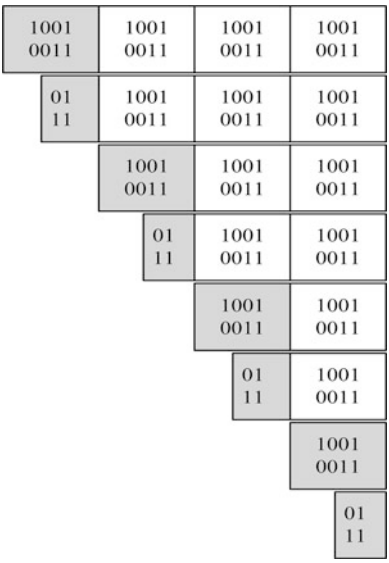

Fig. 7.2 A numerical example showing the partitioning scheme of input binary digits based on the sliding groups. Note as indicated before how some of the groups are redundant, and are not required to actually be generated during the implementation

Table 7.2 CVNS truncated addition between two arbitrary values based on the sliding group partitioning of the input digits

i	Tr_i	$((z))_i$	z_{2i}	z_{2i-1}
7	0	$\left[(1+0)+2^{-1}(0+0)+2^{-2}(0+1)+2^{-3}(1+1)+Tr_7\right] \bmod 2 = (1.5) \bmod 2 = 1.5$	1	1
6	0	$\left[(0+1)+2^{-1}(1+1)+2^{-2}(0+0)+2^{-3}(0+1)+Tr_6\right] \bmod 2 = (2.125) \bmod 2 = 0.125$	0	0
5	0.125	$\left[(0+0)+2^{-1}(0+1)+2^{-2}(1+0)+2^{-3}(1+1)+Tr_5\right] \bmod 2 = (1.125) \bmod 2 = 2.125$	1	0
4	0.5	$\left[(1+0)+2^{-1}(1+1)+2^{-2}(1+1)+2^{-3}(1+1)+Tr_4\right] \bmod 2 = (2.875) \bmod 2 = 0.875$	0	1
3	0.125	$\left[(1+1)+2^{-1}(0+0)+2^{-2}(1+0)+2^{-3}(1+0)+Tr_3\right] \bmod 2 = (3.75) \bmod 2 = 1.75$	1	1
2	0.5	$\left[(1+1)+2^{-1}(1+0)+2^{-2}(0+1)+2^{-3}(1+0)+Tr_2\right] \bmod = (3) \bmod 2 = 1$	1	0
1	n/a	$\left[(0+1)+2^{-1}(1+0)+2^{-2}(1+0)+2^{-3}(1+1)\right] \bmod 2 = (2) \bmod 2 = 0$	0	0
0	n/a	$\left[(1+0)+2^{-1}(0+0)\right] \bmod 2 = (2) \bmod 2 = 0$	0	0

7.4.2 Uniform Groups

In this method, as shown in Fig. 7.3, binary digit partitioning starts from the least significant binary digit, ensuring that all of the generated information is used efficiently. The number of redundant groups is higher in this format, which means that the required groups is lower as follows:

$$G = -\left\lceil \frac{m+1}{\varphi} \right\rceil + \sum_{i=\left\lceil \frac{\varphi}{v} \right\rceil}^{n} \left\lceil \frac{m-(n-i)v-\varphi}{\varphi} \right\rceil \tag{7.28}$$

The values of the gt and rt signals are:

$$gt_{(p)} = \begin{cases} 1 & \sum_{i=u_{tu}(p)-\varphi+1}^{u_{tu}(p)} f_{tu}(i) \geq 2 \\ 0 & \text{otherwise} \end{cases} \tag{7.29}$$

$$rt_{(p)} = \begin{cases} 1 & \sum_{i=u_{tu}(p)-\varphi+1}^{u_{tu}(p)} f_{tu}(i) \geq 2-2^{-\varphi} \\ 0 & \text{otherwise} \end{cases} \tag{7.30}$$

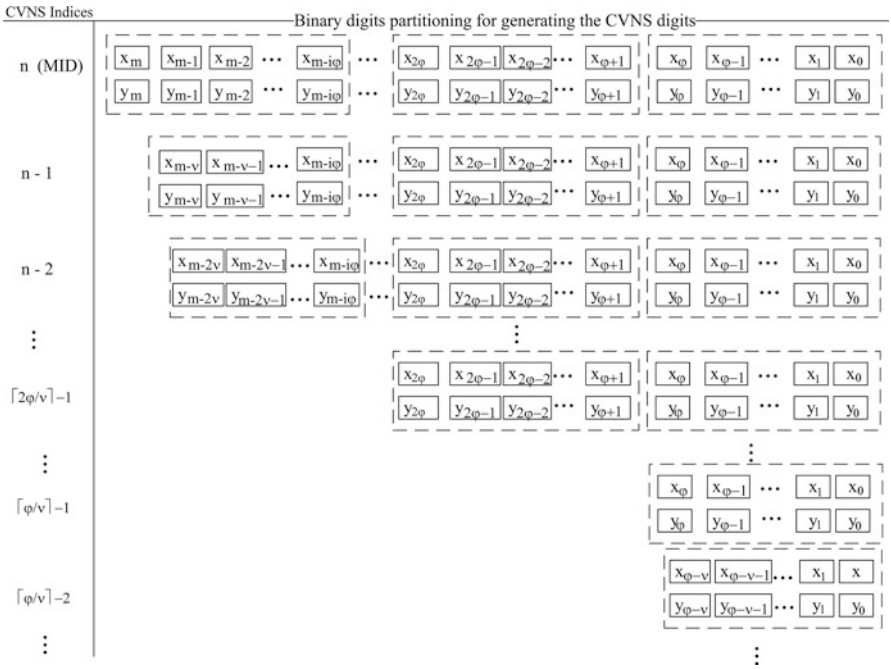

Fig. 7.3 Binary input bits partitioning in the CVNS digits based on the "uniform groups" showing the redundant and non-redundant groups

where

$$f_{tu}(i) = (x_i + y_i)2^{i-p\varphi-1} \tag{7.31}$$

$$u_{tu}(p) = p\varphi - 1 \tag{7.32}$$

and p is an integer in the range of $1 \leq p \leq \left\lceil \frac{m-\varphi+1}{\varphi} \right\rceil$.

The binary outputs of these units are used in the second layer of the CVNS adder for generating the truncation signal. The fact that the truncation signals are binary provides some extra flexibility in the choice of circuitry (analog or digital). The second layer of the CVNS adder can be implemented with analog modular blocks, as well as regular digital blocks. The choice depends on the design criteria and appropriate technology parameters. For example, low-noise applications may take advantage of analog blocks for their inherent low-noise properties.

The last group in each of the CVNS digits contains $(m - j\nu)$ down to the $\left(\left(\left\lceil \frac{m-j\nu}{\varphi} \right\rceil - 1\right)\varphi\right)$th binary input digits. The truncated addition result is equal to:

$$(\!(z)\!)_{n-j} = \left(\sum_{i=\left(\left\lceil \frac{m-j\nu}{\varphi} \right\rceil - 1\right)\varphi}^{m-\nu j} (x_i + y_i)2^{i+(1+\nu)j-m-1} + Tr_{n-j} \right) \bmod \beta \tag{7.33}$$

where

$$\begin{aligned} Tr_{n-j} = 2^{k+j\nu-m-1} \cdot \Big(& gt_{(i)} + rt_{(i)} \cdot gt_{(i-1)} + rt_{(i)} \cdot rt_{(i-1)} . gt_{(i-2)} \\ & + \cdots + rt_{(i)} \cdot rt_{(i-1)} \ldots rt_{(2)} \cdot rt_{(1)} \cdot C_{in} \Big) \end{aligned} \tag{7.34}$$

and $k = \left(\left\lceil \frac{m-j\nu+1}{\varphi} \right\rceil - 1\right)\varphi$, and $1 \leq i \leq \left\lceil \frac{m-j\nu-\varphi+1}{\varphi} \right\rceil$.

It should be mentioned here that the main difference between expressions (7.18) and (7.34) is due to the fact that in this method groups that perform the truncation addition do not have the same number of binary digits. This partitioning method results in a multi-layer CVNS adder which consists mainly of modular reduction gates. For high numbers of input bits, the CVNS adder may require one or more layers for computing the truncation signal, due to fan-in limitations based on the implementation techniques used to compute Eq. (7.34). In fact, when $m \leq 2\varphi - 1$, the CVNS adder is implemented in only three layers.

If the number of *gt* and *rt* signals to the second layer is higher than the maximum fan-in of the gates, the term Tr_{n-j} has to be split into smaller units. Adding more layers increases the delay, which is the same for all forms of two operand adders. The results of Eq. (7.34) are now ready to be converted back into binary form.

To clarify this scheme binary digit partitioning and details of additions are shown in Fig. 7.4 and Table 7.3, respectively. In this case, only three groups are

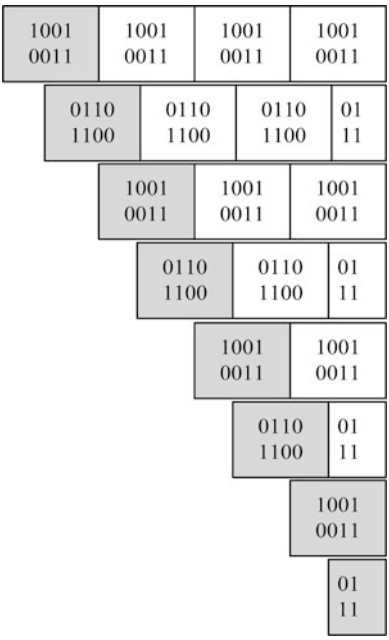

Fig. 7.4 A numerical example showing the partitioning scheme of input binary digits based on the uniform groups. Note as indicated before in comparison to sliding groups more redundancy exists here, which reduces the number of required gates during the implementation

Table 7.3 CVNS truncated addition between two arbitrary values based on the uniform group partitioning of the input digits

i	Tr_i	$((z))_i$	z_{2i}	z_{2i-1}
7	0	$[(1+0)+2^{-1}(0+0)+2^{-2}(0+1)+2^{-3}(1+1)+Tr_7]$ mod $2=(1.5)$ mod $2=1.5$	1	1
6	0	$[(0+1)+2^{-1}(1+1)+Tr_6]$ mod $2=(2)$ mod $2=0$	0	0
5	0.125	$[(0+0)+2^{-1}(0+1)+2^{-2}(1+0)+2^{-3}(1+1)+Tr_5]$ mod $2=(1.125)$ mod $2=2.125$	1	0
4	0.5	$[(1+0)+2^{-1}(1+1)+Tr_4]$ mod $2=(2.5)$ mod $2=0.5$	0	1
3	0.125	$[(1+1)+2^{-1}(0+0)+2^{-2}(1+0)+2^{-3}(1+0)+Tr_3]$ mod $2=(3.75)$ mod $2=1.75$	1	1
2	0.5	$[(1+1)+2^{-1}(1+0)+Tr_2]$ mod $=(3)$ mod $2=1$	1	0
1	n/a	$[(0+1)+2^{-1}(1+0)+2^{-2}(1+0)+2^{-3}(1+1)]$ mod $2=(2)$ mod $2=0$	0	0
0	n/a	$[(1+0)+2^{-1}(0+0)]$ mod $2=(2)$ mod $2=0$	0	0

generated, where each group generates two output signals, *gt* and *rt*. These signals are combined to generate the required three *Tr* signals. Every two CVNS digits requires the same signal; for example, CVNS digits with index 7 and 6 have the same truncation signal. In comparison, for the same example with sliding groups partitioning, 6 groups are required, which in turn means 12 signals are required to combine, and generate 6 truncation signals.

Using the above equations an efficient CVNS addition process is defined, and a method for interfacing between the conventional positional number system and the CVNS system is developed. In the next section the processing element count and system complexity of the CVNS adder are computed and compared with both previously published CVNS adders [3, 8].

7.5 Processing Element Count

The CVNS adder can reduce the number of interconnections and achieve area and power savings [14]. In this section, the basic CVNS adder (ADD_b) [3], the Group adder (ADD_g) [8], and our new methods, truncation based on the uniform method (ADD_{tu}) and sliding method (ADD_{ts}), are compared. Comparisons are also made between the system complexities in terms of number of required gates, fan-in and fan-out of the system, system resolution, and finally gate and system sizes [14].

The original CVNS method of addition [3], ADD_b, does not introduce any approximations into the system. This means that the MID requires a fan-in of $2m$ and a resolution of $r_b = B^{-m}$. For a high number of input bits, such a high fan-in and resolution in an analog implementation is not feasible. Consequently, the group method, ADD_g, offers a trade-off in terms of reducing the fan-in of the gates and the maximum required resolution at the cost of increased delay.

The ADD_g [8] reduces the maximum fan-in to $2tu$ and the required resolution to $r_g = B^{-\varphi}$.

In both truncation methods, ADD_{ts} and ADD_{tu}, a maximum fan-in of $2\varphi + 1$ is required, and the required resolution is equal to $r_{ts,tu} = B^{-\varphi}$. Therefore, the group adder and the truncated adder have similar implementation requirements.

The ADD_b adds two radix-2 values in four stages of binary to CVNS conversion, modulus addition, CVNS to digital conversion, and reverse evolution to remove the uncertainty imposed by the implementation medium. The size of the gates directly increases with the gate fan-in and maximum weight in the adder. The conversion unit in the CVNS adder is the most area-consuming unit. The number of required conversion units is equal to $G_{(\text{conv}-b)} = \lceil \frac{m+1}{\nu} \rceil$. For ADD_g, this is equal to $G_{(\text{conv}-g)} = \lceil \frac{m+1}{\nu} \rceil$.

The ADD_{ts} and ADD_{tu} require:

$$G_{(\text{conv}-ts)} = \left\lceil \frac{m+1}{\nu} \right\rceil \times \left\lceil \frac{m+1}{\varphi} \right\rceil \tag{7.35}$$

$$G_{(\text{conv}-tu)} = \left\lceil \frac{m+1}{\nu} \right\rceil + \left\lceil \frac{m+1}{\varphi} \right\rceil \tag{7.36}$$

The number of gates can be used as an indicator of the size that a system may occupy. However, for this assumption to be valid, transistor sizing should be very uniform, and gate size should dominate interconnect wiring in the neighborhood of the gates. The analysis is accurate when the circuit is made up of uniform basic gates such as AND, OR, and Inverters. The theory fails, however, when applied to

the CVNS, since gate sizing is directly affected by the number of inputs and weight values for binary to CVNS conversion.

In [15] a new function is proposed for area estimation, which takes into account the transistor sizing. To estimate the area in a CVNS system, a new parameter is defined here which is close to the suggestion in [15], but much simpler, since the CVNS theory requires only positive weight values. The value of this parameter, s, is:

$$s = 2\sum_{i} B^{i} \tag{7.37}$$

In the CVNS adder, conversion units are the most area-consuming, and the value of the parameter, s, of each conversion gate for each of the CVNS digits is:

$$s_{(\mathrm{conv}-b_j)} = 2\sum_{i=0}^{m-j\nu} B^{m-i-(1+j)\nu+1} \tag{7.38}$$

The total for all of the $(n+1)$ CVNS digits is:

$$s_{(\mathrm{conv}-b)} = 2\sum_{j=0}^{n}\sum_{i=0}^{m-j\nu} B^{m-i-(1+j)\nu+1} \tag{7.39}$$

For the group adder, ADD_g, this parameter is:

$$s_{(\mathrm{conv}-g)} = 2\sum_{j=0}^{\lceil\frac{\varphi}{\nu}\rceil-1}\sum_{i=0}^{m-(n-j)\nu} B^{i} + 2\left(n-\left\lceil\frac{\varphi}{\nu}\right\rceil+1\right)\sum_{i=0}^{\varphi-1} B^{i} \tag{7.40}$$

The estimated area of the conversion units for the truncated-sliding group adder may be higher than that of the group adder, since in this method full information redundancy is obtained. The size parameters of conversion gate area of the ADD_{ts}, and ADD_{tu} are:

$$\begin{aligned} s_{(\mathrm{conv}-ts)} = {} & 2\sum_{j=n-\lceil\frac{\varphi}{\nu}\rceil+1}^{\ell}\left(\left\lceil\frac{m-(n-j)\nu-\varphi+1}{\varphi}\right\rceil-1\right)\sum_{i=0}^{\varphi-1} B^{i} \\ & +4\sum_{j=0}^{\lceil\frac{\varphi}{\nu}\rceil-1}\sum_{i=0}^{m-(n-j)\nu} B^{i} + 2\left(n-\left\lceil\frac{\varphi}{\nu}\right\rceil+1\right)\sum_{i=0}^{\varphi-1} B^{i} \end{aligned} \tag{7.41}$$

$$s_{(\mathrm{conv}-tu)} = 2\sum_{j=0}^{\lceil\frac{\varphi}{\nu}\rceil-1}\sum_{i=0}^{m-(n-j)\nu} B^{i} + 2\left(\left\lceil\frac{m+1}{\varphi}\right\rceil-1\right)\sum_{i=0}^{\varphi-1} B^{i}$$

$$+2\sum_{j=\lceil \frac{\varphi}{\nu} \rceil}^{n} \sum_{i=(l-1)\varphi}^{m-(n-j)\nu} B^{i-(l-1)\varphi} \tag{7.42}$$

where ℓ is defined as $\left\lceil \frac{m-(n-j)\nu+1}{\varphi} \right\rceil$.

At first glance, the sliding group adder area has higher gate area cost than the group adder, ADD_g. However, ADD_g requires an adjustment to the final results after each two operand addition, which requires additional units thus increasing the area and the delay proportional to the number of the CVNS digits. Based on the given expression in [3] each correction unit requires to compare two neighboring digits and perform the correction by scaling the more informed digit and comparing it with the adjacent less informed digit. Therefore, in total there are n correction units and size parameter, s_{RE}, which is used to evaluate the size of the reverse evolution process is $G_{(\mathrm{RE}-b,g)} = n$ and $s_{\mathrm{RE}} = n(B+1)$.

This increases the total area is required for the ADD_b and ADD_g implementations proportional to the number of CVNS digits. All three types of the CVNS adders require A/D conversion units, hence it is assumed that in comparing area these parts cancel each other out. Therefore, for area comparisons these parts are not included considering that similar techniques are used for all of them. The main difference between the $\mathrm{ADD}_{b,g}$ and $\mathrm{ADD}_{ts,tu}$ structures is the reverse evolution process.

Delay is another important issue for two operand addition. ADD_b is not feasible for analog implementation due to its high fan-in and resolution requirements. It does, however, allow a parallel architecture. The original CVNS adder, ADD_b, and the group adder, ADD_g, perform addition in four stages: digital to analog conversion, modular reduction operations, analog to digital conversion, and reverse evolution, where each stage adds its own delay into the signal path. Nevertheless, the final results are required to be adjusted. The main source of error for the AAD_b is the implementation medium, whereas the error in AAD_g is generated by removing the data overlap between the CVNS digits.

$$D_{b,g} = T_{D/A} + T_{\mathrm{mod}} + T_{A/D} + (n-1).T_{\mathrm{RE}} \tag{7.43}$$

where $T_{D/A}$ is the delay caused by the binary to the CVNS conversion, T_{mod} is the modulus gate delay, $T_{A/D}$ is the CVNS to binary conversion delay, and T_{RE} is the delay from the error correction process (reverse evolution delay).

On the other hand, ADD_{ts} and ADD_{tu} forms do not need to correct the results after each two additions, which drastically reduces the delay of the adder. The delay, D, is independent of the number of input bits into the system, and is equal to:

$$T_{ts,tu} = T_{D/A} + T_{\mathrm{mod}} + T_{A/D} \tag{7.44}$$

A case study for a 32-bit, two operand addition is presented in Table 7.4, using the expressions in this section as an example for all four types of the CVNS adders.

Table 7.4 Comparison between the four different types of the CVNS adder for a 32-bit radix-2 ($\beta = 2$ and $\nu = 1$) addition

Adder type	ADD_b [3]	ADD_g [8]	ADD_{ts}	ADD_{tu}
Fan-in	64	8	9	9
r	2^{32}	2^4	2^4	2^4
G_{conv}	32	32	60	39
s_{conv}	$4(2^{33} - 1)$	892	1664	444
G_{RE}	32	32	N/A	N/A
s_{RE}	96	96	N/A	N/A
$s_T = s_{\text{conv}} + s_{\text{RE}}$	34×10^9	988	1664	444
T	$\propto 34$	$\propto 34$	$\propto 3$	$\propto 3$

$\varphi = 4$ and expressions developed in this section are used

The size parameter, s, indicates that the size of the basic and group CVNS adders are higher, and both have greater delay, proportional to the number of input digits, than our new adder structures. For large word lengths, these approaches may not be feasible.

7.6 CVNS Multiplication Algorithm

In this section, previously developed CVNS multiplication algorithm [3] is briefly introduced. The absolute value of two multiplication operands X and Y using fixed-point number system with a radix of B can be shown as follows:

$$X = \sum_{i=-N_{fx}}^{N_{ix}-1} x_i \times B^i \tag{7.45}$$

$$Y = \sum_{i=-N_{fy}}^{N_{iy}-1} y_i \times B^i \tag{7.46}$$

where N_{ix} and N_{fx} are the number of integer and fractional digits of the multiplication operand X, respectively. N_{iy} and N_{fy} are the number of integer and fractional digits of the multiplication operand Y, respectively. x_i and y_i are the digits.

According to the algorithm developed in [3], the CVNS digit $((p))_m$ representing the multiplication result of two operands, X and Y, can be written in the following form:

$$((p))_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i ((x))_{m-i} \right) \bmod B \tag{7.47}$$

where $((x))_{m-i}$ is the CVNS digit representing the multiplication operand X in the CVNS format.

The algorithm developed in [3] is based on the assumption that the environment resolution is as high as the resolution required for multiplication. However, the resolution of analog circuits exploited for implementation of the developed algorithm is limited. This limits the resolution of the CVNS multiplication result provided by the algorithm developed in [3] to the environment resolution. To illustrate this, an example is provided.

Example. Finding the multiplication result of $X = 217$ and $Y = 49{,}084$ using the algorithm developed in [3] with a radix of two and environment resolution, φ, of four.

To perform the multiplication, analog-digits of $((x))$ representing $X = 217$ should be obtained. The digits are obtained using the truncated digit generation method similar to the addition method described in previous section. The set of CVNS digits representing the input X in an environment with resolution of φ is as follows:

$$((x))_m = \sum_{i=m-\varphi+1}^{m} x_i \times B^{i-m} \tag{7.48}$$

For this example, using Eq. (7.48), the CVNS analog-digit set is as follows:

$$\begin{aligned} ((x)) &= \{((x))_{10}, ((x))_9, \ldots, ((x))_1, ((x))_0\} \\ &= \{0.125, 0.375, 0.75, 1.625, 1.375, 0.75, 1.5, 1.125, 0.25, 0.5, 1\} \end{aligned} \tag{7.49}$$

However, the input Y is kept at its binary format and is equal to $Y = 1011111110111100$ in binary format.

Since the resolution required to represent the inputs X and Y are 8 and 16 bits, respectively, the multiplication result resolution is 24 bits. Considering that the environment is equal to four bits, each CVNS digit includes the information corresponding to four bits.

Therefore, the CVNS digit set $((p)) = \{((p))_{23}, ((p))_{19}, ((p))_{15}, ((p))_{11}, ((p))_7, ((p))_3\}$ can represent the output.

Using (7.47), the CVNS digit set $((p)) = \{0.875, 1.75, 0.5, 0.5, 0.625, 1.5\}$ represents the multiplication result in the CVNS format. However, the correct multiplication result in decimal format is equal to 10,651,228. Therefore, using (7.48), the CVNS digit set representing the output should be equal to $\{1.25, 0.25, 1, 0.75, 0.625, 1.5\}$. This example reveals that the multiplication algorithm developed in [3] fails to provide correct multiplication results in low-resolution environment. Therefore, a new CVNS multiplication algorithm is required.

7.7 CVNS Multiplication Algorithm in Low-Resolution Environment

It can be seen that an alternative CVNS multiplication algorithm for low-resolution environment [12] is required. The CVNS digit $(\!(z)\!)_m$ representing the multiplication result of two operands, X and Y, can be written in the following form [3]:

$$(\!(z)\!)_m = (\!(X \times Y)\!)_m = (X \times Y \times B^{-m}) \bmod B \tag{7.50}$$

If X and Y are replaced with their fixed-point representation as shown in (7.45) and (7.46), the (7.50) can be written in the following form:

$$(\!(z)\!)_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i \times B^i \sum_{j=-N_{fx}}^{N_{ix}-1} x_j \times B^j \times B^{-m} \right) \bmod B \tag{7.51}$$

One of the basic properties of mod B operation is that, if it is applied on values which are a multiple of B and greater than one, the outcome of this continuous modular reduction operation will be zero. This can be shown in the following form:

$$k \geq 0 \;\Rightarrow\; \left(B^k\, x_j\right) \bmod B = 0 \tag{7.52}$$

Using this basic property, Eq. (7.51) can be modified in the following form:

$$(\!(z)\!)_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i \sum_{j=-N_{fx}}^{m-i} x_j \times B^{j-(m-i)} \right) \bmod B \tag{7.53}$$

Equation (7.48) can be exploited to represent the $(\!(z)\!)_m$ in terms of the CVNS digits $(\!(x)\!)_m$ which represent the multiplication operand X in the CVNS format. In order to do this, (7.53) is modified in the following form:

$$\begin{aligned}(\!(z)\!)_m = \Bigg(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i \Bigg(& \sum_{j=m-i-\varphi+1}^{m-i} x_j \times B^{j-(m-i)} + B^{-\varphi} \sum_{j=m-i-2\varphi+1}^{m-i-\varphi} x_j \times B^{j-(m-i+\varphi))} \\ & + \cdots + B^{-(n_i+1)\varphi} \sum_{j=-N_{fx}}^{m-i-n_i\varphi} x_j \times B^{j-(m-i+n_i\varphi)} \Bigg) \Bigg) \bmod B \end{aligned} \tag{7.54}$$

Since each CVNS digit $(\!(x)\!)_m$ has φ terms, the value of n_i is found based on the following condition:

$$m - i - n_i\varphi + N_{fx} \leq \varphi - 1 \tag{7.55}$$

Therefore, n_i can be calculated using the following equation:

$$n_i = \left\lceil \frac{m + (N_{fx} + 1) - (i + \varphi)}{\varphi} \right\rceil \tag{7.56}$$

in which $\lceil\ \rceil$ is the ceiling function.

Using (7.48) and (7.54), $(\!(z)\!)_m$ is written as follows:

$$(\!(z)\!)_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i \big((\!(x)\!)_{m-i} + B^{-\varphi} (\!(x)\!)_{m-i+\varphi} + \cdots + B^{-(n_i+1)\varphi} (\!(x)\!)_{m-i+n_i\varphi)} \big) \right) \bmod B \tag{7.57}$$

Repeating the same procedure from (7.50) to (7.57) for $(\!(z)\!)_{m-\varphi}$, $(\!(z)\!)_{m-\varphi}$ can be calculated in the following form:

$$(\!(z)\!)_{m-\varphi} = C_{m-\varphi} \bmod B = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i \big((\!(x)\!)_{m-i+\varphi} + B^{-\varphi} (\!(x)\!)_{m-i+2\varphi} + \cdots + B^{-(n_i+1)\varphi} (\!(x)\!)_{m-i+n_i\varphi)} \big) \right) \bmod B \tag{7.58}$$

Using (7.54) and (7.58), we can write:

$$(\!(z)\!)_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i (\!(x)\!)_{m-i} + B^{-\varphi} C_{m-\varphi} \right) \bmod B \tag{7.59}$$

According to (7.48), the term $B^{-(\varphi-1)}$ is the term with the lowest resolution present in the CVNS digits. Therefore, only the terms greater than $B^{-(\varphi-1)}$ should be considered. Thus, (7.59) can be written in the following form:

$$(\!(z)\!)_m = \left(\sum_{i=-N_{fy}}^{N_{iy}-1} y_i (\!(x)\!)_{m-i} + B^{-(\varphi-1)} \left\lfloor \frac{C_{m-\varphi}}{B} \right\rfloor \right) \bmod B \tag{7.60}$$

where $\lfloor\ \rfloor$ is the floor function.

In (7.60), the CVNS output $(\!(z)\!)_m$ and the CVNS input $(\!(x)\!)_{m-i}$ have a resolution of φ. Therefore, the proposed algorithm is compatible with the environment resolution and provides accurate results in the low-resolution environment. To illustrate the effectiveness of the proposed algorithm, the example explored in the previous section is resolved using the proposed multiplication algorithm.

Example. Finding the multiplication result of $X = 217$ and $Y = 49{,}084$ using the proposed algorithm with a radix of two and environment resolution of four.

Since the X and Y are the same as example discussed in the previous section, the same CVNS digit set and binary number can be used to represent X and Y. Moreover, as discussed in the previous example, the multiplication result resolution is 24 bits. Considering $\varphi = 4$, the CVNS digit set of the output is equal to:

$$(\!(z)\!) = \{(\!(z)\!)_{23}, (\!(z)\!)_{19}, (\!(z)\!)_{15}, (\!(z)\!)_{11}, (\!(z)\!)_{7}, (\!(z)\!)_{3}\}$$

Using the modified algorithm the CVNS digit set of the output is equal to:

$$(\!(z)\!) = \{1.25, 0.25, 1, 0.75, 0.625, 1.5\}$$

As discussed in the previous example, the CVNS digit set representing the output should be equal to $\{1.25, 0.25, 1, 0.75, 0.625, 1.5\}$. Therefore, the proposed CVNS multiplication algorithm generates the correct multiplication result in low-resolution environment.

The multiplication operands X and Y in the proposed algorithm are represented by the analog CVNS digits $(\!(x)\!)_{m-i}$ and the digits y_i, respectively. Therefore, the proposed multiplication algorithm accepts the inputs X and Y in the analog and digital format, respectively. Thus, it may be a suitable candidate for VLSI implementation of high resolution mixed-signal synapse multipliers using low-resolution analog mixed-signal circuits.

7.8 VLSI Implementation of the CVNS Synapse Multiplier

In this section, VLSI implementation of the CVNS multiplication algorithm is discussed. The VLSI implementation is realized using TSMC CMOS 0.18 μm technology. A multiplier with a resolution of 16×8 bits is considered which is useful for applications such as neurochips with on-chip learning [16, 17]. The synaptic weight is stored on the digital registers. The output of the activation function which is the other input to the multiplier is in the CVNS format. Therefore, to use (7.60) for implementation of the multiplier, weights are denoted by y while the other input is denoted by the CVNS digits $(\!(x)\!)$.

As shown in [16], variation range of $(-8, 8)$ as synaptic weight range is adequate. Therefore, 3 integer bits are required to represent the weight variation range. Moreover, since the weights are signed variables, 1 bit is required for the sign bit. The 12 remaining bits are used to represent the fractional part of the weights.

Weights are stored in 2's complement format. To have the multiplication result in 2's complement format correctly, the sign bit extension is exploited. For the 16 by 8 bit multiplication, sign bit is extended by 8 bits to produce the correct result. Considering the sign extension, both N_{iy} and N_{fy} parameters in (7.60) are equal to 12.

The output of the sigmoid activation function is always positive and less than one. Therefore, all of the 8 bits representing the output of the sigmoid activation function are fractional digits. This results in $N_{ix} = 0$ and $N_{fx} = 8$.

Assuming an environment resolution of 4 ($\varphi = 4$) and a radix of 2, (7.60) can be written in the following form:

$$(\!(z)\!)_m = \left(\sum_{i=-12}^{11} y_i (\!(x)\!)_{m-i} + 2^{-3} \left\lfloor \frac{C_{m-4}}{2} \right\rfloor \right) \bmod 2 \tag{7.61}$$

In the equation above, due to the sign extension, y_4 to y_{11} are equal to each other and have the same value as y_3 which is the sign bit of the synaptic weight.

The multiplication result resolution for a 16×8 multiplier is 24 bits. Since the variation range of the inputs to the multiplier are $(-8, 8)$ and $(0, 1)$, the output range of the multiplier is $(-8, 8)$. Therefore, 3 bits are required to represent the integer part of the multiplication result while 1 bit represents the sign bit. The remaining 20 bits represent the fractional part of the multiplication result.

Considering that $\varphi = 4$, six CVNS digits are used to represent the 24-bits multiplication result. Therefore, the CVNS digit set for presenting the output is equal to:

$$(\!(z)\!) = \{(\!(z)\!)_3, (\!(z)\!)_{-1}, (\!(z)\!)_{-5}, (\!(z)\!)_{-9}, (\!(z)\!)_{-13}, (\!(z)\!)_{-17}\}$$

Current-mode circuits are used to realize (7.61). In the VLSI implementation of (7.61), 8 μA is considered indicator of 1. Therefore, (7.61) can be written in the following form:

$$(\!(z)\!)_m = \left(\sum_{i=-12}^{11} y_i (\!(x)\!)_{m-i} + 1\,\mu\text{A} \times \left\lfloor \frac{C_{m-4}}{16\,\mu\text{A}} \right\rfloor \right) \bmod 16\,\mu\text{A} \tag{7.62}$$

Considering that the addition in the current-mode circuits is easily performed through wiring the nodes carrying the signals, the building blocks required for implementation of (7.62) are mod16 μA and $\left\lfloor \frac{C_{m-4}}{16\,\mu\text{A}} \right\rfloor$.

According to (7.62), the CVNS digits representing the multiplication result in the CVNS format can be shown in the following form:

$$(\!(z)\!)_3 = \left(\sum_{i=1}^{11} y_i (\!(x)\!)_{3-i} + I_1 \times \left\lfloor \frac{(\!(z)\!)_{-1}}{I_2} \right\rfloor \right) \bmod I_2$$

$$(\!(z)\!)_{-1} = \left(\sum_{i=-3}^{7} y_i (\!(x)\!)_{-1-i} + I_1 \times \left\lfloor \frac{(\!(z)\!)_{-5}}{I_2} \right\rfloor \right) \bmod I_2$$

$$(\!(z)\!)_{-5} = \left(\sum_{i=-7}^{3} y_i (\!(x)\!)_{-5-i} + I_1 \times \left\lfloor \frac{(\!(z)\!)_{-9}}{I_2} \right\rfloor \right) \bmod I_2$$

Table 7.5 Synapse multiplier input values and their corresponding output

y	$(\!(x)\!)$ (μA)	$(\!(z)\!)$ (μA)
1111 1111 1111 1111	{1, 3, 6, 12, 8, 1, 3, 6, 12, 8, 0}	{15, 15, 15, 15, 3, 10}
0010 1010 1010 1010	{0, 0, 0, 0, 1, 3, 7, 15, 14, 12, 8}	{0, 2, 7, 15, 15, 6}
0111 1111 0000 0000	{1, 2, 5, 10, 5, 10, 5, 10, 4, 8, 0}	{5, 4, 5, 6, 0, 0}
1101 0101 0101 0101	{1, 3, 7, 15, 14, 12, 8, 0, 0, 0, 0}	{13, 7, 15, 15, 11, 0}

$$(\!(z)\!)_{-9} = \left(\sum_{i=-11}^{-1} y_i (\!(x)\!)_{-9-i} + I_1 \times \left\lfloor \frac{(\!(z)\!)_{-13}}{I_2} \right\rfloor \right) \bmod I_2$$

$$(\!(z)\!)_{-13} = \left(\sum_{i=-12}^{-5} y_i (\!(x)\!)_{-13-i} + I_1 \times \left\lfloor \frac{(\!(z)\!)_{-17}}{I_2} \right\rfloor \right) \bmod I_2$$

$$(\!(z)\!)_{-17} = \left(\sum_{i=-12}^{-9} y_i (\!(x)\!)_{-17-i} \right) \bmod I_2$$

VLSI implementation of different CVNS digits representing the multiplication result has been discussed in detail in [12]. Layout of the proposed CVNS multiplier is shown in Fig. 7.5.

7.8.1 Post-Layout Simulation Results

Figure 7.6 shows the post-layout simulation results for four different input values. These inputs are applied with intervals of 0.5 μs to the CVNS multiplier. The digital weights y and the CVNS inputs $(\!(x)\!)$ corresponding to these four inputs are shown in Table 7.5. Using (7.8), the expected output for each input can be calculated. The relation between the output CVNS digits with inputs to the CVNS multiplier is summarized in Table 7.5. The post-layout simulation results shown in Fig. 7.6 are in good agreement with the mathematical derivations presented in Table 7.5.

Since the implemented multiplier is a mixed-signal design, the effect of mismatch is studied. The main building blocks of the proposed multiplier are the mod 16 μA and $\lfloor \frac{C_{m-4}}{16\,\mu\text{A}} \rfloor$ circuits. To investigate the effect of mismatch, Monte Carlo simulation of these circuits for different inputs with the temperature changing between −55 and 125 °C is conducted. The number of Monte Carlo simulation runs is set to 2500. For the mod 16 μA circuit, the maximum output variation is equal to 0.57 μA. This happens when the input to this block is equal to 15 μA.

The Monte Carlo simulation results of the mod 16 μA circuit for the input equal to 15 μA and for two extreme temperatures equal to −55° and 125° are shown in Figs. 7.7 and 7.8.

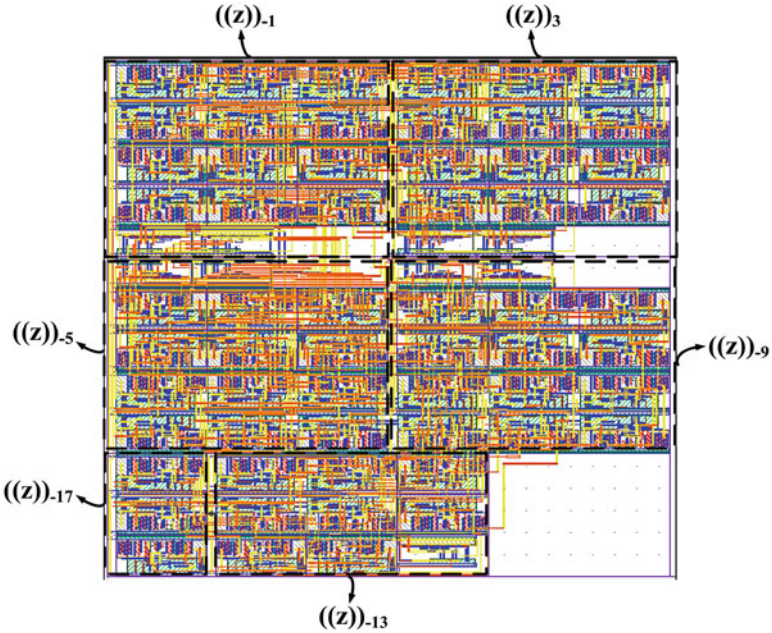

Fig. 7.5 Layout of the proposed CVNS multiplier

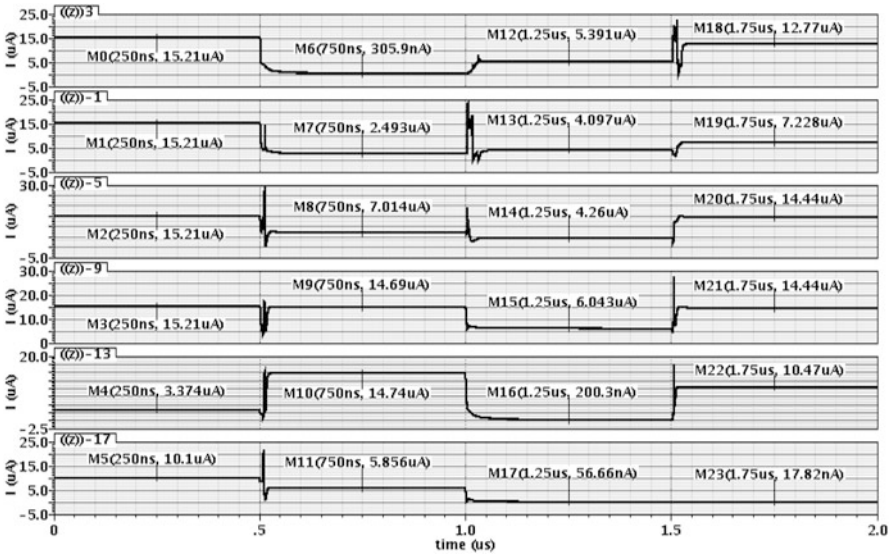

Fig. 7.6 Post-layout simulation results of the proposed CVNS multiplier

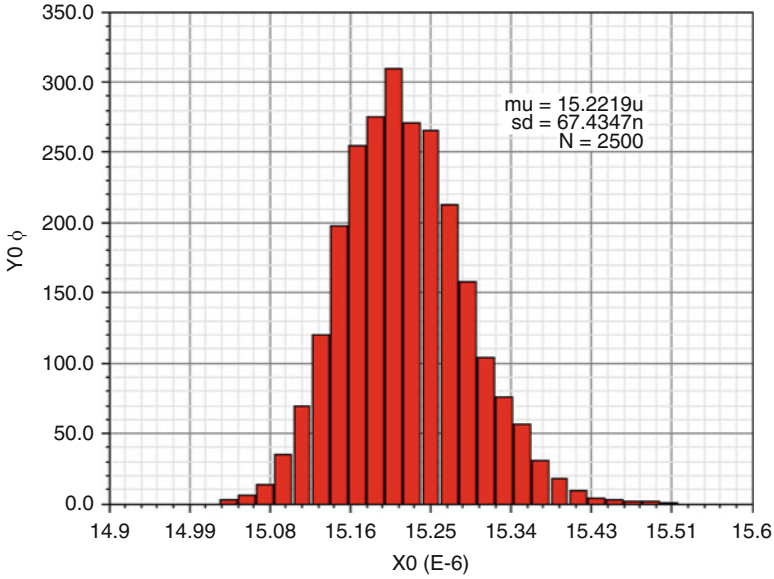

Fig. 7.7 Monte Carlo simulation result of the mod 16 μA circuit for the temperature equal to −55°

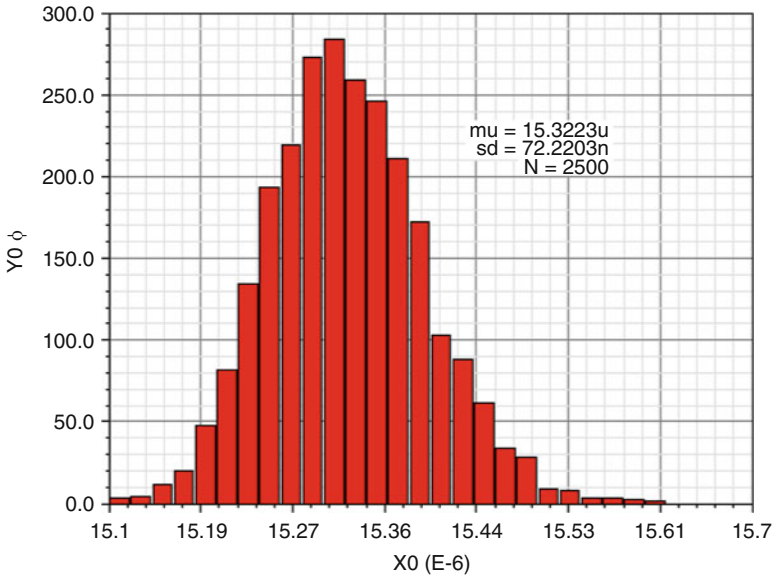

Fig. 7.8 Monte Carlo simulation result of the mod 16 μA circuit for the temperature equal to 125°

The effect of the power supply voltage variation on the mod 16 μA and $\lfloor \frac{C_{m-4}}{16\,\mu A} \rfloor$ circuits is also studied. The effect of the power supply voltage variation for different process corners and the two temperatures of −55 and 125 is simulated.

Since the maximum output variation of the mod 16 μA circuit occurs for the input equal to 15 μA, the input to this circuit is set to 15 μA.

The simulation results show that to have the output variation lower than 1 μA, the power supply voltage should be in the range of (1.63 − 2.05) V. For the $\lfloor \frac{C_{m-4}}{16\,\mu A} \rfloor$ circuit, all of the inputs are set to 1.8 V. This results in the maximum output variation.

The simulation results show that to have the output variation lower than 1 μA, the power supply voltage should be in the range of (1.55 − 2.2) V. Therefore, to have the maximum output variation of the mod 16 μA and $\lfloor \frac{C_{m-4}}{16\,\mu A} \rfloor$ circuits lower than 1 μA, the power supply voltage should be in the range of (1.63 − 2.05) V. The implemented multiplier is immune to 9.4% variation in the power supply voltage.

7.9 Conclusion

In this chapter the most fundamental aspects of the CVNS were reviewed. The CVNS addition and multiplication schemes allow discrete- and multiple-valued arithmetic to be performed without increasing the demand on analog circuitry. The CVNS arithmetic has been applied for implementation of adders, multipliers, Sigmoid function, and full implementation of neural network. To store CVNS values on the IC, CVNS registers were developed that were used for storage of synapse values. The redundancy in the CVNS can be modified based on the design restrictions. If accuracy of the implementation is not critical, the redundancy of information between the digits can be reduced. While in more critical situations, the redundancy and error correction method can be used to remove the errors.

Acknowledgements The authors would like to acknowledge the use of the following source material.

- Parts of Sect. 7.5: Copyright l' IEEE. All rights reserved. Reprinted, with permission, from B. Zamanlooy, A. Novak, M. Mirhassani, "Complexity Study of the Continuous Valued umber System Adders," IEEE International Symposium on Multiple Valued Logic, 116–121, 2012.
- Sections 7.6 and 7.7: Copyright l' IEEE. All rights reserved. Reprinted, with permission, from "CVNS Synapse Multiplier for Robust Neurochips with On-Chip Learning," *IEEE Transaction on Very Large Scale Integration (VLSI) System*, Vol. 23, No. 11, pp. 2540–2551, 2015.

References

1. I.M. Thoidis, D. Soudris, J.M. Fernandez, A. Thanailakis, The circuit design of multiple-valued logic voltage-mode adders, in *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 4 (2001), pp. 162–165
2. M.C. Mekhallalati, M.K. Ibrahim, A new high radix maximally redundant signed digit adder, in *Procedings of IEEE International Symposium on Circuits and Systems*, vol. 1 (1999), pp. 459–462
3. A. Saed, M. Ahmadi, G.A. Jullien, A number system with continuous valued digits and modulo arithmetic. IEEE Trans. Comput. **51**(11), 1294–1304 (2002)
4. A. Saed, M. Ahmadi, G.A. Jullien, W.C. Miller, Overlap resolution: arithmetic with continuous valued digits for hybrid architectures, in *IEEE Asilomar Conference on Signals, Systems and Computers*, vol. 2 (1997), pp. 1188–1191
5. A. Saed, M. Ahmadi, G.A. Jullien, W.C. Miller, Overlap resolution: arithmetic with continuous valued digits for hybrid architectures, in *Proceeding of IEEE Midwest Symposium on Circuits and Systems* (1997), pp. 377–380
6. A. Saed, M. Ahmadi, G.A. Jullien, W.C. Miller, Circuit tolerances and word lengths in Overlap resolution, in *IEEE International Symposium on Circuits and Systems* (1998), pp. 197–200
7. A. Saed, M. Ahmadi, G.A. Jullien, W.C. Miller, Arithmetic circuit analog digits, in *IEEE International Symposium on Multiple Valued Logic* (1999), pp. 186–191
8. R. Aroca, M. Ahmadi, R. Hashemian, G.A. Jullien, A B-compliment continuous valued digit adder, in *Proceeding of IEEE 9th International Conference on Electronics, Circuits and Systems*, vol. 2 (2002), pp. 433–436
9. M. Mirhassani, M. Ahmadi, G.A. Jullien, Digital multiplication using continuous valued digits, in *IEEE International Symposium on Circuits and Systems (ISCAS)* (2007), pp. 3263–3266
10. M. Mirhassani, M. Ahmadi, G.A. Jullien, Reconfigurable 64-bit binary adder based on continuous digits. IEEE Trans. Very Large Scale Integr. VLSI Syst. **16**(9), 1141–1150 (2008)
11. G. Khodabandehloo, M. Mirhassani, M. Ahmadi, CVNS-based storage and refreshing scheme for a multi-valued dynamic memory. IEEE Trans. Very Large Scale Integr. VLSI Syst. **19**(8), 1517–1521 (2011)
12. B. Zamanlooy, M. Mirhassani, CVNS synapse multiplier for Robust neurochips with on-chip learning. IEEE Trans. Very Large Scale Integr. VLSI Syst. **23**(11), 2540–2551 (2015)
13. M. Mirhassani, M. Ahmadi, G.A. Jullien, 16-bit radix-4 continuous valued digit adders, *Conference on Advanced Signal Processing, Algorithms, Architectures, and Implementations XVI, SPIE Symposium*, ID 631302 (2006)
14. B. Zamanlooy, A. Novak, M. Mirhassani, Complexity study of the continuous valued umber system adders, in *IEEE International Symposium on Multiple Valued Logic* (2012), pp. 116–121
15. J.F. Ramos, A. Gago, Two operand binary adders with threshold logic. IEEE Trans. Comput. **48**(12), 1324–1337 (1999)
16. K. Asanovir, N. Morgan, Experimental determination of precision requirements for back-propagation training of artificial neural networks, in *2nd International Conference on Microelectronics for Neural Network* (1991), pp. 9–15
17. J. Holi, J.-N. Hwang, Finite precision error analysis of neural network hardware implementations. IEEE Trans. Comput. **42**(3), 281–290 (1993)

Part II
Applications of Unconventional Number Representations

Chapter 8
RNS Applications in Digital Signal Processing

Gian Carlo Cardarilli, Alberto Nannarelli, and Marco Re

8.1 Introduction

The Residue Number System (RNS) as an alternative to binary representation in digital systems has been studied extensively in the last decades [1, 26]. The attractive feature of the RNS is that for operations such as addition and multiplication, a large dynamic range (bit-width) can be divided into sub-ranges of reduced bit-width by splitting the datapath in independent parallel channels where the operations can be executed faster (shorter carry propagation) [26].

The main drawback is that converters are required to interface an RNS datapath to the conventional systems based on binary representation, so the use of RNS is justified when the number of operations to be performed in RNS, without conversions, is significant.

Over the years, several applications in Digital Signal Processing (DSP), such as digital filters, have benefit from the RNS implementation, e.g., [8, 11, 15, 24].

Here, we describe the RNS implementation of a number of DSP algorithms by comparing their performance (clock rate, area, and power dissipation) to the same algorithms implemented in the traditional two's complement number system (TCS), and discussing some trade-offs.

In Sect. 8.2, we recall the main RNS concepts to introduce the notation used in the chapter.

In Sect. 8.3, we explain the main criteria to choose the RNS base and discuss same trade-offs in choosing the right base for a given application.

G.C. Cardarilli • M. Re
Department of Electronics Engineering, University of Rome "Tor Vergata", Rome, Italy
e-mail: g.cardarilli@uniroma2.it; marco.re@uniroma2.it

A. Nannarelli (✉)
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark
e-mail: alna@dtu.dk

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_8

In Sect. 8.4, we introduce the general architecture of RNS filters, perform a design space exploration for optimal RNS base selection, and present the RNS implementation of several filters: real/complex coefficients FIR filters, partially and fully serial filters, and adaptive filters.

In Sect. 8.5, we demonstrate how RNS can be used to implement fault-tolerant filters.

In Sect. 8.6 and in Sect. 8.7, we summarize the results of our studies on RNS based DSP by highlighting the past trends and forecasting the future ones.

In Sect. 8.8, we draw the final conclusions.

8.2 RNS Background and Notation

A Residue Number System is defined by a set of P relatively prime integers

$$\{m_1, m_2, \ldots, m_P\} .$$

The dynamic range of the system is given by the product of all the moduli m_i:

$$M = m_1 \cdot m_2 \cdot \ldots \cdot m_P . \tag{8.1}$$

Any integer $X \in [0, M-1]$ has a unique RNS representation given by:

$$X \overset{\text{RNS}}{\rightarrow} (\langle X \rangle_{m_1}, \langle X \rangle_{m_2}, \ldots, \langle X \rangle_{m_P})$$

where

$$\langle X \rangle_{m_i} = X \bmod m_i \qquad \left(\text{i.e., } \frac{X}{m_i} = q_i + \langle X \rangle_{m_i} \right)$$

and q_i is the integer quotient of $\frac{X}{m_i}$, and $\langle X \rangle_{m_i}$ is its remainder ($0 \leq \langle X \rangle_{m_i} < m_i$).

A comprehensive description of the RNS theory and its application to computer systems can be found in [1, 24, 26, 29].

Operations such as addition and multiplication are done in parallel on the different RNS moduli

$$Z = X \text{ op } Y \overset{\text{RNS}}{\rightarrow} \begin{cases} Z_{m_1} = \langle X_{m_1} \text{ op } Y_{m_1} \rangle_{m_1} \\ Z_{m_2} = \langle X_{m_2} \text{ op } Y_{m_2} \rangle_{m_2} \\ \ldots \qquad \ldots \qquad \ldots \\ Z_{m_P} = \langle X_{m_P} \text{ op } Y_{m_P} \rangle_{m_P} \end{cases} \tag{8.2}$$

and the dynamic range of each of the RNS processor channels is limited to $m_i - 1$.

8.2.1 Quadratic Residue Number System

In the case of the representation of complex numbers, we can transform the imaginary term into an integer if the equation $q^2 + 1 = 0$ has two distinct roots q_1 and q_2 in the ring of integers modulo m_i (Z_{m_i}). A complex number $x_R + jx_I = (x_R, x_I) \in Z_{m_i}$, with q root of $q^2 + 1 = 0$ in Z_{m_i}, has a unique Quadratic Residue Number System (QRNS) [24] representation given by

$$\begin{aligned} (x_R, x_I) \overset{\text{QRNS}}{\rightarrow} (X_i, \hat{X}_i) \qquad & i = 0, 1, \ldots, P \\ X_i = \langle x_R + q \cdot x_I \rangle_{m_i} & \\ \hat{X}_i = \langle x_R - q \cdot x_I \rangle_{m_i} & \end{aligned} \tag{8.3}$$

The inverse QRNS transformation is given by

$$\begin{aligned} x_R &= \langle 2^{-1}(X_i + \hat{X}_i) \rangle_{m_i} \\ x_I &= \langle 2^{-1} \cdot q^{-1}(X_i - \hat{X}_i) \rangle_{m_i} \end{aligned} \tag{8.4}$$

where 2^{-1} and q^{-1} are the multiplicative inverses of 2 and q, respectively, modulo m_i:

$$\langle 2 \cdot 2^{-1} \rangle_{m_i} = 1 \qquad \text{and} \qquad \langle q \cdot q^{-1} \rangle_{m_i} = 1 \ .$$

Moreover, it can be proved that for all the prime integers which satisfy

$$p = 4k + 1 \qquad\qquad k \in N$$

the equation $q^2 + 1 = 0$ has two distinct roots q_1 and q_2.

As a consequence, the product of two complex numbers $x_R + jx_I$ and $y_R + jy_I$ is in QRNS

$$(x_R + jx_I)(y_R + jy_I) \overset{\text{QRNS}}{\rightarrow} (\langle X_i Y_i \rangle_{m_i}, \langle \hat{X}_i \hat{Y}_i \rangle_{m_i}) \tag{8.5}$$

and it is realized by using two integers multiplications instead of four. Table 8.1 shows an example of QRNS multiplication in the ring mod 13.

8.2.2 Modular Multiplication

Because of the complexity of modular multiplication in RNS, it is convenient to implement the product of residues by the isomorphism technique [29]. By using isomorphisms, the product of the two residues is transformed into the sum of their indices which are obtained by an isomorphic transformation. According to [29], if m is prime there exists a primitive radix q such that its powers modulo m cover the set $[1, m - 1]$:

Table 8.1 Example of QRNS multiplication mod 13

Modulus: $m = 13$	
Root: $q = q_1 = 5 \leftrightarrow \langle 5 \cdot 5 \rangle_{13} = -1$	
Product: $(x_R + jx_I)(y_R + jy_I) = (3 + j)(2 + j2) = 4 + j8$	
Conversion to QRNS—Eq. (8.3)	
$X = \langle 3 + 5 \cdot 1 \rangle_{13} = 8$	$Y = \langle 2 + 5 \cdot 2 \rangle_{13} = 12$
$\hat{X} = \langle 3 - 5 \cdot 1 \rangle_{13} = 11$	$\hat{Y} = \langle 2 - 5 \cdot 2 \rangle_{13} = 5$
mod 13 multiplications	
$X \cdot Y = \langle 8 \cdot 12 \rangle_{13} = 5$	$\hat{X} \cdot \hat{Y} = \langle 11 \cdot 5 \rangle_{13} = 3$
Conversion from QRNS—Eq. (8.4)	
$Z_R = \langle 7(5 + 3) \rangle_{13} = 4$	being $\langle 2^{-1} \rangle_{13} = 7$
$Z_I = \langle 7 \cdot 8(5 - 3) \rangle_{13} = 8$	being $\langle q^{-1} \rangle_{13} = \langle 5^{-1} \rangle_{13}$

$$n = \langle q^w \rangle_m \qquad \text{with } n \in [1, m-1] \text{ and } w \in [0, m-2].$$

where w is the index of the isomorphic transformation.

The transformations $n \to w$ and $w \to n$ can be implemented with $m - 1$ entries look-up tables, if the moduli are not too large (for $m_i \leq 251$ the tables have less than 2^8 entries).

Therefore, the product $A \cdot X$ mod m can be obtained as:

$$\langle A \cdot X \rangle_m = \langle q^W \rangle_m$$

where

$$W = \langle \hat{A} + \hat{X} \rangle_{m-1} \qquad \text{with } A = \langle q^{\hat{A}} \rangle_m \text{ and } X = \langle q^{\hat{X}} \rangle_m$$

Summarizing, to implement the modular multiplication the following operations are performed:

1. Two direct isomorphic transformations (DIT) to obtain $\hat{A}$ and $\hat{X}$;
2. One modulo $m - 1$ addition $\langle \hat{A} + \hat{X} \rangle_{m-1}$;
3. One inverse isomorphic transformations (IIT) to obtain the product.

By using isomorphic transformations, the modular multipliers are replaced by modular adders which are simpler and faster.

8.2.3 Input/Output Conversion

Several methods to perform the conversions from $\mathcal{N}$ to RNS, and vice versa, have been presented over the years, e.g., [5, 19, 25, 30]. A survey of those methods is included in [22]. Clearly, the conversions constitute a significant overhead in systems implemented in RNS. However, when the number of RNS operations to

process for each input sample is large, the input and output conversions' overhead becomes less important.

8.2.3.1 Input Conversion

The conversion of the input data X in the RNS format is obtained by computing the integer remainder of the division of X by m_i

$$X_i = \langle X \rangle_{m_i} \qquad X_i = \langle X \rangle_{m_i} \tag{8.6}$$

The works presented in the literature on this subject can be categorized depending on the class of used moduli: i.e., special moduli sets and general moduli sets.

For general moduli sets with dynamic ranges 3–8 bits, input converters can be simply implemented by Look-Up Tables (LUTs). If n is the number of input bits, the LUT size in terms of number of cells is 2^n and the number of bits that are needed for the residue binary coding is

$$b = \sum_{i=1}^{P} \lceil \log_2 m_i \rceil \tag{8.7}$$

This approach can be efficiently used when the number of bits of the data to be converted in RNS is not large. A careful choice of the moduli set helps in containing the memory resources by minimizing the coding overhead (Sect. 8.3.1).

In the case in which n is large, and consequently, the amount of memory becomes too large, the techniques that can be used are based on the observation that finding a residue of a number with respect to a given modulus is basically obtained by computing modular sums of different combinations of modular powers of two [25]. Usually these techniques are fast (systolic) and viable if some latency can be tolerated.

8.2.3.2 Output Conversion

The conversion from the RNS representation of the set Z_i that represents the data encoded in RNS can be accomplished by two different techniques: the Mixed Radix Conversion (MRC) and the Chinese Remainder Theorem (CRT) [26]. The most used technique for high-speed implementations is the CRT that is based on the following reconstruction formula:

$$Z = \left\langle \sum_{i=1}^{P} \overline{m_i} \cdot \langle \overline{m_i}^{\,-1} \rangle_{m_i} \cdot Z_i \right\rangle_M = \langle H \rangle_M \tag{8.8}$$

with $\overline{m_i} = \frac{M}{m_i}$ and $\overline{m_i}^{\,-1}$ obtained by $\langle \overline{m_i} \cdot \overline{m_i}^{\,-1} \rangle_{m_i} = 1$.

8.2.4 RNS and Fault Tolerance

In the last decades, RNS fault-tolerant techniques have been studied extensively. These techniques are normally based on an RNS representation with suitable redundancy. In this way the resulting RNS structure can detect and correct errors in the output data. There are two methods for adding redundancy to the RNS representation:

1. addition of one or more redundant moduli to the normal RNS representation obtaining the so-called Redundant Residue Number Systems (RRNS) representation [2];
2. use of the Residue Number System Product Code (RNS-PC) [3].

In the following, a basic introduction on the general principles in RNS error detection and correction techniques is given. Given a K moduli RNS representation, the addition of R moduli, such that for each of the added moduli $m_i > m_K$ (being m_K the largest modulo in the RNS representation), gives an RRNS representation. The set of non-redundant moduli constitutes the application dynamic range M,

$$M = \prod_{i=1}^{K} m_i \tag{8.9}$$

The remaining R moduli form the set of the redundant moduli allowing for error detection and correction. M_K is the product of the redundant moduli

$$M_K = \prod_{i=K+1}^{P} m_i \tag{8.10}$$

From (8.9) and (8.10) derives that $M_T = M \times M_K$. The interval $[0, M_K - 1]$ is the legitimate range, while the interval $[M_K,\ M_T - 1]$ is the illegitimate range. This extended number of states to represent the legitimate range allows error detection and correction [26].

For example, the addition to the normal moduli set of one more modulo gives the possibility to determine whether a single error has occurred just by checking if it belongs to the legitimate or the illegitimate range. The same principle can be used to obtain single error correction. It can be obtained by using a set of two redundant moduli where each one must be larger than any of the moduli in the basic set. As a general consideration, the use of these algorithms requires MRC or CRT and consequently it is very important to rely on optimized hardware implementations.

8.3 Moduli Selection

The choice of the moduli in an RNS based DSP system is of particular importance because of its impact on complexity, power consumption, and speed. Moreover, it is not easy to find a simple methodology for moduli selection due to conflicting conditions. For example, in order to maximize the parallelism, the maximum number of moduli to obtain the required dynamic range should be used, but this choice is in conflict with the implementation complexity of some important operations, such as the mixed radix conversion.

In the following, we briefly illustrate a number of important criteria for RNS moduli selection.

1. **Necessary condition:** The moduli in the chosen set must be co-prime.
2. **Dynamic range:** The moduli selected must cover the dynamic range of the application.
3. **Isomorphism technique use:** To easily implement multiplications by using the isomorphism technique [29], the selected moduli must be prime. These moduli are characterized by the existence of the primitive radices that are the base of the isomorphism.
4. **Balance:** The differences between the moduli should be as small as possible. By this way the hardware complexity, and consequently, the speed performance of each of the channel of the modular processor are similar.
5. **Low coding overhead:** Because the moduli must be co-prime, we can utilize at most one power-of-two modulus. It is preferable to choose the remaining moduli among those with minimum coding overhead (Sect. 8.3.1).
6. **Simplification of the arithmetic operations:** The moduli can be selected to simplify arithmetic operations. For example, moduli that are near to powers of two (e.g., $m_i = \{2^k - 1,\ 2^k,\ 2^k + 1\}$) make modular addition and multiplication simpler.
7. **Size of individual moduli:** To better exploit the RNS characteristics in terms of parallelism, and consequently speed, the individual moduli should not be too large.
8. **Parallelism:** the use of a high cardinality set of moduli permits to exploit the parallelism of the RNS representation, but makes the output conversion more complex.

In practice, from (8.1) and (8.7) we can roughly assume that the average number of bits (bit-width) of each modulus composing the RNS base is

$$d_{\text{ave}} = \frac{\sum_{i=1}^{P} \lceil \log_2 m_i \rceil}{P} \tag{8.11}$$

To cover a given dynamic range D (such as $D \leq M$) we can select the RNS base by using two different criteria:

1. RNS base composed of a small number P of moduli. In this case, by (8.11) the dynamic range of each modulus in the base is medium/large.
2. RNS base composed of a relative large number P of moduli. In this case, the dynamic range of each modulus in the base is medium/small.

By applying criterion (1) for the implementation of a typical high performance digital filter (20–40 bits of output dynamic range), the bit-width of each base is rather large. For example, to implement a 24-bit dynamic range by using a $P = 3$ RNS base (e.g., $\{2^k - 1, 2^k, 2^k + 1\}$), we need at least $k = 9$ resulting in $d_{\text{ave}} \simeq 9$.

In contrast, by using criterion (2) we can cover the 24-bit dynamic range by $P = 6$ and the base $\{7, 11, 13, 17, 31, 32\}$ for a resulting $d_{\text{ave}} = 4.3$.

The main advantage in using several prime moduli of small dynamic range is that we can implement modular multiplication by table look-up and addition (no actual multiplication) as explained in Sect. 8.2.2.

Moreover, by criterion (2), we have more freedom in selecting the co-prime moduli of the RNS base at expenses of more complicated input and output converters. However, converters constitute an overhead independently of the chosen RNS base and, generally, the complexity of the converters does not have a significant impact for applications in which the number of RNS operations (additions and multiplications) is large.

In other words, if we obtain a significant gain in area, or power, in the RNS computation, the overhead introduced by different conversion schemes becomes marginal.

Another criticism to the use of an RNS base composed of a large number of moduli is that, due to the difference in the moduli's bit-width, the delay in the different paths is unbalanced. However, as modern synthesis tools optimize slack for low power (e.g., selection of high-threshold voltage gates in non-critical paths), unbalanced paths might result at advantage for reducing the power dissipation.

8.3.1 Coding Overhead

In [9], we introduced the *"coding overhead"* (OH) defined as the amount of extra bits required in the RNS representation of an integer compared to its TCS representation.

If the dynamic range of the TCS is $D = 2^d$, for the RNS representation the base must be chosen such that

$$\prod_{i=1}^{P} m_i = M \geq D = 2^d. \tag{8.12}$$

The total number of bits necessary for the RNS base is expressed by (8.7):

$$b = \sum_{i=1}^{P} \lceil \log_2 m_i \rceil.$$

The coding overhead is defined as

$$\mathrm{OH} = b - d. \tag{8.13}$$

A large OH has a significant impact on registers and multiplexers, i.e., on the infrastructures of the RNS datapath for data memorization and selection.

Moreover, the coding overhead heavily penalizes the RNS in datapaths requiring the storage of the operands of multiplication: *multiplicand* and *multiplier*. In an $n \times m$-bit TCS multiply unit (the *multiplicand* is an n-bit integer and the *multiplier* is an m-bit integer) the required storage for the operands is $n + m$ bits (e.g., $n + m$ flip-flops). In RNS, both *multiplicand* and *multiplier* must be expressed in the whole RNS base requiring $2 \times (n + m + \mathrm{OH})$ bits of storage.

8.4 RNS Filters

In this section, we compare the implementation of digital filters in RNS to the implementation in the conventional two's complement system (TCS). We use state-of-the-art CAD tools and contemporary standard cell libraries to evaluate the RNS benefits.

We propose a number of experiments based on the implementation of different architectures (parallel, mixed serial–parallel, totally serial) and types of Finite Impulse Response (FIR) filters such as real coefficient FIR filters and complex coefficient FIR filters (e.g., used in wavelet transforms and channel equalization).

An FIR filter is described by

$$y(n) = \sum_{k=0}^{N-1} a(k) \cdot x(n-k) \tag{8.14}$$

where N is the order of the filter, $x(n)$ and $y(n)$ are, respectively, the input and the output sequences, and $a(k)$ are the filter coefficients (filter mask). By (8.2), the filter (8.14) is specified in RNS as

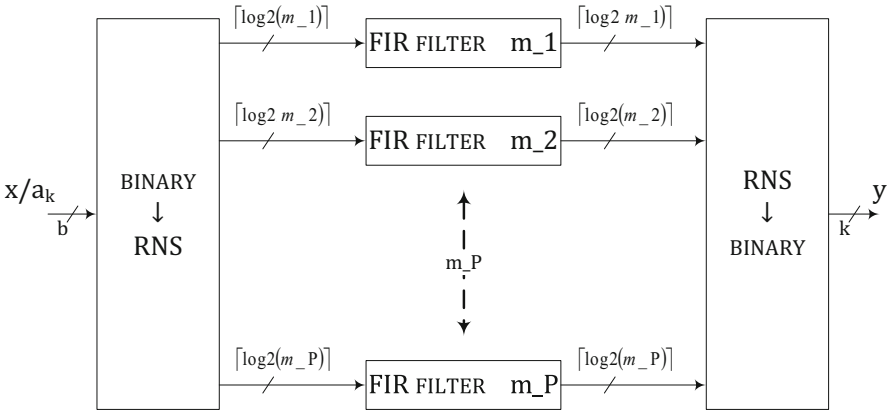

Fig. 8.1 RNS implementation of an FIR filter

$$y(n) = \sum_{k=0}^{N-1} a(k) \cdot x(n-k) \overset{\text{RNS}}{\rightarrow} \begin{cases} y_{m_1}(n) = \left\langle \sum_{k=0}^{N-1} \langle a_{m_1}(k) \cdot x_{m_1}(n-k) \rangle_{m_1} \right\rangle_{m_1} \\ y_{m_2}(n) = \left\langle \sum_{k=0}^{N-1} \langle a_{m_2}(k) \cdot x_{m_2}(n-k) \rangle_{m_2} \right\rangle_{m_2} \\ \qquad \cdots \\ y_{m_P}(n) = \left\langle \sum_{k=0}^{N-1} \langle a_{m_P}(k) \cdot x_{m_P}(n-k) \rangle_{m_P} \right\rangle_{m_P} \end{cases} \tag{8.15}$$

and its hardware architecture is shown in Fig. 8.1.

Our analysis is based on a Design Space Exploration (DSE) to investigate the performance of the TCS and RNS multiply-add (MADD) units in terms of delay, area, and power dissipation for a set of different dynamic ranges.

By the DSE, we identify design corners, for each operator, where the RNS is advantageous over the TCS. Then, we verify and validate the results of the DSE for selected corners by implementing actual digital filters in both RNS and TCS. Namely,

- real coefficients FIR filters,
- a parallel/serial FIR filter,
- a polyphase (complex) filter,
- an adaptive filter.

In the experiments, we synthesized the filters by using Synopsys's Design Compiler in a commercial CMOS 45 nm standard cell library. We used Synopsys's Integrated Circuit Compiler (ICC) to layout the circuits, and Synopsys's VCS to simulate the layout extracted netlists for generating the switching activity files necessary for the power estimation.

For the RNS implementations, the overhead of the input/output converters is always taken into account.

8.4.1 Design Space Exploration

The DSE is based on the characterization in the 45 nm library of standard cells of a fused (merged) multiply-add (MADD) unit in both TCS and RNS. The characterization is done on a set of dynamic ranges which are typical of the implementation of digital filers. Namely, 16, 20, 24, 28, 32, 36, 40, 44, and 48 bits.

In the DSE, we consider two design corners:

Ms the maximum speed corner, obtained by synthesizing the functional unit to achieve the maximum speed.

ma the minimum area corner, obtained by synthesizing the unit to achieve the minimum area.

The two corners for the TCS MADD are illustrated in Fig. 8.2. The 3D plot (area, dynamic range, timing constraint) is projected on the *area–dynamic range* plane in Fig. 8.2 (bottom). The 2D graph of Fig. 8.2 can be read as follows. For example, a 36-bit MADD can be synthesized in a circuit with resulting area between 2200 μm^2 (*ma*) and 5600 μm^2 (*Ms*) depending on the synthesis timing constraint.

Similarly, we can have a projection of a 3D delay plot on the *delay–dynamic range* plane.

In the DSE, for each dynamic range, we perform a delay, area, and power dissipation characterization for 12 different synthesis timing constraints T_C, ranging from the *ma* to the *Ms* corners.

As discussed in Sect. 8.3, to cover a given dynamic range in RNS several sets of co-prime moduli (RNS base) can be chosen.

Among the several selection criteria, we choose one based on the minimization of a specific cost function. To cover the given dynamic range D, we consider all the possible combinations of co-prime moduli[1] which cover the range and do not exceed D too much (e.g., 2 bits): $2^d = D \leq \prod^P m_i = M \leq 2^{d+2}$. Then, we choose the set (RNS base) which minimizes the cost function (e.g., area or power dissipation of the RNS operator) at a given timing constraint T_C.

8.4.1.1 TCS and RNS MADDs

In TCS, the MADD $Z = X \times Y + W$ is efficiently implemented by adding W (properly aligned) in the partial products reduction tree of the multiplier. We opted for a radix-4 multiplication to reduce the number of partial products.

[1]Prime numbers from 3 to 71 and at most one of power-of-two (2^k) from 4 to 256.

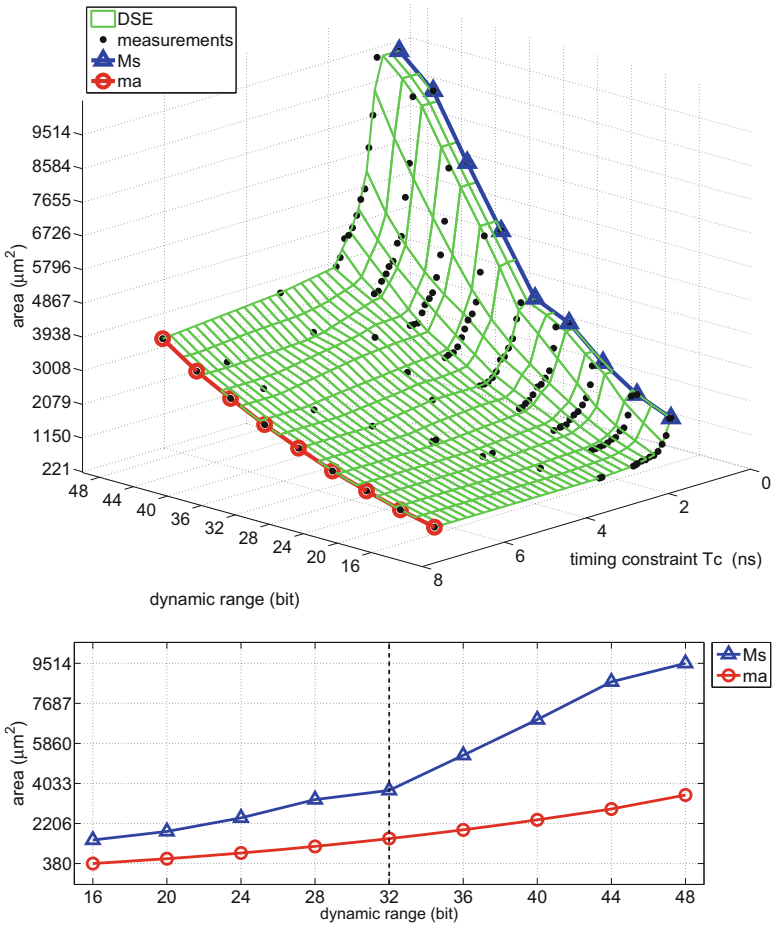

Fig. 8.2 TCS MADD plots. 3D Area (*top*). The minimum area (ma) side for area (*bottom*)

The RNS MADD architecture is implemented by cascading a isomorphic multiplier and a modular adder, for each modulus of the RNS base.

Because we use prime numbers, we can transform a modular multiplication into a modular addition of the isomorphic indices, as explained in Sect. 8.2.2.

If the modulus is small (its bit-width is 6–8 bits), the forward and reverse isomorphic transformations can be implemented by look-up tables of reasonable size which are then synthesized into multi-level combinational logic.

For the implementation of the isomorphism, we considered the two alternatives of [17, 21], and selected the one with the lower cost for the specific RNS base.

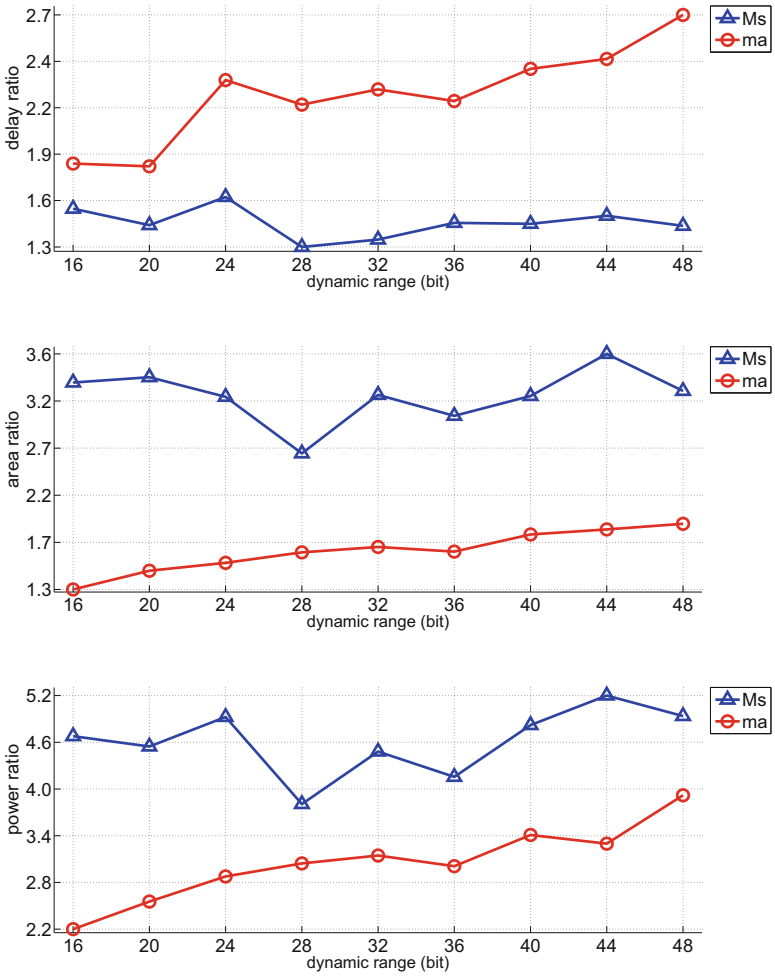

Fig. 8.3 DSE multiply-add: ratios TCS/RNS in *Ms* and *ma* corners

8.4.1.2 Characterization Results

Figure 8.3 shows the plots of the ratio between the TCS and RNS values for DSE of the MADD units. In both the *Ms* and the *ma* corners, the RNS MADD is always faster, smaller, and more power efficient than the TCS MADD (ratio > 1.0).

The RNS MADD implementation is 30–60% faster than the corresponding TCS implementation in the *Ms* corner and about two times faster in the *ma* corner. Moreover, the RNS MADDs consume roughly one quarter of the power in the *Ms* corner and about one third of the power in the *ma* corner.

The characterization of the MADD is carried out by applying random test-vectors uniformly distributed in the whole dynamic range of the input signals.

However, when a MADD unit is used in FIR filters, one of the inputs is connected to one of the coefficients specifying the filter mask. As an example, in linear-phase frequency filters with N coefficients, the coefficients closer to the mid-point $N/2$ have a full dynamic range, while the ones closer to the end-points (i.e., 0 and $N-1$), also called *filter's tails*, have a reduced dynamic range.

To simulate the multiplication by a reduced dynamic range operand, we generate random test-vectors at a progressively reduced dynamic range. The dynamic range reduction characterizing this experiment only affects input Y of the MADD unit ($Z = X \times Y + W$) while at inputs X and W we apply random test-vectors at full dynamic range.

The results of the experiment are shown in Fig. 8.4. On the x-axis we indicate how many Most-Significant Bits (MSBs) are held constant (to 0 or 1) to reduce the dynamic range of the test-vectors. In the mid-chart point (labeled "0") in Fig. 8.4 the MADD works at full dynamic range. The dynamic range is reduced moving from the center of the chart to the right for positive values (MSBs held at 0) and to the left for negative values (MSBs held at 1).

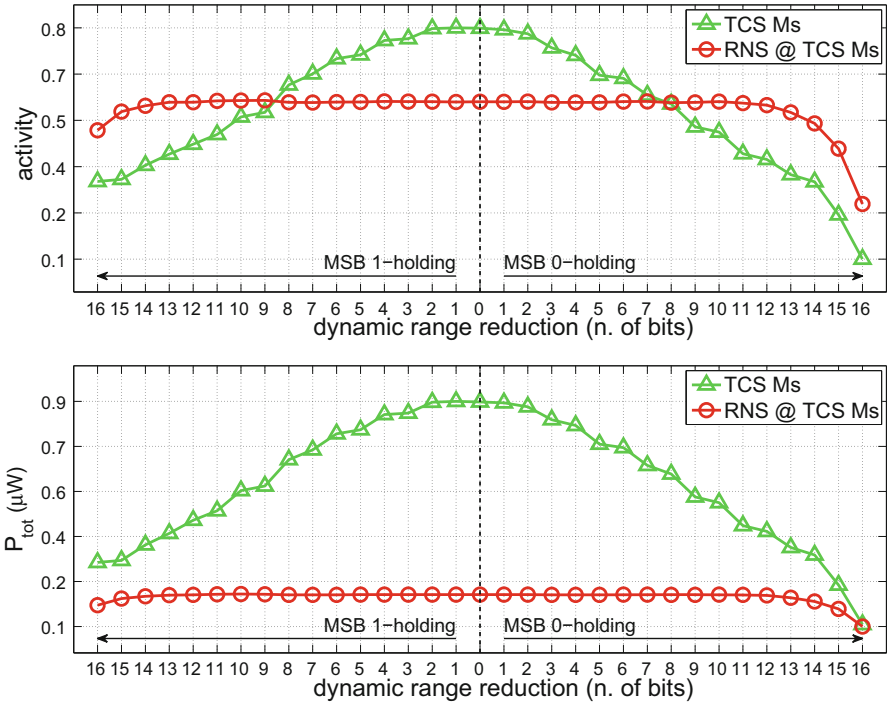

Fig. 8.4 Switching activity (*top*) and total power consumption (*bottom*) in 32-bit MADD when the dynamic range of test-vectors is progressively reduced

Figure 8.4 shows that the TCS MADD is sensitive to the variations in the dynamic range of one of the operands, while the RNS MADD is almost insensitive to those variations.

In summary, although the data of Fig. 8.3 points to a better power efficiency of the RNS over the TCS MADD, the benefits are reduced in actual filters since in the filter's tails the TCS MADDs have a reduced power dissipation due to the smaller dynamic range of one of the operands.

8.4.2 Real FIR Filters

The real coefficients FIR filter in two's complement is implemented by using the transposed form architecture, as shown in Fig. 8.5 for a section of the filter. The filter is composed of a regular repetition of blocks (multiply-add and register in the figure), the so-called *filter's tap*.

In RNS, the structure of the each modular filter in Fig. 8.1 is obtained from Fig. 8.5 by using one modular multiply-add unit for each filter tap.

Moreover, since the coefficients a_k are loaded in the registers at filter initialization (serially in Fig. 8.5) and do not change value until a new mask is selected, the registers holding the coefficients are clock-gated to save power (both in TCS and RNS).

We implement and analyze the following real FIR filters:

- **EXP-1**: Low dynamic range (16 bits) and low-order filter (16 coefficients)
- **EXP-2**: High dynamic range (48 bits) and high-order filter (64 coefficients).

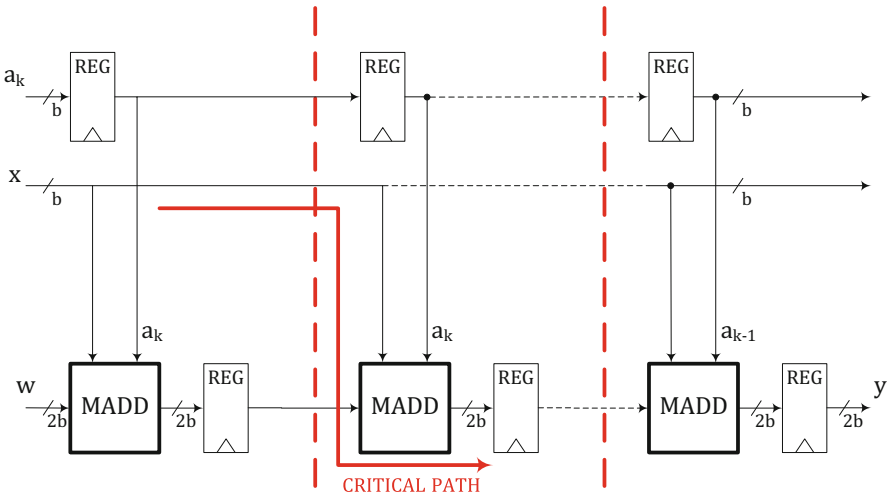

Fig. 8.5 TCS implementation of the transposed form real coefficients FIR filter (section)

In this comparison, we evaluate the performance of the filters in terms of the traditional figures of merit such as delay, area, and power dissipation, and we provide other parameters such as: power density, average capacitance (C_{ave}), average switching activity (a_{ave}), percentage of glitches in the activity of the internal nodes, and average wirelength (w_{ave}).

As done for the DSE, we show the performance of the filter implementations for two distinct design corners: maximum speed corner (*max speed, Ms*) and the minimum area corner (*min area, ma*). For the *Ms* corner, both TCS and RNS filters are implemented to operate at a maximum speed achievable by the TCS filter (always slower than RNS maximum speed), while for the *ma* corner, the clock period of the units is not constrained. In all experiments, the average power consumption is estimated at clock frequency 100 MHz.

The results relative to EXP-1 are reported in Table 8.2. In the table, the rightmost columns show the ratios TCS/RNS for the corresponding metric. At *Ms* corner, the RNS filter implementation, synthesized at the maximum operating speed of the TCS, is almost half the area and consumes 41% less power than the TCS filter. However, this is no longer valid at *ma* corner, where the TCS implementation is smaller and consumes less power.

Figure 8.6 shows the power dissipation in each tap of the filter and confirms that, as explained in Sect. 8.4.1.2, the power advantages of RNS are reduced, or nullified, in the tails of the filter. These results apply to frequency filters, if we implement correlation filters or linear transformations the power dissipation in TCS taps may be uniform along the filter.

Moreover in the *ma* corner, the power savings in the RNS taps (Fig. 8.6) are offset by the overhead of input/output converters.

The results of EXP-2 (64-tap FIR filter) are reported in Table 8.3. In this case, the results for the *Ms* corner are even more advantageous for the RNS implementation, but also in the *ma* corner the RNS filter is advantageous over the TCS. In this latter case, the power savings in the RNS taps are not offset by the input/output converters.

The results of these experiments on real coefficients FIR filters confirm that for high-order and high-speed parallel filters an RNS implementation is preferable over an implementation in TCS, while for low-order filters with relaxed time requirements a TCS implementation is more appropriate.

8.4.3 Parallel/Serial FIR Filter

If the resources are limited, a high-order FIR filter can be implemented partly in parallel and partly serially.

In this second case study, we consider a 128-tap FIR filter working at a frequency of 20 MHz. The filter's dynamic range is 36 bits with 18-bit input samples $x(n)$ and 18-bit coefficients a_k. The selected RNS base is $\{3, 5, 7, 11, 13, 17, 19, 23, 31, 32\}$ ($P = 10$).

The filter expression (8.14) is *"folded"* in a datapath of reduced parallelism

Table 8.2 Post-layout results of EXP-1 (16-tap filter) in TCS max speed (top) and min area (bottom) design corners

EXP-1							
Max speed	TCS			RNS			$\frac{\text{TCS}}{\text{RNS}}$
Td [ns]	1.26			1.26 (*Ms* 0.9)			1.00
Area [mm^2]	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
	0.023	0.003	0.026	0.010	0.004	0.014	1.86
Power [mW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	2.477	0.004	2.481	1.460	0.001	1.461	1.70
$\frac{\text{Power [mW]}}{\text{Area [mm}^2\text{]}}$	97.11			102.14			0.95
C_{ave} [fF]	6.21			4.58			1.35
a_{ave}	0.44			0.47			0.94
Glitches (%)	30			25			1.24
w_{ave} [μm]	9.99			11.09			0.90
Min area							
Td [ns]	3.67			2.73			1.34
Area [mm^2]	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
	0.008	0.003	0.011	0.007	0.004	0.012	0.91
Power [mW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	0.986	0.001	0.987	1.137	0.001	1.138	0.87
$\frac{\text{Power } [mW]}{\text{Area [mm}^2\text{]}}$	92.97			97.08			0.96
C_{ave} [fF]	4.40			4.07			1.08
a_{ave}	0.47			0.52			0.90
Glitches (%)	27			26			1.04
w_{ave} [μm]	11.04			11.18			0.99

$$y(n) = \sum_{i=0}^{15} \left(\sum_{j=0}^{7} a_{8i+j} x(n - 8i - j) \right) \tag{8.16}$$

and executed serially on a 16-tap filter (direct form) as depicted in Fig. 8.7. The core of each sub-filter is a multiply-add unit and an accumulator register (MACC in Fig. 8.7). Each serial sub-filter works at $8 \times 20 = 160$ MHz. The details of the filter's architecture are explained in [18].

The synthesis of the TCS and RNS parallel/serial filters, referred to as EXP-3, is done at 160 MHz clock frequency. Also in this case, clock-gating is applied for both coefficients and the $x(n)$ register files (see Fig. 8.7).

The results of EXP-3, summarized in Table 8.4, show that the RNS unit is much larger than the TCS one. The extra area is mostly due to the overhead (OH) introduced by RNS in the register files for the filter coefficients and the input samples: $\text{OH}(x) = 41 - 18 = 23$, $\text{OH}(a_k) = 41 - 18 = 23$.

However, with respect to the power dissipation, this huge overhead is mitigated by the use of clock gating. Moreover, since multiply-accumulate (MACC) inputs

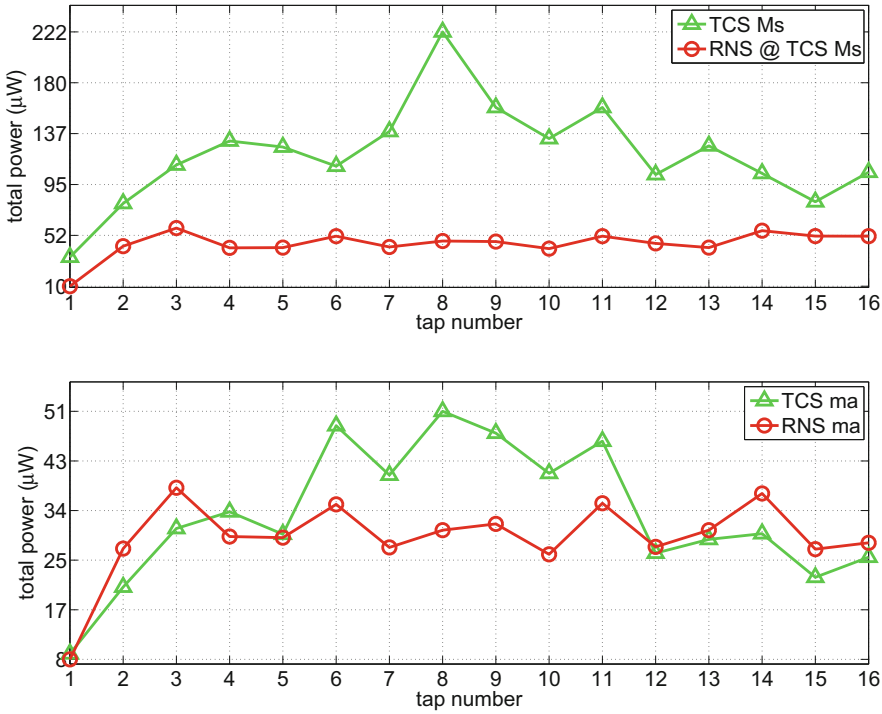

Fig. 8.6 Power dissipation per tap in EXP-1. Max speed (*top*) and min area (*bottom*) design corners

change every clock cycle, the benefits of reduced dynamic range in the filter tails are reduced for the TCS implementation.

Clearly, a marginal reduction in the power dissipated by the RNS filter (−13%) might not justify a doubled area, especially if the silicon area is limited. However, if the power resources are limited, the RNS unit might be preferable.

A further reduction in power dissipation is achieved if we synthesize the RNS filter at a tighter timing constraint, such as at maximum speed. The power due to the extra capacitance in the combinational part of the filter can be compensated by reducing the supply voltage until the cycle time of the faster unit is as slow as the nominal clock cycle (6.25 ns). This method to save power is explained in detail in [18].

8.4.4 Polyphase (Complex) Filter

The polyphase decomposition allows the implementation of uniform filter banks with reduced computational complexity and sampling frequency [28].

Table 8.3 Post-layout results of EXP-2 (64-tap filter) in TCS max speed (top) and min area (bottom) design corners

EXP-2							
Max speed	TCS			RNS			$\frac{\text{TCS}}{\text{RNS}}$
Td [ns]	2.10			2.10 (*Ms* 1.8)			1.00
Area [mm^2]	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
	0.504	0.028	0.532	0.130	0.044	0.174	3.06
Power [mW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	69.61	0.083	69.69	19.74	0.015	19.76	3.53
$\frac{\text{Power [mW]}}{\text{Area [mm}^2\text{]}}$	130.9			113.4			1.15
C_{ave} [*fF*]	6.73			4.66			1.44
a_{ave}	0.63			0.55			1.13
Glitches (%)	41			33			1.26
w_{ave} [μm]	14.18			12.69			1.12
Min area							
Td [ns]	9.07			7.92			1.15
Area [mm^2]	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
	0.244	0.028	0.272	0.112	0.044	0.156	1.74
Power [mW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	40.35	0.018	40.36	18.21	0.010	18.22	2.22
$\frac{\text{Power [}mW\text{]}}{\text{Area [mm}^2\text{]}}$	48.5			116.8			1.27
C_{ave} [fF]	5.53			4.28			1.29
a_{ave}	0.83			0.65			1.27
Glitches (%)	53			38			1.39
w_{ave} [μm]	16.28			12.25			1.33

For the specific filter of order 1024 chosen as a case study (EXP-4), the architecture is composed of a decimator followed by $C = 128$ FIR sub-filters (channels) of $1024/128 = 8$ taps working at frequency f_0/C followed by a Inverse Discrete Fourier Transform (IDFT) unit. The filter bank is sketched in Fig. 8.8 and the complete specifications of the design can be found in [10]. The filter data rate is $f_0 = 200\,\text{MHz}$.

A complex representation (real and imaginary) is required. Therefore, the filter is implemented in Complex TCS, or CTCS, and in a restricted RNS base suitable for the Quadratic RNS, or QRNS (Sect. 8.2.1).

Due to the low (decimated) data rate, the 128 8-tap filters are implemented serially (similarly to Fig. 8.7 bottom) with a resulting operating frequency of $8 \times \frac{200}{128} = 12.5\,\text{MHz}$ in the serial filters.

The serial filters have a dynamic range of 23 bits for the real and imaginary parts (x_R and x_I are 10 bits, and a_{kR} and a_{kI} are 12 bits). The QRNS base is $\{13, 17, 29, 37, 41\}$.

The implementation results are reported in Table 8.5.

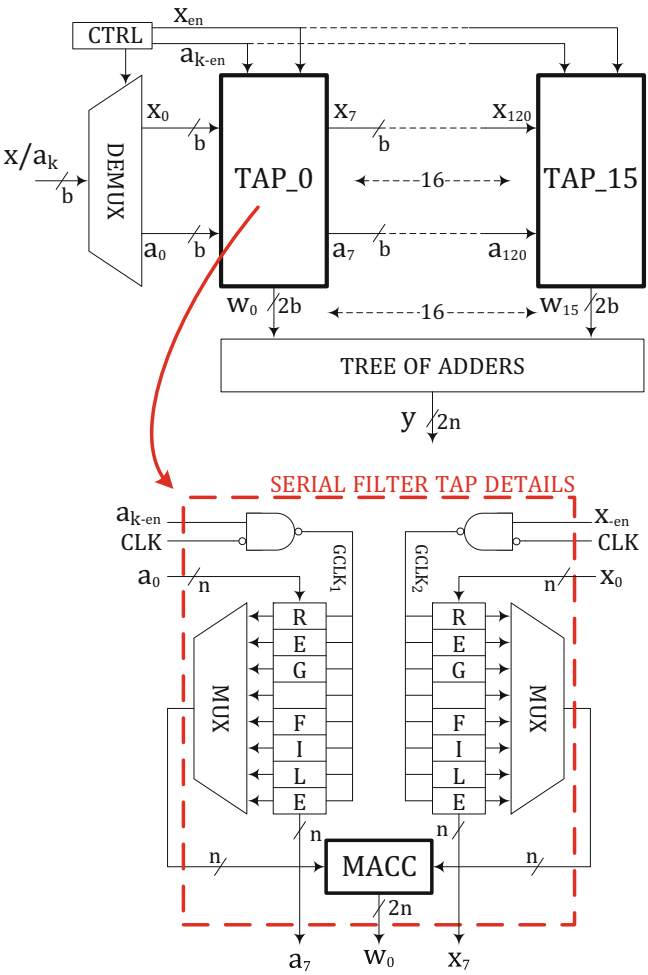

Fig. 8.7 Serial implementation of 128-tap filter by one 16-tap parallel filter. Top-level (*top*), serial implementation of the inner convolution of (8.16) in one TAP_i (*bottom*)

The RNS coding overhead on the coefficients $a_{kR/kI}$ is $\mathrm{OH} = 26-12 = 14$ and it has a dramatic impact on the corresponding register files and it reflects on the larger area of the QRNS serial filter.

However from a power dissipation perspective, the impact of the larger area is neutralized by the use of clock gating that reduces the switching activity in the register files.

Moreover, for serial filters, as in EXP-3, the switching activity of the multipliers is less sensitive of the filter mask in CTCS.

Table 8.4 Post-layout results of EXP-3 (128-tap serial-8 filter) at $T_C = 6.25$ ns (min area corner)

EXP-3							
	TCS			RNS			$\frac{\text{TCS}}{\text{RNS}}$
Td [ns]	6.25			5.59			1.12
Area [mm^2]	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
	0.046	0.033	0.079	0.045	0.072	0.117	0.67
Power [mW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	9.524	0.005	9.529	8.46	0.007	8.467	1.13
$\frac{\text{Power [mW]}}{\text{Area [mm}^2\text{]}}$	120.95			72.36			1.67
C_{ave} [fF]	5.59			5.54			0.73
a_{ave}	0.82			0.55			1.49
Glitches (%)	55			49			1.11
w_{ave} [μm]	16.85			19.62			0.86

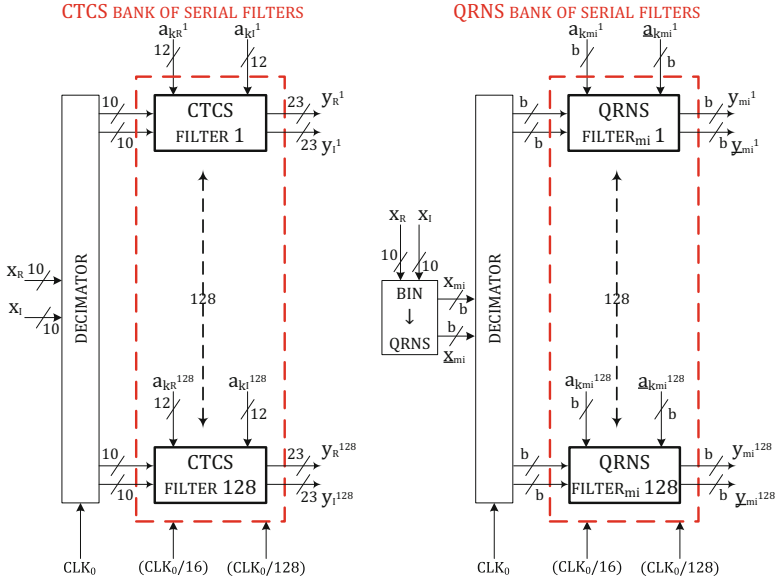

Fig. 8.8 Structure of polyphase filter bank. IDCT is omitted

Finally, in QRNS only two modular multipliers are necessary, while for CTCS four integer multipliers are needed for each complex multiply-add.

In summary, when the operational frequency is not critical for the given technology, the use of (Q)RNS results in larger units, but, due to the reduced switching activity, the (Q)RNS is still more power efficient than the (C)TCS.

Table 8.5 Post-layout results of EXP-4 for single serial complex filter (clocked at $f = 12.5$ MHz)

EXP-4							
	CTCS			QRNS			$\frac{\text{CTCS}}{\text{QRNS}}$
Td [ns]	5.00			4.45			1.12
Area	Comb.	Seq.	Tot.	Comb.	Seq.	Tot.	
[1000 · mm^2]	4.6	3.0	7.6	3.0	6.4	9.4	0.81
Power [μW]	Dyn.	Leak.	Tot.	Dyn.	Leak.	Tot.	
	93	0.5	93.5	70	0.6	70.6	1.33
$\frac{\text{Power [mW]}}{\text{Area [mm}^2\text{]}}$	12.3			7.4			1.66
C_{ave} [fF]	4.03			4.84			0.83
a_{ave}	0.52			0.30			1.75
Glitches (%)	39			22			1.77
w_{ave} [μm]	11.05			15.19			0.73

8.4.5 Adaptive Filters

Adaptive filters are often used for channel equalization in digital communication systems for time-varying channels and for echo cancellation in full-duplex systems [16, 27].

Different updating algorithms (LMS, RLS, FBLS) can be used depending on the performance of the adaptive algorithm such as steady state error and convergence rate.

Some classes of applications require a relative small number of coefficients for the FIR filter and a fast adaptation. In this case, a parallel implementation of the updating algorithm is feasible due to the small number of taps. Other applications require a large number of taps, but block or serial updating of the filter coefficients is used since the channel variations in time are slow with respect to the data rate.

In the following, we present the RNS implementation of an adaptive LMS FIR filter. Detail can be found in [4].

The block diagram of the system is shown in Fig. 8.9. The input signal $d(n)$ is distorted by the channel, the radio frequency, and the acquisition part of the system. This distortion is modeled with an FIR filter characterized by a strong disequalization in amplitude and phase.

We chose the LMS algorithm for coefficient updating, since it is not too expensive in terms of hardware requirements with respect to other algorithms, while sufficiently fast in tracking.

A training sequence, sent to the receiver in a time-slot mode, is used to update the filter coefficients. Therefore, a sufficient amount of time is available for channel recovery and the new estimated coefficients are computed by using a serial architecture (e.g., a multiply-and-accumulate, or MAC, filter).

Figure 8.10 displays the block diagram of the adaptive filter. When the system is processing the user data, the serial LMS block is freezed and its coefficients are the

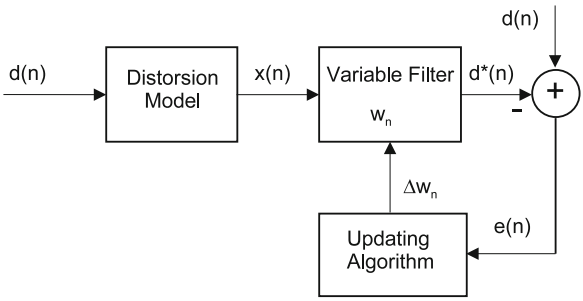

Fig. 8.9 Model of the system

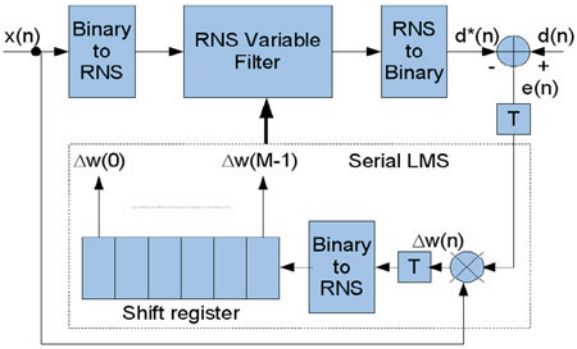

Fig. 8.10 Adaptive filter architecture

same ones of the Variable Filter. Once the training sequence is detected, its samples are delivered to the MAC filter used to evaluate the signal $d^*(n)$. After the MAC filter latency, the error signal $e(n)$ and the new coefficients are computed serially. When all the coefficients have been updated, an iteration of the LMS algorithm is completed and a second iteration is started by feeding the MAC filter with the training sequence stored in the loop filter, and so on in an iterative manner. Finally, when a new training sequence is detected, the MAC filter coefficients are copied in the Variable Filter and the cycle starts again as described above.

By simulations we determined the number of the taps of the equalizer to be $N = 192$, and the dynamic range of the FIR filter is 32 bits.

8.4.5.1 Hybrid RNS Architecture and Implementation

The straightforward implementation of the equalizer in RNS would require a dynamic range extension to 44 bits, because the scaling operation is not trivial in RNS. To limit the dynamic range of the RNS implementation, and the size of the filter, the multiplication needed for the LMS algorithm implementation is carried out in binary arithmetic. In this way, the adaptive filter is partly implemented in RNS (the FIR filter) and partly in two's complement (the LMS algorithm). In this hybrid architecture, we take advantage of the better characteristics of RNS filters

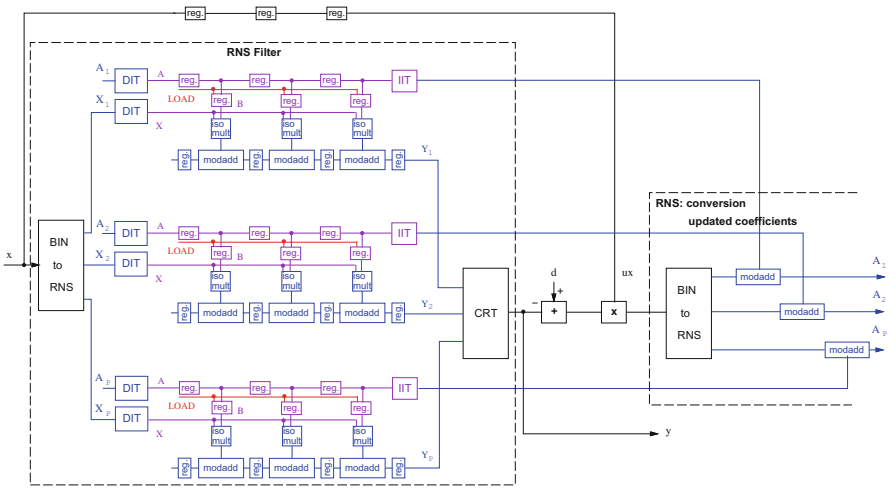

Fig. 8.11 Hybrid adaptive filter architecture

Table 8.6 Results of adaptive filters implementations

	Clock freq.	Area		Power	
	[MHz]	[mm^2]	Ratio	[mW]	Ratio
TCS	200	3.1	1.00	215	1.00
RNS	200	1.7	0.55	125	0.58

concerning reduced area and power dissipation, and we perform the scaling to obtain the new filters coefficients in two's complement by a simple truncation.

The resulting system block diagram is shown in Fig. 8.11 (3 RNS channels, $P = 3$). In the figure, the parts implemented in RNS are displayed inside dotted boxes. In addition to the hardware necessary for a programmable RNS FIR filter, the variable filter of Fig. 8.11 requires an extra binary-RNS converter (right-hand side of the figure) to convert the updated coefficients in RNS and an extra bank of registers to store the new coefficients while the update is in progress.

The characteristics of the RNS filters depend on the choice of moduli to cover the given dynamic range. To select the RNS base that gives the best trade-offs delay/area/power, we used the tool of [12]. The selection of moduli done by the tool is based on the characterization of all the RNS components of the filter (modular adders and multipliers, converters, etc.) performed under several timing conditions.

For the 192-tap and 32-bit dynamic range filter of this system, the tool selected the following set of moduli (RNS base): {5, 7, 11, 13, 17, 19, 23, 128}.

The hybrid RNS adaptive filter is compared with a similar filter implemented in two's complement (TCS). Table 8.6 reports the characteristics of the variable filter implemented in RNS and in TCS.

The data in Table 8.6 show that the implementation of the variable filter in RNS offers savings in area and power dissipation of about 50%.

8.5 RNS Fault-Tolerant Filters

Fault detection and fault masking are of fundamental interest for the implementation of complex DSP systems in modern sub-nanometric technologies. The scaling of microelectronic technologies in fact reduces the charge accumulated in the nodes of the circuit and consequently, also at sea level high-energy radiations are able to create reliability problems (single event upset, or SEU, faults). Fault detection and fault masking in the RNS representation have been studied, in several papers, such as [2, 13, 14, 23]. These works are based on the Redundant Residue Number System (RRNS) already presented in Sect. 8.2.4.

The RRNS representation does not guarantee that a fault inside the output conversion block or inside the error detection and correction block will be detected and/or corrected, so it is important to mask the effect of a fault inside these blocks to increase the reliability of the entire system. This can be obtained by using a dedicated voter. This voter performs both the correction of an error inside a modulus of the RRNS DSP and of an error inside the output converter. The use of this technique allows the implementation of a totally fault-tolerant RNS based FIR filter saving resources with respect to a Triple Modular Redundancy (TMR) implementation of the blocks performing RNS based error correction.

An RRNS is defined as a residue number system with R additional moduli. The first K moduli form a set of non-redundant moduli, and their product represents the legitimate range, M that is, $M = \prod_{i=1}^{K} m_i$. The remaining $P - K = R$ moduli form the set of redundant moduli that allows error detection and correction. The corresponding integer X of a residue vector $(x_{m_1}, \ldots x_{m_P})$ belongs to the interval $[0; M_T - 1]$, where $M_T = \prod_{i=1}^{P} m_i$. This interval, usually called total range, can be divided into two adjacent intervals: the interval $[0; M - 1]$ is called the legitimate range and the interval $[M; M_T - 1]$ is the illegitimate range. The m_i-projection of X, denoted X_i, is defined as the residue vector $(x_{m_1}, \ldots, x_{m_{i-1}}, x_{m_{i+1}}, \ldots, x_{m_P})$, i.e., the RNS representation of X with the ith residue digit deleted. In [2] the authors prove that in a redundant RNS with $R = 2$ for any error affecting an element i of the residue vector the projections X_j, for $j \neq i, i = 1, \ldots, P$ are all illegitimate, while the X_i projection is in the legitimate range. From this consideration, it is straightforward to detect and correct an error in the RRNS representation. The erroneous module is characterized by its i-projection belonging to the legitimate range, while the correct value of the integer can be obtained performing the output conversion of the X_i projection. The architecture of the RNS FIR filter with the error correction capability is shown in Fig. 8.12.

The blocks called Chinese Remainder Theorem (CRT) perform the output conversion for the m_i projections, while the block called "CHOOSE LEGITIMATE" selects which input is in the legitimate range.

The architecture in Fig. 8.12 allows the correction of a fault inside the m_i filters and the input converters. The correction of the errors caused by a fault in a CRT block is not guaranteed by the architecture in Fig. 8.12.

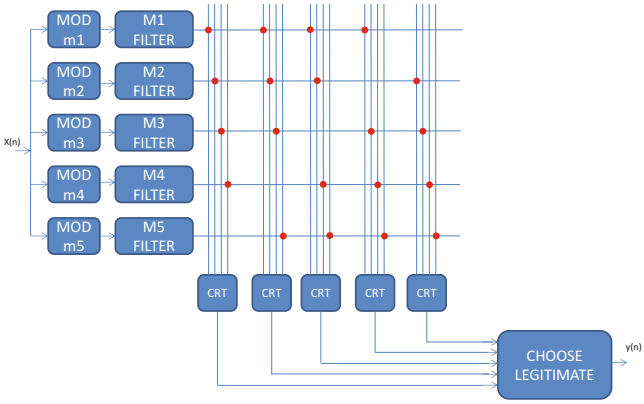

Fig. 8.12 RNS implementation of an FIR filter with error correction ($P = 5$)

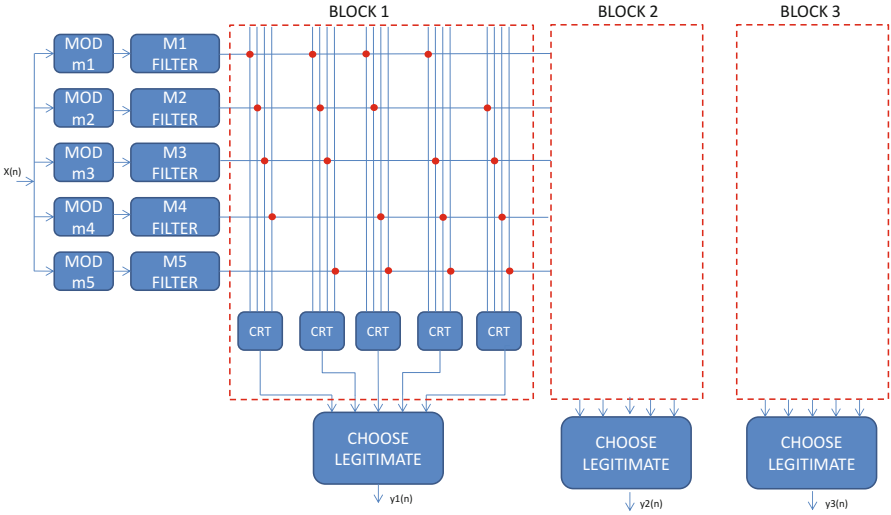

Fig. 8.13 RNS implementation of an FIR filter with error correction and TMR output converter ($P = 5$)

The trivial way to supersede this problem is represented in Fig. 8.13, in which the TMR approach is used in the blocks implementing the output conversion and the error correction. Here, the final output is triplicated and the fault is discarded by using a minority voter. The minority voter allows detecting and disabling the incorrect output.

8.5.1 Fault Masking in the Output Converters

The complete fault coverage for the RNS based FIR filter requires the detection and correction of the faults that can occur in the following blocks:

1. the input converter,
2. the mod m_i filters,
3. the output converter.

The input converter is realized implementing P independent m_i modulo reduction blocks. A fault in the input converter produces an incorrect input only to the corresponding m_i filter and therefore has the same behavior of a fault inside one of the mod m_i filters. The RRNS guarantees the detection and correction of a fault in these blocks.

Instead, if a fault occurs in one of the output converters (Fig. 8.12) the fault masking property is not guaranteed. All the input values for the block CHOOSE LEGITIMATE belong to the legitimate range and the block cannot detect an error in one of its inputs. However, if the fault occurs in one of the output converters, the other output converters produce as output the correct value. Substituting the block CHOOSE LEGITIMATE with a block performing a majority voting of the inputs a fault inside one of the CRT blocks can be masked. The block that substitutes the CHOOSE LEGITIMATE performs two operations:

1. Selection of a legitimate value between all the input values if a fault affects only a residue digit.
2. Selection of the majority of its inputs if the fault affects one of the output converters.

To this purpose, a block called "LEGITIMATE VOTER" that selects the majority of the inputs belonging to the legitimate range to perform together these two operations is introduced. The LEGITIMATE VOTER is therefore able to provide the correct output for all the faults occurring inside the RRNS based FIR filter. The only single point of failure of this architecture is represented by the LEGITIMATE VOTER itself. To obtain a total fault tolerance architecture for the filter, we can triplicate this block. This choice is more convenient than a trivial TMR implementation of the output conversion requiring three of the m_i projection output converters and three CHOOSE LEGITIMATE blocks.

In Fig. 8.14 the implementation of the totally fault-tolerant RNS based FIR filter is presented. Also in this case a minority voter can be used to obtain the correct output.

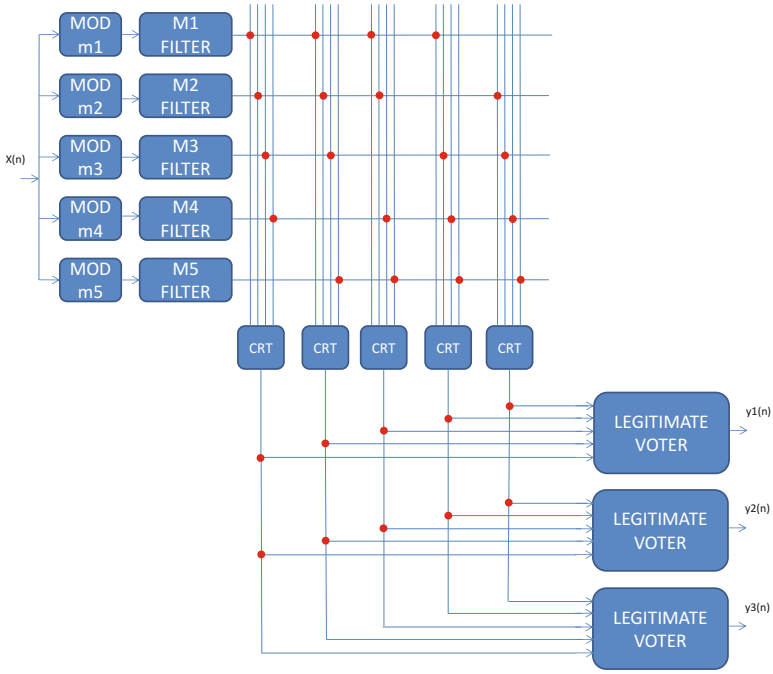

Fig. 8.14 Totally fault-tolerant RNS based FIR filter

8.5.2 Hardware Implementation

The technique presented in the previous section is used in an experiment based on a set of FIR filters. The filters are implemented on an FPGA (Xilinx Virtex-V) and the resulting area is compared to the TMR case.

Two cases are evaluated: 8 and 12 bits for both input sample and coefficients. The number of taps of the filters is from 16 to 256. The set of non-redundant moduli (the first K moduli) depends on the dynamic range of the filter, and the performance of RNS filter is strictly related to this set.

An RNS filter without error detection and correction capability requires the use of only a CRT module, while the RRNS presented in Fig. 8.12 requires a CRT block for each module of the moduli set. The TMR version of the RRNS filter (Fig. 8.13) requires three CRT blocks for each of the moduli. Instead, our proposed technique uses a CRT for each moduli, but maintains the error correction capabilities of the output conversion procedure. The area of the CRT is significant and it is one of the problems that must be faced by a designer to optimize the implementation of the filter. For filters with error detection and correction capabilities this issue is yet more significant, due to the number of CRT blocks that must be realized.

Therefore, the number of moduli of the RNS system must be taken as small as possible and therefore in our implementation we use a minimal set of moduli ($K = 3$

Table 8.7 Parameters of the filters in the experiment

Prj. name	Sample n. bits	Coeff n. bits	n. Taps	Dyn. range	Moduli set
FIR 1	8	8	64	20	129, 128, 127, 125, 121
FIR 2	8	8	128	22	257, 256, 255, 253, 251
FIR 3	8	8	256	24	257, 256, 255, 253, 251, 241
FIR 4	12	12	64	28	257, 256, 255, 253, 251, 241
FIR 5	12	12	128	30	257, 256, 255, 253, 251, 241
FIR 6	12	12	256	32	513, 512, 511, 253, 251, 241

Table 8.8 Overhead comparison for the TMR-RRNS and the presented technique

Prj. name	TMR-RRNS overhead (n. of LUTs)	Overhead for our technique	Percent (%)s
FIR 1	7401	2931	40
FIR 2	9774	3763	35
FIR 3	17,037	5780	34
FIR 4	17,127	5927	35
FIR 5	17,196	5951	35
FIR 6	19,242	7044	37

or 4) to realize the filters. In Table 8.7 the different parameters, the dynamic range of the filter, and the chosen moduli set are presented for the filter taken into account for the evaluation of area occupation.

We report the comparison between the presented techniques (Fig. 8.14) and the TMR approach of Fig. 8.13 in Table 8.8. Both methods can correct an error in all the blocks composing the filter. For the TMR-RRNS filter we define the overhead as the number of LUTs to implement the triplicated CRT blocks and three CHOOSE LEGITIMATE blocks, while for our technique we use the number of LUTs for the CRT blocks and three LEGITIMATE VOTER blocks.

Table 8.8 shows how the use of the LEGITIMATE VOTER blocks allows saving up than 33% of resource with respect to the TMR implementation of the blocks performing the error correction.

8.6 Future Trends in RNS

The RNS was originally proposed as a technique useful to implement area efficient and high-speed digital systems.

Over the years the technological advances caused the birth of new devices, the continuous evolution of the design tools, and the mutation of the most critical design figures of merit.

In earlier days, the aim of the RNS was the implementation of very fast and small circuits [26]. However, the power consumption has emerged as an increasingly important design constraint, and it is today the most important figure of merit in digital design for high performance and portable applications.

Moreover, because of the many technological and algorithmic advances, implementation results over the years and across many different tools and technologies have changed the trade-offs, sometimes significantly.

We summarize how those changes affected the two most popular platforms next.

8.6.1 ASIC Platforms

For ASICs, technology scaling and algorithmic advances affect differently the performances of different number systems.

For example, the RNS is expected to be locally bounded and to have shorter datapaths. Although this feature belongs to the RNS by definition, in recent years it has been partially hidden by the efforts in equalization of loads and delays during the synthesis and place-and-route processes. As a consequence, the wires are kept as short as possible and the wirelength differences between RNS and TCS implementations are marginal.

Especially timing driven placement algorithms are developed specifically targeting wires on critical paths rather than focusing on minimizing the total wirelength. The effects are evident in large TCS implementations characterized by unbalanced datapaths where the maximum delay can be significantly reduced. On the contrary, the effects on the smaller RNS moduli datapaths are less important, resulting in reduced advantages for the RNS.

Even if some of the benefits of the RNS seem to be reduced, the locality property of the RNS results in larger savings in terms of glitching power consumption: glitches mainly arise from delay mismatch at the input of the gates. Although the timing may be unbalanced among moduli, each modular path has highly equalized datapaths. Moreover, the timing in most of the moduli (the larger part of the system) is relaxed with the exception of the critical one that is, however, a small circuit compared with the TCS design.

These intrinsic features of the RNS result in the distribution of logic on a fewer number of levels of reduced drive strength cells. All these factors significantly affect the distribution of the delays in the circuit and, consequently, the number of spurious transitions in the nodes. The locality of the RNS becomes evident in the smaller glitch percentage due to the reduced average delay mismatches at the gates inputs.

A technological technique developed to reduce the static power consumption consists in the utilization of multi-threshold voltage (multi-V_t, or MVT) cells libraries. The most recent algorithms map high-V_t (HVT) cells[2] in an MVT design

[2]High-V_t cells are slower than standard-V_t cells, but consume significantly less power.

flow trying to compensate the different delays of input signals by changing standard-V_t cells in slower HVT cells. These algorithms are able to use the HVT cells in an RNS design to reduce both leakage and glitching power consumption by equalizing the small amount of delays unbalance and by filtering out glitches. On the contrary, in a TCS design the delays distribution is so unbalanced that these effects on glitching power are only marginal.

To show the reader how the technological advances affected the RNS and TCS implementations over the years, we present the differences in the results of the implementation of the same architecture at the distance of a decade.

In 2000, we presented an effective RNS technique to lower the power consumption in DSP applications [6]: the RNS implementation of an FIR filter showed to be about 50% smaller and power hungry than the TCS filter when designed at the same timing constraint.

In 2012, we presented the implementation of an RNS FIR filter based on one of the most recent technology processes [18]. The RNS filter is still advantageous when implemented at maximum speed, but the area and power savings are reduced to about a 30%. Moreover, as shown in Table 8.2, when the filter is implemented with a relaxed timing constraint (*ma* corner), the RNS is no longer advantageous in terms of power dissipation and area: the RNS filter is larger than the TCS.

8.6.2 FPGA Platforms

The FPGA is a very important platform to investigate RNS implementations from two different points of view:

- Higher speed and reduced logic resources by exploiting the look-up table (LUT) based architecture of the FPGAs.
- Lower power consumption.

The internal structure of the FPGAs, that is LUT based, is very suitable for the implementation of residual arithmetic. Moreover, the unavailability of embedded multipliers in early FPGAs was an important driver for the use of RNS in the implementation of parallel FIR filters.

In 2002, we showed that the RNS implementation of FIR filters on FPGAs were significantly advantageous [7] obtaining a 2.5 times less power consumption per tap and almost the same area than the TCS implementation when realized at the same timing constraint [7]. The implementation was done on a Xilinx Virtex-E FPGA. The power saving ratios were higher also than those obtained for the standard cell implementation of [6], where a power saving of 2 times was obtained per tap. The savings were mostly due to the implementation of RNS multipliers by isomorphism (LUTs plus modular adder), while in TCS multipliers were so large to compensate for the converters area.

As the FPGA architecture evolved toward a hybrid architecture containing specialized full-custom processing elements, such as multipliers, or more complex

Table 8.9 Power consumption of TCS and RNS filters implemented on FPGAs

Design	2002, Virtex-E [mW]	2012, Virtex-5 [mW]
TCS	387.9	2.01
RNS	198.7	3.15
Ratio	1.95	0.64

and flexible processing elements (e.g., Xilinx's DSP48 blocks), the advantages in implementing isomorphic multipliers in general purpose FPGA resources, such as LUTs, almost vanished.

In 2012, we compared the TCS and RNS filter implementations on a Xilinx Virtex-5 FPGA equipped with DSP48 blocks [18]. The results show that the RNS is no more a competitive solution for the device: the distributed implementation of multipliers in LUTs cannot compete with the DSP48 based TCS implementation, which resulted to be more area and power efficient.

The comparison of the results between the implementations of 2002 and 2012 is illustrated (power dissipation only) in Table 8.9.

In any case, there are some interesting classes of applications, such as space applications, in which the technology is naturally delayed, and where power savings and the lowering of the non-recurring costs are of fundamental importance.

In the outer space there are needs for devices able to work in an environment characterized by particles and high-energy electromagnetic radiations. A programmable radiation-hardened device is a good solution to reduce the fixed costs to fabricate an ASIC device.

Radiation-hardened components are based on their non-hardened equivalents and enhanced with extra circuitry to reduce the radiation damages. Because of the higher production costs and the lower market demand, radiation-hardened FPGAs tend not to incorporate the latest technological advances. For example, several families of FPGA space-certified still do not have embedded multipliers, or their number is small.

In this context, for high performance DSP requiring a large number of multipliers, such as a parallel high-order filter, the RNS implementation can still be advantageous [20].

8.7 RNS Perspectives

In the following, we point out our vision about RNS future perspectives.

First, we see an increase in parallelism: traditional microprocessors are evolving toward parallelism due to the impossibility to increase the clock rate. Therefore, we need to increase the number of cores to increase the throughput. This trend toward parallelism is also true for arithmetic cores used in general purpose processors: GPU based accelerators or vector processors (e.g., the NEON unit in ARM processors).

We see an increase in parallelism also in DSP architectures: for example, to keep up the giga-sample rates of the latest ADCs, the samples are organized in frames requiring DSP blocks with memory to use matrix-structured implementations.

Second, we see the advent of DSP partitioning techniques at number system level to map a part of an algorithm to the most efficient number system for its implementation. For example, the parts of an algorithm characterized by the absence of feedback and long linear combinations, such as high dynamic range and high-order FIR filters, and the implementation of transforms.

Third, we see a trend in the direction of fault-tolerant DSP systems. Due to the aggressive technology scaling, devices are so small that can be upset by radiations even at sea level. Because an increasing number of life critical applications (medical, automotive, avionics, etc.) are implemented in those devices, it is very important to design such system in a fault-tolerant and self-correcting manner. The RNS could be an enabling technology in this case due to its inherent parallelism that can be exploited to create the necessary redundancy.

Fourth, there is a general trend toward the implementation of DSP in floating-point to cope with the increased complexity of the DSP applications and the reduce time-to-market of the new applications. In this context, the use of RNS is at disadvantage (scaling is required, etc.).

Fifth, the RNS can help the mitigation of hot-spot problems. A hot-spot is generated by high power consumption concentrated in a limited area of the die causing the temperature to rise. Because the RNS is characterized by a quite uniform distribution of node capacitance and RNS paths produce less glitches than TCS, the temperature profile of the die is expected to be rather flat.

8.8 Conclusions

In this chapter, we presented a number of RNS implementations of DSP units, and analyzed the advantages and the disadvantages of the RNS with respect to the traditional TCS. In addition, we investigated the impact of technological advances in implementing DSP in RNS and TCS.

Although the technological advances and the modern CAD tools favor the implementation of the common case (TCS), the benefits of RNS in terms of area, delay, and power consumption are still remarkable, especially for systems implemented on ASICs.

As for FPGA platforms, RNS will still offer advantages in fault-tolerant systems and special devices used in special applications, such as space applications.

References

1. P.V. Ananda Mohan, *Residue Number Systems: Algorithms and Architectures*. (Kluwer Academic Publishers, New York, 2002)
2. F. Barsi, P. Maestrini, Error correction properties of redundant residue number system. IEEE Trans. Comput. **22**, 370–375 (1973)
3. F. Barsi, P. Maestrini, Error detection and correction by product codes in residue number system. IEEE Trans. Comput. **23**, 915–924 (1974)
4. G.L. Bernocchi, G.C. Cardarilli, A. Del Re, A. Nannarelli, M. Re, Low-power adaptive filter based on RNS components, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2007)* (2007), pp. 3211–3214
5. G. Cardarilli, M. Re, R. Lojacono, A residue to binary conversion algorithm for signed numbers, in *European Conference on Circuit Theory and Design (ECCTD'97)*, vol. 3 (1997), pp. 1456–1459
6. G.C. Cardarilli, A. Nannarelli, M. Re, Reducing power dissipation in FIR filters using the residue number system, in *Proceedings of 43rd IEEE Midwest Symposium on Circuits and Systems, 2000*, vol. 1 (2000), pp. 320–323
7. G.C. Cardarilli, A. Del Re, A. Nannarelli, M. Re, Power characterization of digital filters implemented on FPGA, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, vol. 5 (2002), pp. 801–804
8. G.C. Cardarilli, A. Del Re, A. Nannarelli, M. Re, Residue number system reconfigurable datapath, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, vol. 2 (2002), pp. 756–759
9. G.C. Cardarilli, A. Del Re, A. Nannarelli, M. Re, Impact of RNS coding overhead on FIR filters performance, in *Proceedings of 41st Asilomar Conference on Signals, Systems, and Computers* (2007), pp. 1426–1429
10. G.C. Cardarilli, A. Nannarelli, Y. Oster, M. Petricca, M. Re, Design of large polyphase filters in the quadratic residue number system, in *Proceedings of 44th Asilomar Conference on Signals, Systems, and Computers* (2010), pp. 410–413
11. R. Conway, J. Nelson, Improved RNS FIR filter architectures. *IEEE Trans. Circuits Syst. Express Briefs* **51**(1), 26–28 (2004)
12. A. Del Re, A. Nannarelli, M. Re, A tool for automatic generation of RTL-level VHDL description of RNS FIR filters, in *Proceedings of 2004 Design, Automation and Test in Europe Conference (DATE)* (2004), vol. 48, pp. 686–687
13. M.H. Etzel, W.K. Jenkins, Redundant residue number systems for error detection and correction in digital filters. IEEE Trans. Acoust. Speech Signal Process. **28**, 538–544 (1980)
14. W.K. Jenkins, The design of error checkers for self-checking residue number arithmetic. IEEE Trans. Comput. **C-32**, 388–396 (1983)
15. W. Jenkins, B. Leon, The use of residue number systems in the design of finite impulse response digital filters. IEEE Trans. Circuits Syst. **24**(4), 191–201 (1977)
16. J.V. Lopez, J.C. Sanchez, H.P. Meana, Adaptive echo canceller using a modified LMS algorithm, in *Proceedings of 2nd International Conference on Electrical and Electronics Engineering* (2005), pp. 93–96
17. A. Nannarelli, G.C. Cardarilli, M. Re, Power-delay tradeoffs in residue number system, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, vol. 5 (2003), pp. 413–416
18. M. Petricca, P. Albicocco, G.C. Cardarilli, A. Nannarelli, M. Re, Power efficient design of parallel/serial FIR filters in RNS, in *Proceedings of 46th Asilomar Conference on Signals, Systems, and Computers* (2012), pp. 1015–1019
19. S. Piestrak, A high-speed realization of a residue to binary number system converter. IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. **42**, 661–663 (1995)
20. S. Pontarelli, G.C. Cardarilli, M. Re, A. Salsano, Totally fault tolerant RNS based FIR filters, in *2008 14th IEEE International On-Line Testing Symposium* (2008), pp. 192–194

21. D. Radhakrishnan, Y. Yuan, Novel approaches to the design of vlsi RNS multipliers. IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. **39**(1), 52–57 (1992)
22. M. Re, Metodologie di conversione ingresso-uscita in processori in aritmetica finita. Ph.D. thesis, University of Rome Tor Vergata, 1996
23. A. Sengupta, S. Bandyopadhyay, G.A. Jullien, A systolic array for fault tolerant digital signal processing using a residue number system approach, in *Proceedings of the International Conference on Systolic Arrays* (1988), pp. 577–586
24. M.A. Sodestrand, W.K. Jenkins, G.A. Jullien, F.J. Taylor, *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing* (IEEE Press, New York, 1986)
25. T. Stouraitis, Efficient convertors for residue and quadratic-residue number systems. IEE Proc. G, **139**(6), 626–634 (1992)
26. N.S. Szabo, R.L. Tanaka, *Residue Arithmetic and its Applications to Computer Technology* (McGraw-Hill, New York, 1967)
27. Y. Tsuda, T. Shimamura, An improved NLMS algorithm for channel equalization, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, vol. 5 (2002), pp. 353–356
28. P.P. Vaidyanathan, Filter banks in digital communications. IEEE Circuits Syst. Mag. **1**, 4–25 (2001)
29. I.M. Vinogradov, *An Introduction to the Theory of Numbers* (Pergamon Press, New York, 1955)
30. T.V. Vu, Efficient implementation of the Chinese remainder theorem for sign detection and residue decoding. IEEE Trans. Circuits Syst. I **45**, 667–669 (1985)

Chapter 9
RNS-Based Image Processing

Nikolay Chervyakov and Pavel Lyakhov

9.1 Introduction

Images are produced by a variety of physical devices, including still and video cameras, scanners, X-ray devices, electron microscopes, radar, and ultrasound, and are used for a variety of purposes, including entertainment, medical, business, industrial, military, civil, security, and scientific. The interests in digital image processing stem from the improvement of pictorial information for human interpretation and the processing of scene data for autonomous machine perception [1].

Digital image processing starts with one image and produces a modified version of that image. A digital image is an image $f(x, y)$ that has been discretized in both spatial coordinates and brightness (intensity). The image is divided into small regions called picture elements or pixels (see Fig. 9.1) [2].

Image digitization includes image sampling (i.e., digitization of spatial coordinates (x, y)) and gray-level quantization (i.e., brightness amplitude digitization). An image is represented by a rectangular array of integers. The image sizes and the number of gray levels are usually integer powers of 2. The number at each pixel represents the brightness or darkness (generally called the intensity) of the image at that point. For example, Fig. 9.2 shows a digital image of size 8×8 with 1 byte (i.e., 8 bits $= 256$ gray levels) per pixel [1].

Nowadays, there is almost no area that is not impacted in some way by digital image processing. Examples of such applications are listed below.

1. ***Remote sensing***. Images acquired by satellites and other spacecrafts are useful in tracking Earth's resources, solar features, geographical mapping, and space image applications.

N. Chervyakov • P. Lyakhov (✉)
North Caucasus Federal University, Stavropol, Russia
e-mail: ljahov@mail.ru

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_9

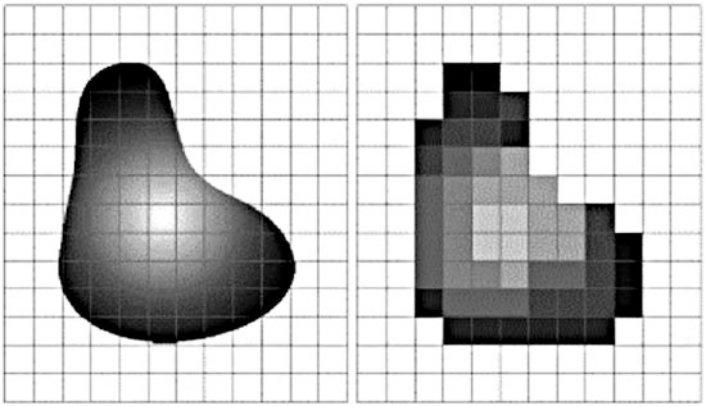

Fig. 9.1 Image digitization

1	8	219	51	69	171	81	41
94	108	20	121	17	214	15	74
233	93	197	83	177	215	183	78
41	84	118	62	210	71	122	38
222	73	197	248	125	226	210	5
35	36	127	5	151	2	197	165
196	180	142	52	173	151	243	164
254	62	172	75	21	196	126	224

Fig. 9.2 A digital image and its numerical representation

2. ***Image transmission and storage for business***. Its applications include broadcast television, teleconferencing, transmission of facsimile images for office automation, communication over computer networks, security monitoring systems, and military communications.
3. ***Medical processing***. Its applications include X-ray, cineangiogram, transaxial tomography, and nuclear magnetic resonance. These images may be used for patient screening and monitoring or for detection of tumors or other diseases in patients.
4. ***Radar, sonar, and acoustic image processing***. For example, the detection and recognition of various types of targets and the maneuvering of aircraft.
5. ***Robot/machine vision***. Its applications include the identification or description of objects or industrial parts in 3D scenes. This area is very demanding on the speed of operation as well as energy efficiency of calculations. Therefore, developers of machine vision systems one of the first pay attention to the use of modular arithmetic for this task [3].

In general, image processing operations can be categorized into four types [1].

1. ***Pixel operations***. The output at a pixel depends only on the input at that pixel, independent of all other pixels in that image. Thresholding, a process of making the corresponding input pixels above a certain threshold level white and others black, is simply a pixel operation. Other examples include brightness addition/subtraction, contrast stretching, image inverting, log, and power law.
2. ***Local (neighborhood) operations***. The output at a pixel depends on the input values in a neighborhood of that pixel. Some examples are edge detection, smoothing filters (e.g., the averaging filter and the median filter), and sharpening filters (e.g., the Laplacian filter and the gradient filter). This operation can be adaptive because results depend on the particular pixel values encountered in each image region.
3. ***Geometric operations***. The output at a pixel depends only on the input levels at some other pixels defined by geometric transformations. Geometric operations are different from global operations, such that the input is only from some specific pixels based on geometric transformation. They do not require the input from all the pixels to make its transformation.
4. ***Global operations***. The output at a pixel depends on all the pixels in an image. It may be independent of the pixel values in an image, or it may reflect statistics calculated for all the pixels, but not a local subset of pixels. A popular distance transformation of an image, which assigns to each object pixel the minimum distance from it to all the background pixels, belongs to a global operation. Other examples include histogram equalization/specification, image warping, Hough transform, and connected components.

Image processing techniques are highly variable in complexity of the mathematical apparatus. Some objectives can be achieved by performing the convolution operations (which is a combination of arithmetic multiplication and addition), while for some others should be used complicated numerical methods for solving of differential equations [4]. This fact means that the possibility of effective use of RNS for image processing depends on the type of mathematical operations used in processing algorithm. We should also note the issue of images protection using RNS. This direction is interdisciplinary therefore we list only the main works in this area. Image protection based on multiplexing technique is proposed in [5]. In the papers [6, 7] watermarking algorithms using RNS are described. Homomorphic encryption scheme to ensure privacy and security in the processing of medical images proposed in [8].

As in most other RNS applications the highest efficiency of RNS achieved in those methods and algorithms that require intensive computation of sums and products without using the significant amounts of comparisons, division, and so on. Currently, RNS is used most effectively for filtering images for different purposes, as we describe in detail below.

9.2 Edge Detection and Sharpening Filters in RNS

RNS is best used in those applications which require only the operations of addition, subtraction, and multiplication. Edge detection and sharpening of images are such applications. Edge detection is an important tool in image analysis, and is necessary for applications of computer vision in which objects need to be recognized by their outlines. An edge detection algorithm should show the locations of major edges in the image while ignoring false edges caused by noise [4]. Sharpening filter is used to enhance the edges of objects and adjust the contrast of object and background transitions. They are sometimes also used as edge detectors by combining with thresholding.

We will consider grayscale images below. The number of gray levels is integer power of 2, that is, the pixel represents the brightness or darkness. Thus, a larger number encoding pixel indicates brighter color. In standard image processing applications, grayscale image pixels are encoded by 8-bit numbers and located in the range [0, 255] where 0 represents black color and 255 represents white color. If as a result of image processing is obtained a negative number, then it is replaced by 0 (black color). In case of getting the numbers greater than 255, they are replaced by 255 (white color). Process of image filtering is to perform the convolution operation between image parts with mask filter elements, which is a square matrix of filter coefficients. Figure 9.3 shows the principle of displaying numbers in the grayscale pixel color.

Consider an image I composed of R rows and C columns of grayscale pixels. Let $I(x, y)$ denote the image pixel at position (x, y), with $0 \le x < C$ and $0 \le y < R$. The output I_f of a linear filter in position (x, y) in the spatial domain is given by:

$$I_f(x, y) = \sum_{i=0}^{d-1} \sum_{j=0}^{d-1} W_{i,j} I(x + i, y + j), \tag{9.1}$$

where

$$W = \begin{bmatrix} w_{0,0} & \cdots & w_{0,d-1} \\ \vdots & \vdots & \vdots \\ w_{d-1,0} & \cdots & w_{d-1,d-1} \end{bmatrix} \tag{9.2}$$

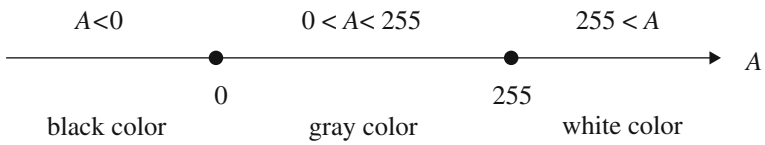

Fig. 9.3 The color of the grayscale pixel A

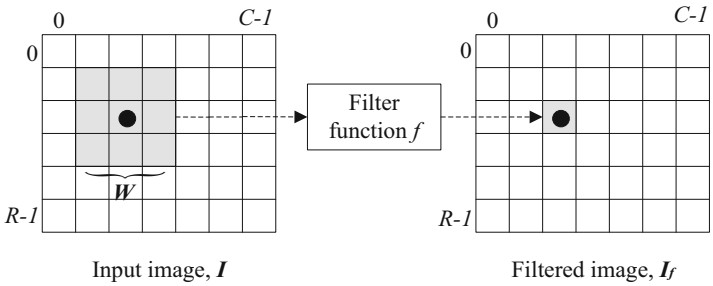

Fig. 9.4 Filtering an image with a window function f

is a $d \times d$ square matrix with $w_{i,j}$ being the filter coefficients. In order to acquire the whole filtered image I_f, the window W should slide throughout the input image (Fig. 9.4). In other words, the image filtering is a process of convolution operation performance between parts of the image and filter mask, which is a square matrix of filter coefficients.

Example 9.1. Let us assume that the part of the grayscale image has the following values of the pixels:

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix}$$

Suppose that sharpening filter is used:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

In the binary number system we obtain the following value of the pixel after the filtering:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 255 & 0 \\ 255 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix} = -1020$$

The number -1020 is obtained as the sum of multiplications of the corresponding elements of the matrices. Since the negative values are considered as 0, in this case -1020 is black color.

When using the RNS for the digital image processing particular attention should be paid to choosing the right moduli set thatwill be to give the enough dynamic

range. In the works [9] and [10] proposed moduli sets {5,7,8} and {7,15,16}, respectively, and said about adequacy of their dynamic range for digital image processing. In the following examples we will show that it is not always correct.

Example 9.2. The use of the RNS with moduli {5,7,8} gives the following result for the values from example 9.1:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 255 & 0 \\ 255 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix} =$$
$$= \begin{bmatrix} \{0,0,0\} & \{4,6,7\} & \{0,0,0\} \\ \{4,6,7\} & \{0,5,5\} & \{4,6,7\} \\ \{0,0,0\} & \{4,6,7\} & \{0,0,0\} \end{bmatrix} \times \begin{bmatrix} \{0,0,0\} & \{0,3,7\} & \{0,0,0\} \\ \{0,3,7\} & \{0,0,0\} & \{0,3,7\} \\ \{0,0,0\} & \{0,3,7\} & \{0,0,0\} \end{bmatrix} =$$
$$= \{0,2,4\} = 100.$$

Dynamic range for RNS with moduli {5,7,8} is equal to $M = 5 \cdot 7 \cdot 8 = 280$. Since the resulting number 100 is in the first half of the range $0 \leq 100 < 140$, then it is positive number in this RNS. The number 100 gives the gray color in the resulting image. Obvious difference between the results obtained shows that moduli set {5,7,8} may give a distorted image filtering results due to lack of dynamic range.

In work [10] authors use moduli set {7,15,16} which is also considered by the authors sufficient for filtering. In that study authors demonstrated that the sharpening filter in the RNS exceeds a similar device in the binary system by 30% in terms of frequency and power consumption. The following example shows the lack of dynamic range for moduli set {7,15,16}.

Example 9.3. Suppose that part of the grayscale image has the following values of the pixels:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Using the same filter, we obtain in the binary number system:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \end{bmatrix} = 1275$$

The number $1275 > 255$ and therefore considered as white color.

When image is filtered using RNS with moduli {7,15,16} we have:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \end{bmatrix} = 1275$$
$$= \begin{bmatrix} \{0,0,0\} & \{6,14,15\} & \{0,0,0\} \\ \{6,14,15\} & \{5,5,5\} & \{6,14,15\} \\ \{0,0,0\} & \{6,14,15\} & \{0,0,0\} \end{bmatrix} \times \begin{bmatrix} \{0,0,0\} & \{0,0,0\} & \{0,0,0\} \\ \{0,0,0\} & \{3,0,15\} & \{0,0,0\} \\ \{0,0,0\} & \{0,0,0\} & \{0,0,0\} \end{bmatrix} =$$
$$= \{1,0,11\} = 1275$$

Dynamic range for RNS with moduli {7,15,16} is equal to $M = 5 \cdot 7 \cdot 8 = 280$. The number 1275 is in the second half of the range $840 \leq 1275 < 1680$. Hence the final result after reverse conversion is $1275 - 1680 = -405$. This negative number gives the black color of pixel, which shows an incorrect result of filtering using moduli set {7,15,16}.

The above examples show the importance of choosing the correct RNS moduli set that provides sufficient dynamic range. RNS with insufficient dynamic range leads to incorrect results. The following formula can be used to determine the range of RNS:

$$M > 2 \cdot 255 \cdot \max\{\pi, \nu\} \tag{9.3}$$

where $\pi = \sum_{a_{i,j}>0} a_{ij}$ is the sum of the positive filter mask coefficients; $\nu = \left| \sum_{a_{ij}<0} a_{ij} \right|$ is the absolute value of sum of the negative filter mask coefficients.

Let us consider the use of RNS for image processing with sharpening filters $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$, $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$, and with edge detection filter $\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$. Further in the text these filters will be denoted as filter-1, filter-2, and filter-3, respectively. Applying formula 1 for these filters provides the following requirements for RNS dynamic range.

For the filter-1 we have $M > 2 \cdot 255 \cdot \max\{5, |-4|\} = 2550$.
For the filter-2 we have $M > 2 \cdot 255 \cdot \max\{9, |-8|\} = 4590$.
For the filter-3 we have $M > 2 \cdot 255 \cdot \max\{8, |-8|\} = 4080$.

The correct filtering by all these filters can be provided if the range of RNS is $M > 4590$. Unfortunately there is no possibility of using three RNS moduli that have convenient forms 2^n or $2^n \pm 1$ and requires no more than 4 bits ($m_i \leq 16$) for this case. There is necessary to use the RNS with three modules which require no more than 5 bits (for example, {17,31,32} or {15,31,32}) or use the four RNS modules, but stay within 4-bit representation of the modules. The last case may be implemented by RNS {5,7,9,16}. Dynamic range for this RNS is $M = 5040$

and satisfies the requirement given above. Let us show the correctness of the RNS choice, for example, used above.

Example 9.4. Using the RNS with moduli {5,7,9,16} for data processing from Example 9.1 gives the following result:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 255 & 0 \\ 255 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix} =$$
$$= \begin{bmatrix} \{0,0,0,0\} & \{4,6,8,15\} & \{0,0,0,0\} \\ \{4,6,8,15\} & \{0,5,5,5\} & \{4,6,8,15\} \\ \{0,0,0,0\} & \{4,6,8,15\} & \{0,0,0,0\} \end{bmatrix} \times$$
$$\times \begin{bmatrix} \{0,0,0,0\} & \{0,3,3,15\} & \{0,0,0,0\} \\ \{0,3,3,15\} & \{0,0,0,0\} & \{0,3,3,15\} \\ \{0,0,0,0\} & \{0,3,3,15\} & \{0,0,0,0\} \end{bmatrix} =$$
$$= \{0,2,6,4\} = 4020$$

The number 4020 is in the second half of the range $2520 \leq 4020 \leq 5040$. Hence the final result after reverse conversion is $4020 - 5040 = -1020$. This negative number gives the black color of pixel that is exactly the same as the result obtained in the binary system.

Example 9.5. Using the RNS with moduli {5,7,9,16} for data processing from Example 9.2 gives the following result:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \end{bmatrix} =$$
$$= \begin{bmatrix} \{0,0,0,0\} & \{4,6,8,15\} & \{0,0,0,0\} \\ \{4,6,8,15\} & \{0,5,5,5\} & \{4,6,8,15\} \\ \{0,0,0,0\} & \{4,6,8,15\} & \{0,0,0,0\} \end{bmatrix} \times$$
$$\times \begin{bmatrix} \{0,0,0,0\} & \{0,0,0,0\} & \{0,0,0,0\} \\ \{0,0,0,0\} & \{0,3,3,15\} & \{0,0,0,0\} \\ \{0,0,0,0\} & \{0,0,0,0\} & \{0,0,0,0\} \end{bmatrix} =$$
$$= \{0,1,6,11\} = 1275$$

The number 1275 is in the first half of the range $0 \leq 1275 \leq 2519$ and therefore is positive. This positive number gives the white color of pixel that is exactly the same as the result obtained in the binary system.

And now we will show the simulation results of different grayscale images processing obtained by using different filters and different number systems. We used three RNS with moduli {5,7,8}, {7,15,16}, and {5,7,9,16}, as well as traditional

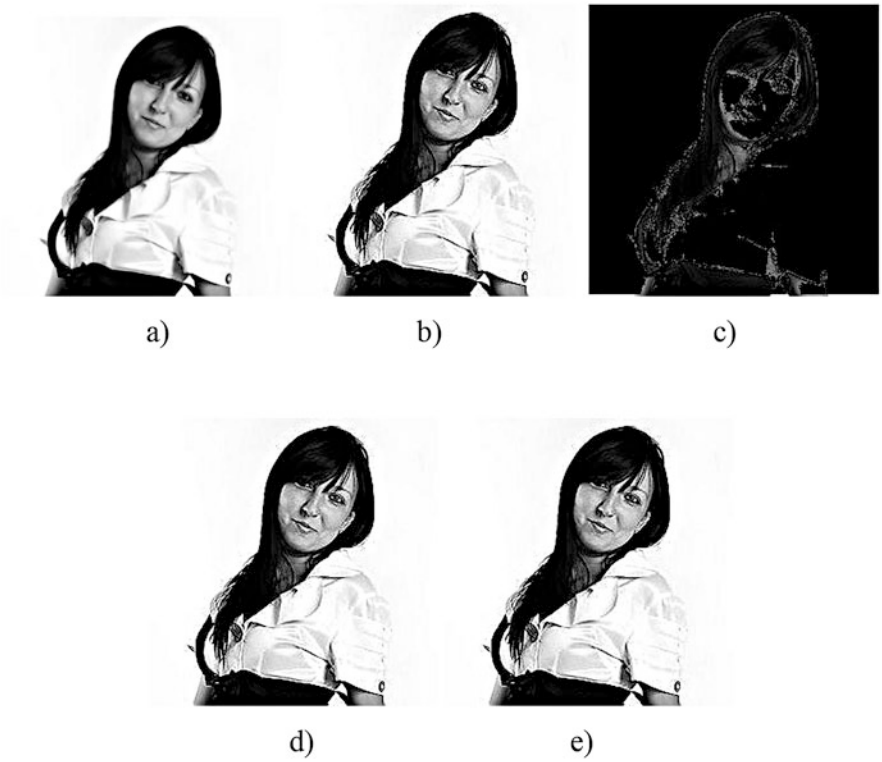

Fig. 9.5 The result of "lady" image processing using filter-1: (**a**) the original image, (**b**) image processed in the binary number system, (**c**) image processed using RNS {5,7,8}, (**d**) image processed using RNS {7,15,16}, (**e**) image processed using RNS {5,7,9,16}

binary number system for modeling. The following masks used as filters. We used filter-1, filter-2, and filter-3 for image processing. All calculations were performed using MATLAB.

The results of applying filter-1 for "lady" image are presented in Fig. 9.5. Figure 9.5a shows the original grayscale image. Figure 9.5b shows the result of image filtering in the binary system. Figure 9.5c–e shows the results of image filtering using RNS with moduli {5,7,8}, {7,15,16}, and {5,7,9,16}, respectively. Figure 9.5c shows a very poor quality of an image processing using RNS {5,7,8}. This fact is explained by too small dynamic range of that RNS, which leads to serious distortions of the image pixels. Results of processing using RNS {7,15,16} and {5,7,9,16} are indistinguishable from the image processed in the binary number system visually.

The results of applying filter-2 for "house" image are presented in Fig. 9.6. The results of the image processing with filter-2 are analogous to the case of processing with a filter-1.

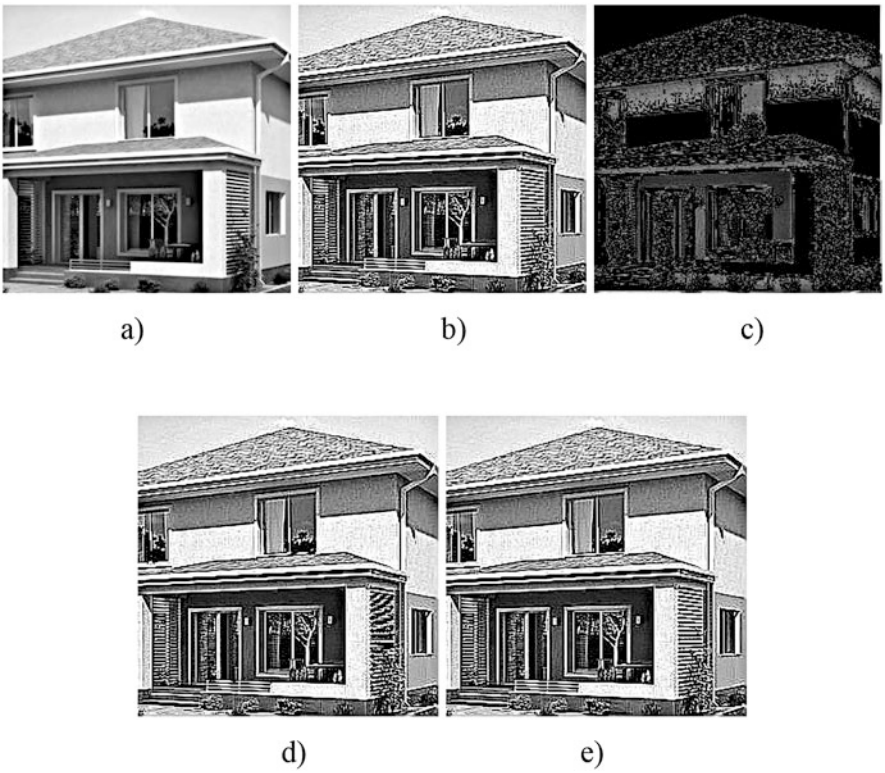

Fig. 9.6 The result of "house" image processing using filter-2: (**a**) the original image, (**b**) image processed in the binary number system, (**c**) image processed using RNS {5,7,8}, (**d**) image processed using RNS {7,15,16}, (**e**) image processed using RNS {5,7,9,16}

The results of applying filter-3 for "tulips" image are presented in Fig. 9.7. Figure 9.7a shows the original grayscale image. Figure 9.7b shows the result of image filtering in the binary system. Figures 9.7c–e shows the results of image filtering using RNS with moduli {5,7,8}, {7,15,16}, and {5,7,9,16}, respectively. Figure 9.7c–e is not very different from Fig. 9.7b visually, but in reality Figs. 9.7d and 9.6e are identical to Fig. 9.7b, whereas Fig. 9.7c is different from 9.7b.

For the quantitative determination of image processing quality using different RNS moduli sets we used two numerical characteristics.

1. *PSNR* or Peak Signal to Noise Ratio between two images [11]. This characteristic is calculated by the formula:

$$PSNR = 10\log_{10}\left(\frac{R^2}{MSE}\right), \tag{9.4}$$

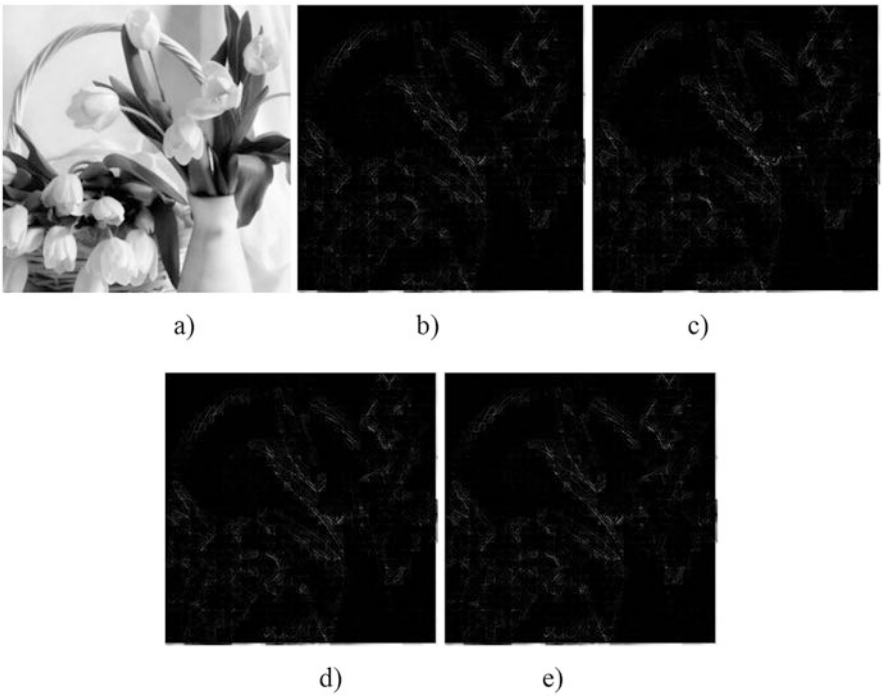

Fig. 9.7 The result of "tulips" image processing using filter-3: (**a**) the original image, (**b**) image processed in the binary number system, (**c**) image processed using RNS {5,7,8}, (**d**) image processed using RNS {7,15,16}, (**e**) image processed using RNS {5,7,9,16}

where $MSE = \frac{\sum_{M,N} [I_1(m,n) - I_2(m,n)]^2}{M \cdot N}$ is mean square error of comparing the image quality; R is the maximum amplitude of the input image. Since the value of *PSNR* has a logarithmic nature, unit of it measure is decibel (dB). Larger the *PSNR* value indicates the better image quality and for identical images $PSNR = \infty$. In the investigation of algorithms for compression and denoise grayscale images the *PSNR* values between 20 dB and 50 dB are considered to be practically appropriate. For our case of filtering using RNS with different moduli sets we calculate the *PSNR* value between the image obtained using traditional binary number system and an image obtained using RNS.

2. *SSIM* or Structural SIMilarity index between two images which is defined on the basis of full comparison of the original and the resulting images [12]. This characteristic is calculated by the formula:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}, \tag{9.5}$$

Table 9.1 Simulation results of image processing using RNS with different moduli sets

Image	Filter	RNS moduli					
		{5,7,8}		{7,15,16}		{5,7,9,16}	
		PSNR, dB	SSIM	PSNR, dB	SSIM	PSNR, dB	SSIM
"Lady"	Filter-1	1.5188	0.1384	∞	1	∞	1
	Filter-2	1.5304	0.1255	43.3496	0.9998	∞	1
	Filter-3	29.8721	0.9806	∞	1	∞	1
"House"	Filter-1	4.3506	0.1010	∞	1	∞	1
	Filter-2	4.1741	0.0535	23.2278	0.9879	∞	1
	Filter-3	27.6603	0.9683	∞	1	∞	1
"Tulips"	Filter-1	3.2489	0.0963	∞	1	∞	1
	Filter-2	3.2068	0.0628	42.1221	0.9997	∞	1
	Filter-3	35.2163	0.9899	∞	1	∞	1

where μ_x is the mean value x, μ_y is the mean value y, σ_x^2 is dispersion x, σ_y^2 is dispersion y, σ_{xy} is covariance x and y, $c_1 = (k_1L)^2$, $c_2 = (k_2L)^2$ are two variables, L is the pixels dynamic range, $k_1 = 0,01$ and $k_2 = 0,03$ are constants. The value of *SSIM* is between 0 and 1 and for identical equal images is 1. For our case of filtering using RNS with different moduli sets we calculate the *SSIM* value between the image obtained using binary number system and an image obtained using RNS.

Table 9.1 shows the results of quality of the processed images using four filters in the RNS with different moduli sets. Analyzing the results, it can be concluded that the moduli set {5,7,9,16} shows absolutely accurate result in all cases same as the result of processing in the binary number system. Moduli set {5,7,8} does not guarantee the correct work ($PSNR < 20\,dB$) of sharpening filters (filter-1 and filter-2), and may limit applies ($PSNR > 20\,dB$) in the image edge detection (filter-3). Moduli set {7,15,16} does not caused errors in image filtering with filter-1 and filter-3. Image quality deteriorates when filter-2 used, but was at an acceptable level for practical use ($PSNR > 20\,dB$). It should be recognized, however, that the occurrence of errors by using RNS with modules {7,15,16} is theoretically possible (see Example 9.3) and during filtration with the filter-1 or filter-3. The absence of errors in the processing of "lady," "house," and "tulips" images is explained to lack of abrupt changes in the values of the adjacent pixels from 0 to 255 and back, which may not happen in the processing of any other images.

We have demonstrated that sharpening and edge detection filters are the simplest to implement in RNS. These devices need to perform only the operations of addition and multiplication of integers. However during their designing must be considered the sufficiency of the system dynamic range since too small range may lead to a distortion of processing results. Next we will consider more complicated image filtering devices in RNS.

9.3 Smoothing Filters in RNS

Smoothing filters perform replacement operation of each pixel value in an input image by the average value of its neighbors, including itself. This has an effect of eliminating pixel values that are unrepresentative of their surroundings. Like other convolution filters, it is based around a kernel, which represents the shape and size of the neighborhood to be sampled in calculation [1]. Smoothing filters are used for image denoising and quality improving. Examples of smoothing filter masks are shown in Table 9.2.

As one can see from Table 9.2 smoothing filters are more complicated than sharpening and edge detection filters to implement in RNS due to the necessity of the division operation by a constant. In the [13] proposed hardware implementation of a smoothing filter in which the division operation is performed outside of the RNS (Fig. 9.8).

Modeling of this architecture (in combination with edge detection filter) in [13] showed the correct operation in MATLAB. Also, hardware implementation of this architecture was performed on HDL. However, the circuit shown in Fig. 9.8 does

Table 9.2 Examples of smoothing filter masks

Filter	Filter masks
Binomial	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}; \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$
Gaussian	$\frac{1}{15}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 1 \end{bmatrix}; \frac{1}{351}\begin{bmatrix} 1 & 5 & 8 & 5 & 1 \\ 5 & 21 & 34 & 21 & 5 \\ 8 & 34 & 55 & 34 & 8 \\ 5 & 21 & 34 & 21 & 5 \\ 1 & 5 & 8 & 5 & 1 \end{bmatrix}$

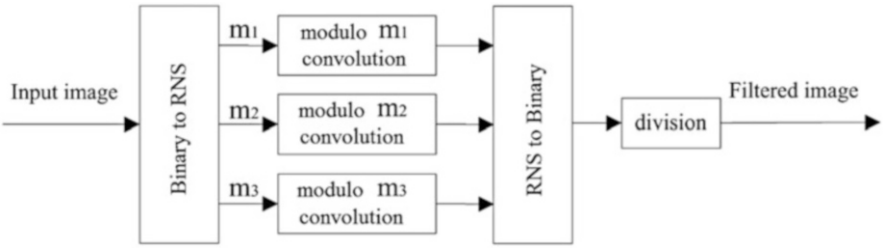

Fig. 9.8 Smoothing filter architecture in RNS similar to [13]

not allow to fully realize RNS potential for image processing. Problematic division operation in the smoothing filter can be effectively implemented in the RNS under certain conditions.

If one of the RNS modulo is the same as divider from smoothing filter mask, then computations can be arranged as follows. Suppose that RNS with three modules $\{m_1,\ m_2,\ m_3\}$ satisfies the condition $m_1 = v$ where v is divider from smoothing filter mask and $m2m3 > 255$. Then the division operation by m_1 can be replaced by subtraction and multiplication. For dividing the number $A = (a_1, a_2, a_3)$ by m_1 with the remainder first calculate $A - a_1 = (a_1,\ a_2,\ a_3) - \left(a_1,\ |a_1|_{m_2},\ |a_1|_{m_3}\right) = \left(0,\ a_2',\ a_3'\right)$. The second step is multiplication of the number $A - a_1$ by multiplicative inversions of m_1 modulo m_2 and m_3. This operation can be considered as computing in RNS with two modules $\{m_2, m_3\}$:

$$(A - a_1) \cdot m_1^{-1} = \left(a_2',\ a_3'\right) \cdot \left(\left|m_1^{-1}\right|_{m_2},\ \left|m_1^{-1}\right|_{m_3}\right) = \left(a_2'',\ a_3''\right) \tag{9.6}$$

RNS with modules $\{m_2,\ m_3\}$ provides sufficient dynamic range since the result of grayscale image smoothing filtration lies in the range from 0 to 255 and, as mentioned above, $m_2 m_3 > 255$. The resulting number (a_2'', a_3'') must be converted to binary number system. Here the reverse conversion operation should be performed only for two modules (instead of three modules for the initial RNS), which further simplifies the calculations.

Example 9.6. Divide the number 2000 by 15 with the remainder in the RNS {15,16,17}. The number 2000 is represented in this RNS as (5,0,11). Calculations are displayed in Table 9.3.

The resulting number (5,14) in RNS $\{m_2,\ m_3\} = \{16,17\}$. Check the results in a positional number system. The result of the division in the decimal number system is $\frac{2000}{15} = 133,33\cdots \approx 133$. The rounded value of quotient is 133, which represented in RNS with modules {16,17} as {5,14}, which is exactly the same as the result of a calculations.

Since in this method partial quotient is actually searched then the result of smoothing filtration may be different by 1 from the calculation result in the binary number system. This happens because of rounding down in this method instead of

Table 9.3 Example of division by modulo m_1 with remainder in RNS

	RNS modules		
Decimal representation	$m_1 = 15$	$m_2 = 16$	$m_3 = 17$
Initial number	5	0	11
Subtracting modulo $m_1 = 15$ remainder	5	5	5
	0	11	6
Multiplication by multiplicative inverse 15^{-1} (division by 15)	–	15	8
	–	5	14
Quotient	–	5	14

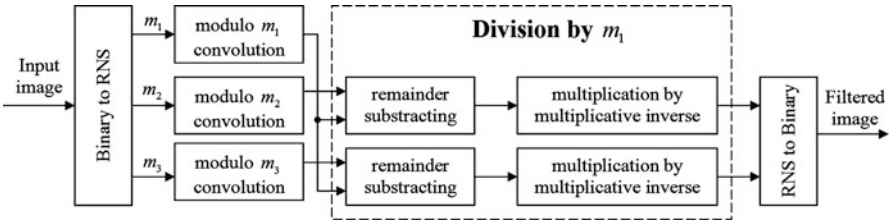

Fig. 9.9 Smoothing filter architecture in RNS with replacement of the division operation to subtraction and multiplication

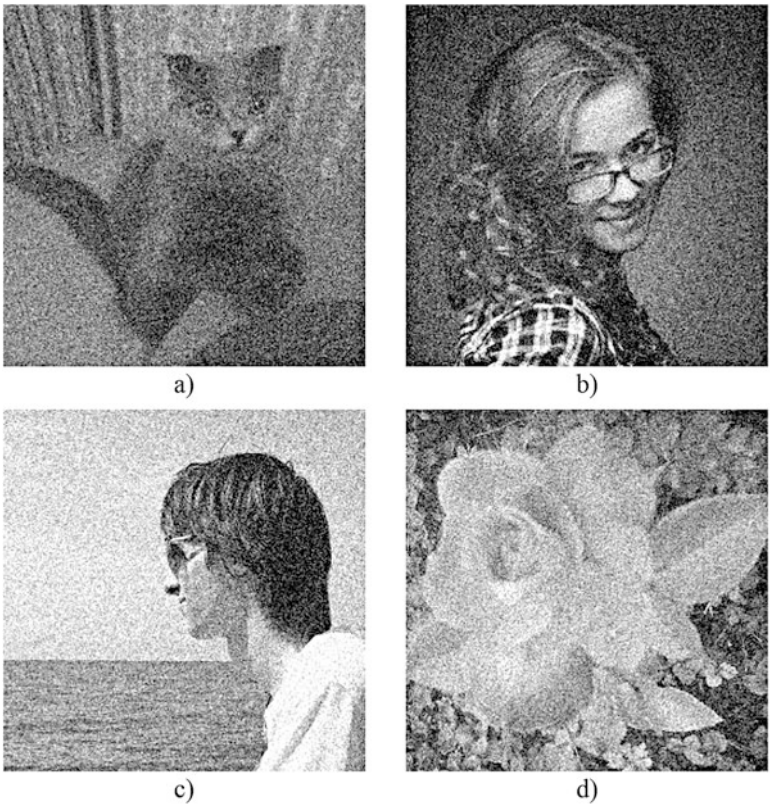

Fig. 9.10 Noisy images ($\sigma^2 = 0.05$): (**a**) "cat," (**b**) "lady," (**c**) "boy," (**d**) "flower"

the usual rounding in the binary number system. The process of smoothing filtration with the division operation implementation in RNS is shown in Fig. 9.9.

Consider working of smoothing filters on the example of processing four noisy images shown in Fig. 9.10. Images from Fig. 9.10 were processed with smoothing filter

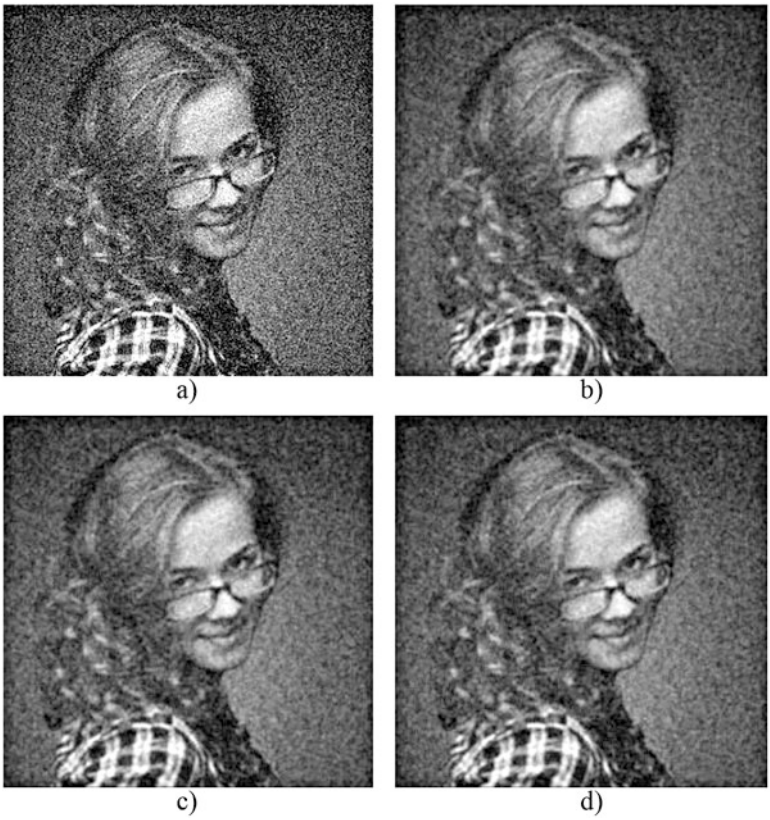

Fig. 9.11 Image "lady" smoothing: (**a**) input image; (**b**) image processed in the binary system; (**c**) image processed in the RNS {15,16,17} using architecture from Fig. 9.8; (**d**) image processed in the RNS {15, 16, 17} using architecture from Fig. 9.9

$$\frac{1}{15}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

in the binary system and in RNS {15,16,17} using architectures shown in Figs. 9.8 and 9.9. All calculations were performed using MATLAB. Figure 9.11 shows an example of image "lady" processing by these three methods. We can see that the results of smoothing in RNS by using both architectures are not differ visually from the results obtained in the binary system. Actually, images 8.11b and 10c are identical and image 8.11d is slightly different from them. The values of PSNR for processed images are shown in Table 9.4. We calculate the PSNR between the image obtained using traditional binary number system and an image obtained using both RNS architectures.

Table 9.4 Quality of images smoothing in RNS {15,16,17} using different architectures compared with the smoothing in the binary system

Image	PSNR, dB	
	Architecture from Fig. 9.8	Architecture from Fig. 9.9
"Cat"	∞	51.4173
"Lady"	∞	51.4609
"Boy"	∞	51.4334
"Flower"	∞	51.4005

Table 9.4 shows that the quality of the smoothing filtering using the architecture from Fig. 9.9 is very high ($PSNR > 50$) in all cases. This means that this approach may be very widely applied in practice since this level of distortion is almost imperceptible to the human eye.

9.4 Image Processing in RNS Using Wavelets

Signal processing using wavelets is a very promising direction nowadays, which is an alternative to the traditional Fourier transform. The Fourier transform allows you to quickly and easily receive information about the frequency component of the signal. Discrete Cosine Transform (DCT) is associated with the Discrete Fourier Transform (DFT) and is widely used in the image processing. For example, Joint Photographic Experts Group (JPEG) and Moving Picture Experts Group (MPEG) data compression algorithms are based on DCT. In the work [14] an adaptive image coder used DCT and Lattice Vector Quantizer based on RNS is proposed. The main advantage of the signal processing using wavelets is the possibility of fast calculation both frequency and local features of the processed signal. Ability to quickly and accurately obtain a time-frequency information about the object led to the fact that currently wavelets are used in image, voice, and video processing [15]. Wavelets are widely applied for denoising and data compression tasks nowadays [4].

Currently the main method of wavelet signal processing is the use of Discrete Wavelet Transform (DWT) which is based on the FIR-filters. FIR-filters, in turn, can be very efficiently implemented in RNS. DWT is a multiresolution (or multiscale) representation of information. Figure 9.12 shows an example of a three-level forward DWT implementation based on a two-channel recursive filterbank, where $h_0(n)$ and $h_1(n)$ are lowpass and highpass analysis filters, respectively, and the block $\downarrow 2$ represents the downsampling operator by a factor two. The input signal $x(n)$ is recursively decomposed into a total of four subband signals: a coarse signal $c_3(n)$ and three detail signals $d_1(n)$, $d_2(n)$, and $d_3(n)$ of three resolutions.

Figure 9.13 shows an implementation of a three-level inverse DWT based on a two-channel recursive filterbank, where $\tilde{h}_0(n)$ and $\tilde{h}_1(n)$ are lowpass and highpass synthesis filters, respectively, and the block $\uparrow 2$ represents the upsampling operator by a factor two. The four subband signals $c_3(n)$, $d_3(n)$, $d_2(n)$, and $d_1(n)$ are recursively combined to reconstruct the output signal $x(n)$.

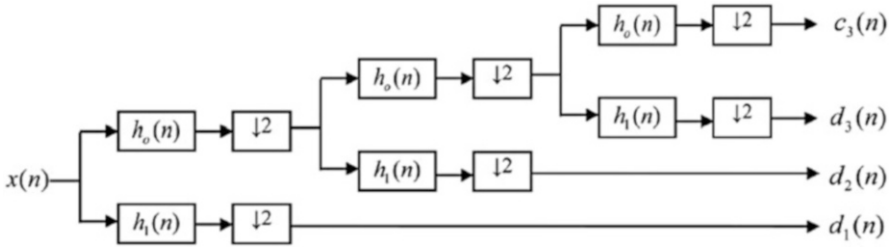

Fig. 9.12 A three-level forward DWT via a two-channel iterative filterbank

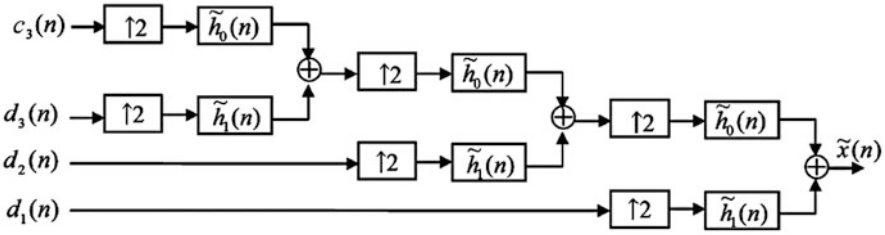

Fig. 9.13 A three-level inverse DWT via a two-channel iterative filterbank

The four FIR-filters $h_0(n)$, $h_1(n)$, $\tilde{h}_0(n)$, and $\tilde{h}_1(n)$ must satisfy the following conditions:

$$h_1(n) = (-1)^n h_0(n) \tag{9.7}$$

$$\tilde{h}_0(n) = h_0\,(1-n) \tag{9.8}$$

$$\tilde{h}_1(n) = (-1)^n h_0\,(1-n) \tag{9.9}$$

If the conditions (9.7), (9.8), and (9.9) are fulfilled, then the output of the inverse DWT is identical to the input of the forward DWT and the resulting DWT is an orthonormal transform [4].

The practical design of wavelet filters in RNS is typically done by conversion to RNS filterbank coefficients of the wavelets over the field of real or complex numbers after rounding. The coefficients of wavelet filter in the real number format may be calculated in many mathematical software packages, such as MATLAB.

Example 9.7. We consider DWT filterbank generated by Daubechies wavelet Db4. This filterbank coefficients are shown in Fig. 9.14 and were computed using MATLAB. In Table 9.5 we show the conversion to RNS $\{127, 128, 129\}$ coefficients of the filter $h_0(n)$. Coefficients of other filters $h_1(n)$, $\tilde{h}_0(n)$, and $\tilde{h}_1(n)$ are obtained from the filter $h_0(n)$ coefficients by simple permutations and sign change according

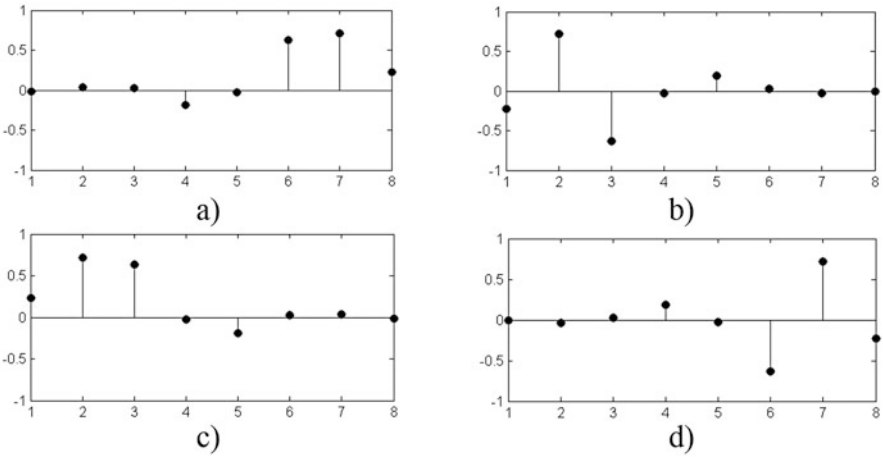

Fig. 9.14 Filterbank generated by wavelet Db4 coefficients: (**a**) $h_0(n)$; (**b**) $h_1(n)$; (**c**) $\tilde{h}_0(n)$; (**d**) $\tilde{h}_1(n)$

Table 9.5 Representation of filter $h_0(n)$ coefficients in different formats

Filter coefficients	Exact value	Format		
		Binary (12, 10)	Integer	RNS {127, 128, 129}
b_0	−0,0105974017 . . .	111111110101	−11	{116, 117, 118}
b_1	0,0328830116 . . .	000000100010	34	{34, 34, 34}
b_2	0,0308413818 . . .	000000100000	32	{32, 32, 32}
b_3	−0,1870348117 . . .	111101000000	−192	{62, 64, 66}
b_4	−0,0279837694 . . .	111111100011	−29	{98, 99, 100}
b_5	0,6308807679 . . .	001010000110	646	{11, 6, 1}
b_6	0,7148465705 . . .	001011011100	732	{97,92, 87}
b_7	0,2303778133 . . .	000011101100	236	{109, 108, 107}

to formulas (9.7)–(9.9).The values in Table 9.5 obtained by rounding exact values of the filter coefficients to 12-bit numbers, in which two bits are used to store the sign.

Since the image is a two-dimensional data array, the need for the twofold application of filters arises. Figure 9.15 shows the image filtering scheme. The processing occurs in two stages: at the first stage, the lines of the image are processed, and the columns are processed at the second stage. The filterbank, which reconstructs the image, is similarly implemented for this case. When the image processed in accordance with the scheme shown in Fig. 9.15 is synthesized, at first it is necessary to reconstruct the columns and, next, the lines.

Figures 9.16 and 9.17 show an example of three-level hierarchy wavelet decomposition of the popular test image Lena from coarse to fine resolutions [4].

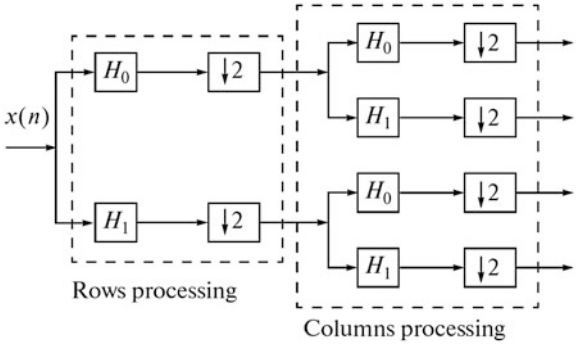

Fig. 9.15 Structure of the image processing filterbank

Wavelets can be regarded as "mathematic microscopes" that permit one to "zoom in" and "zoom out" of images at multiple resolutions.

Architectures for practical implementation of DWT and inverse DWT in RNS developed in the works [16, 17]. In the paper [16] authors developed architectures of DWT FIR-filters which are based on index-transformation over Galois fields for each filter coefficient multiplication. In the work [17] performance improvement up to 156.27% for wavelet filters architectures in RNS based on Distributed Arithmetic over the filters in the binary system is shown.

As can be seen from Example 9.7, the use of wavelets over the fields of real and complex numbers can lead to rounding errors, especially noticeable in systems with small ranges. Errors can be avoided by using finite-fields wavelets [18]. However, finite-field wavelets have a slightly different application area. Unlike wavelets over fields of real and complex numbers, finite-fields wavelets are not used for denoising or compression of the signals. The main application of finite-field wavelets is a cryptographic protection of data [19, 20]. Using finite-field wavelets in computing RNS structures seems natural, since finite-ring and finite-field arithmetic forms the basis for RNS.

Finite fields (Galois fields) can be divided into two types: prime fields $GF(p)$ and polynomial fields $GF(p^n)$, where $n > 1$ is natural number. The prime finite field $GF(p)$ contains a number of elements that are equal to the prime number p. Any finite field of p elements is isomorphical to the residue set $\{0, 1, 2, \ldots, p-1\}$, therefore, the addition, multiplication, and subtraction operations in $GF(p)$ can be considered as similar operations on $\bmod p$ integers. The arithmetic of polynomial fields $GF(p^n)$ is more complex and based on properties of polynomials over $GF(p)$. Next, we will consider only prime fields $GF(p)$.

Let us have the finite field $GF(p)$. Let us determine the vector space V, the elements of which are the vectors over the field $GF(p)$. Let us suppose that this space can be represented as a direct sum of two subspaces:

$$V = V_0 \oplus W_0,\ V_0 \cap W_0 = \{0\} \tag{9.10}$$

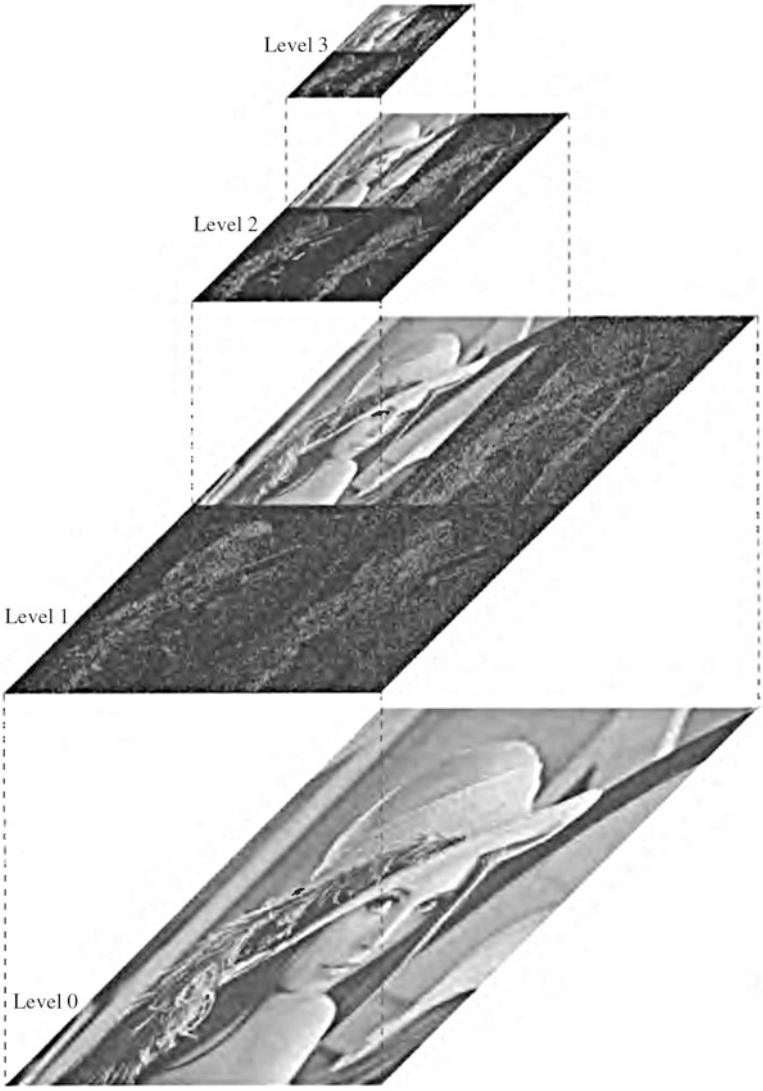

Fig. 9.16 A three-level hierarchy wavelet decomposition of the 512×512 Lena image. Level 1 (512×512) is the one-level wavelet representation of the original Lena at Level 0; Level 2 (256×256) shows the one-level wavelet representation of the lowpass image at Level 1; and Level 3 (128×128) gives the one-level wavelet representation of the lowpass image at Level 2

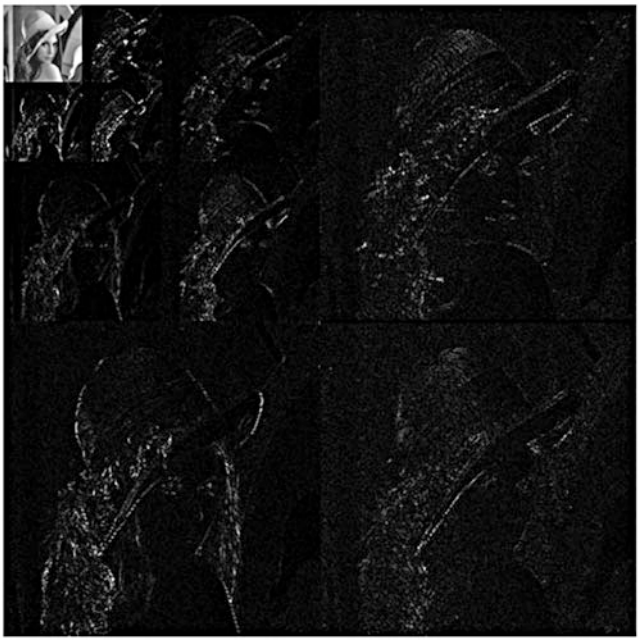

Fig. 9.17 A three-level wavelet representation of the Lena image generated from the top view of the three-level hierarchy wavelet decomposition in Fig. 9.16

Then, mother's wavelet $\psi(x)$ and the scaling function $\varphi(x)$ which determine the wavelet transform over the finite field $GF(p)$ should meet the relationships [18]

$$V_0 = \overline{span\,\{\varphi\,(n-2j)\}}, \forall j \in Z \tag{9.11}$$

$$W_0 = \overline{span\,\{\psi\,(n-2j)\}}, \forall j \in Z \tag{9.12}$$

and, in addition, the orthonormal property of the basis

$$\langle \varphi\,(n-2m)\,,\ \varphi\,(n-2k)\rangle = \delta\,(m-k)\,, \forall m,\ k \in Z \tag{9.13}$$

$$\langle \psi\,(n-2m)\,,\ \psi\,(n-2k)\rangle = \delta\,(m-k)\,, \forall m,\ k \in Z \tag{9.14}$$

$$\langle \varphi\,(n-2m)\,,\ \psi\,(n-2k)\rangle = 0, \forall m,\ k \in Z \tag{9.15}$$

The wavelet transform over the finite field $GF(p)$ is the imaging that associates the vector $x(m)$ with the sequence of coefficients $\langle x(m),\ \psi\,(m-2k)\rangle$. The backward transformation is carried out by the formula

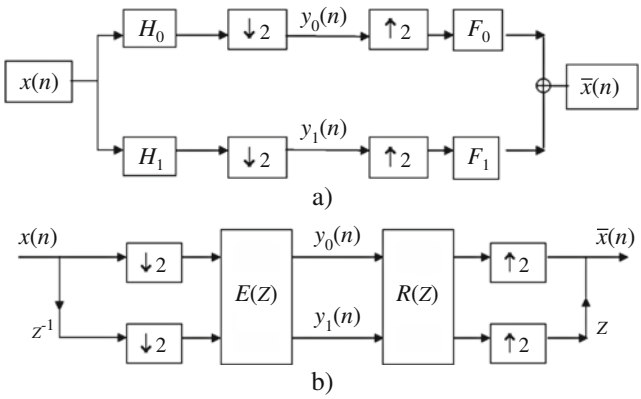

Fig. 9.18 Two-channel filterbank of the DWT: (**a**) direct form; (**b**) polyphase form

$$x(n) = \sum_{k \in Z} \langle x(m),\ \varphi\,(m-2k)\rangle\, \varphi\,(n-2k) + \sum_{k \in Z} \langle x(m),\ \psi\,(m-2k)\rangle\, \psi\,(n-2k) \tag{9.16}$$

Figure 9.18a shows the two-channel filterbank of the DWT over $GF(p)$. The same filterbank can be represented in the polyphase form (Fig. 9.18b) [21]. The following matrix is associated with this filterbank:

$$E(z) = \begin{pmatrix} E_{00}(z) & E_{01}(z) \\ E_{10}(z) & E_{11}(z) \end{pmatrix} \tag{9.17}$$

Its elements belong to the polynomial ring $F(z)$. In order to that the filter bank should possess the perfect reconstruction property, it is necessary that the matrix $E(z)$ should be paraunitary; i.e.

$$E^T\left(z^{-1}\right) E(z) = I \tag{9.18}$$

where I is an identity matrix [22].

A necessary and sufficient condition of perfect reconstruction is the fulfillment of the relationship

$$E_{00}(z)E_{00}\left(z^{-1}\right) + E_{01}(z)E_{01}\left(z^{-1}\right) = I \tag{9.19}$$

between the elements of the matrix (9.17).

The filters H_0 and H_1 can be determined by the formulas

$$H_0(z) = E_{00}\left(z^2\right) + z^{-1}E_{01}\left(z^{-2}\right) \tag{9.20}$$

$$H_1(z) = E_{10}\left(z^2\right) + z^{-1}E_{11}\left(z^{-2}\right) \tag{9.21}$$

The filters F_0 and F_1 are determined from the perfect reconstruction conditions [21]:

$$F_0(z) = H_1(-z), \; F_1(z) = -H_0(-z) \tag{9.22}$$

The construction of the filter bank in Fig. 9.18 is reduced to finding the polynomials $A(z) = \sum_{i=0}^{M} a_i z^i$, $a_0 \neq 0$ and $B(z) = \sum_{i=0}^{M} b_i z^i$, $b_M \neq 0$ from the ring of polynomials $F(z)$, which meet the condition

$$A(z)A\left(z^{-1}\right) + B(z)B\left(z^{-1}\right) = 1 \tag{9.23}$$

Each pair of polynomials $A(z)$ and $B(z)$ determines the polynomials E_{00} and E_{01} by the formulas [18]

$$\begin{aligned} a_i &= e_{0i}, \\ e_{1i} &= 0, \text{ for } i = 0, \ldots, N - M - 1 \\ b_i &= e_{1(N-M+i)}, \text{ for } i = 0, \ldots, M. \end{aligned} \tag{9.24}$$

Example 9.8. Let us construct orthogonal third order filter banks for fields $GF(7)$, $GF(11)$, $GF(13)$, $GF(17)$, $GF(19)$, and $GF(199)$. Since $2N + 1 = 3$, then $N = 1$. The construction will be performed for $M = 1$. In field $GF(7)$, e.g., the polynomials $A(z) = 1 + 6z$ and $B(z) = 2 + 4z$ meet condition (9.23). The construction process of the polynomials $E_{ij}(z)$, $i = 0,1$, $j = 0,1$, and of the analyzing and synthesizing filters is schematically shown in Fig. 9.19. This result and the filters for $GF(11)$, $GF(13)$, $GF(17)$, $GF(19)$, and $GF(199)$ constructed by a similar method are summarized in Table 9.6.

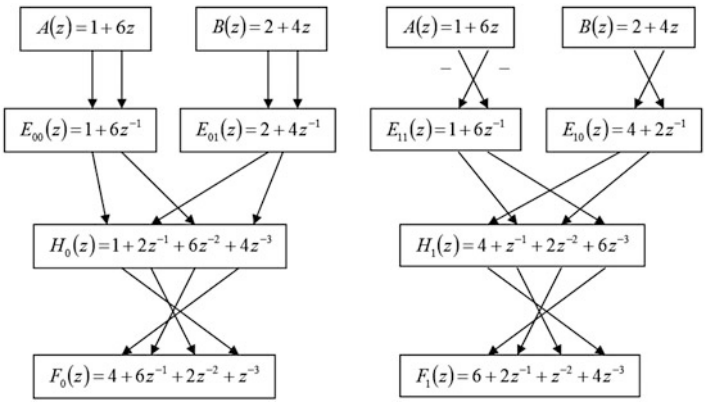

Fig. 9.19 Construction scheme of the analyzing (H_0 and H_1) and synthesizing ($_{F0}$ and $_{F1}$) filters from polynomials A(z) and B(z) cover $GF(7)$

Table 9.6 [!p]Orthogonal filter banks over finite fields

$GF(p)$	$A(z)$	$B(z)$	Analyzing filters	Synthesizing filters
$GF(7)$	$1+6z$	$2+4z$	$H_0(z)=1+2z^{-1}+6z^{-2}+4z^{-3}$	$F_0(z)=4+6z^{-1}+2z^{-2}+z^{-3}$
$GF(11)$	$1+4z$	$1+7z$	$H_1(z)=4+z^{-1}+2z^{-2}+6z^{-3}$	$F_1(z)=6+2z^{-1}+z^{-2}+4z^{-3}$
			$H_0(z)=1+z^{-1}+4z^{-2}+7z^{-3}$	$F_0(z)=7+4z^{-1}+z^{-2}+z^{-3}$
$GF(13)$	$1+12z$	$3+9z$	$H_1(z)=7+7z^{-1}+z^{-2}+10z^{-3}$	$F_1(z)=10+z^{-1}+7z^{-2}+7z^{-3}$
			$H_0(z)=1+3z^{-1}+12z^{-2}+9z^{-3}$	$F_0(z)=9+12z^{-1}+3z^{-2}+z^{-3}$
$GF(17)$	$1+5z$	$1+12z$	$H_1(z)=9+z^{-1}+3z^{-2}+12z^{-3}$	$F_1(z)=12+3z^{-1}+z^{-2}+9z^{-3}$
			$H_0(z)=1+z^{-1}+5z^{-2}+12z^{-3}$	$F_0(z)=12+5z^{-1}+z^{-2}+z^{-3}$
$GF(19)$	$1+3z$	$1+16z$	$H_1(z)=12+12z^{-1}+z^{-2}+16z^{-3}$	$F_1(z)=16+z^{-1}+12z^{-2}+12z^{-3}$
			$H_0(z)=1+z^{-1}+3z^{-2}+16z^{-3}$	$F_0(z)=16+3z^{-1}+z^{-2}+z^{-3}$
$GF(199)$	$1+3z$	$58+72z$	$H_1(z)=16+16z^{-1}+z^{-2}+18z^{-3}$	$F_1(z)=18+z^{-1}+16z^{-2}+16z^{-3}$
			$H_0(z)=1+58z^{-1}+3z^{-2}+72z^{-3}$	$F_0(z)=72+3z^{-1}+58z^{-2}+z^{-3}$
			$H_1(z)=72+196z^{-1}+58z^{-2}+198z^{-3}$	$F_1(z)=198+58z^{-1}+196z^{-2}+72z^{-3}$

As an experimental image, the Lena standard image represented in the color space RGB is used. The value of each red, green, and blue color level for a separate pixel is equal to an integer from the range [0; 255]. To convert the information that the image contains into the RNS format, at first it is necessary to obtain modular components of the image. As one of the RNS modules, there were selected 199. This selection is explained by several reasons. The taking of too small a module (<100) would strongly complicate the visual perception of the transformed image, since, in the above RGB representation, all the pixels of the transformed image would be very close to the color black. The taking of a large module (<255) would result in the fact that the initial image and the image in the modular form would be identical.

Figure 9.20b shows the result of the translation of the image into the modular form of the field $GF(199)$, and the intermediate results of the operation of the image processing system by the trivial filterbank generated by the polynomials $A(z) = 1$ and $B(z) = 0$. It is shown in example 9.8 that for processing data in the field $GF(199)$ is possible to use the filters: $H_0(z) = 1 + 58z^{-1} + 3z^{-2} + 72z^{-3}$, $H_1(z) = 72 + 196z^{-1} + 58z^{-2} + 198z^{-3}$, $F_0(z) = 72 + 3z^{-1} + 58z^{-2} + z^{-3}$, and $F_1(z) = 198 + 58z^{-1} + 196z^{-2} + 72z^{-3}$. Figure 9.20c shows the result of the operation of the image processing system using these filters. It is clear that the images obtained at each processing stage, except for the input and output, are a "chaotic" set of pixels. Thus, the information processing using the finite-field wavelets opens up possibilities for creating cryptographic information protection systems.

For illustration of the cryptographic properties of the finite-field wavelets, let us consider the Lena image processing (Fig. 9.21a) in the RNS {7,11, 13, 17, 19} [23]. Filters from Table 9.6 for the corresponding fields were used in the filtering. Figure 9.21b shows the final result of the execution of all the image processing stages. Now let us assume that a violator gained access to the information obtained after the processing of the columns. Let us consider the worst case, i.e., the violator knows the RNS modules and the coefficients of the modulo-7, −11, −13, and −17 filters. The modulo-19 filter coefficients are unknown. Let the random orthogonal filterbank over field $GF(19)$ be used for analyzing the image. We selected the filters based on the polynomials $A'(z) = 1 + z$ and $B'(z) = 7 + 8z$, instead of true $A(z) = 1 + 3z$ and $B(z) = 1 + 16z$. Polynomials $A'(z)$ and $B'(z)$ also meet condition (9.23), and they are associated with the synthesizing filters $F'_0(z) = 8 + z^{-1} + 7z^{-2} + z^{-3}$ and $F'_1(z) = 18 + 7z^{-1} + 18z^{-2} + 8z^{-3}$ instead of true $F_0(z) = 16 + 3z^{-1} + z^{-2} + z^{-3}$ and $F_1(z) = 18 + z^{-1} + 16z^{-2} + 16z^{-3}$. Figure 9.21c shows the results of the image reconstruction with four true banks of synthesizing filters and with one untrue. It is clear that, even in this worst case, the resulting image does not bear any visual information about the initial image.

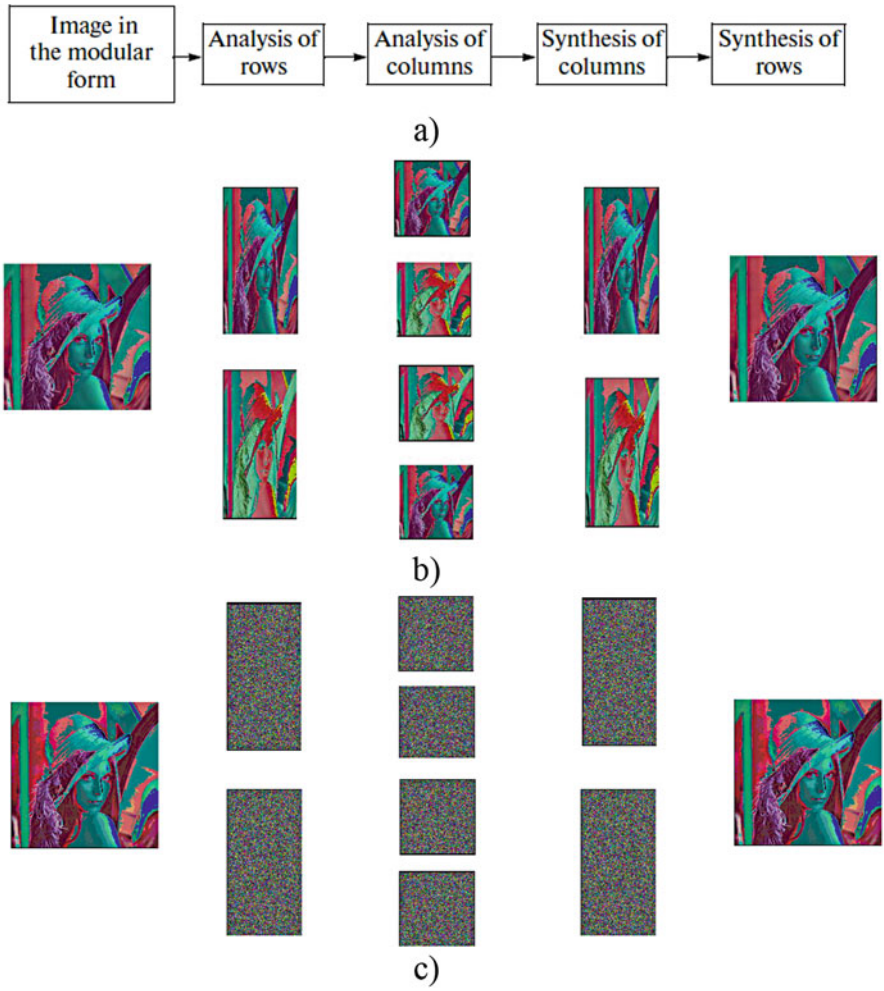

Fig. 9.20 Image processing over the field $GF(199)$: (**a**) general scheme of processing; (**b**) trivial filter; (**c**) orthogonal filter bank with $H_0(z) = 1 + 58z^{-1} + 3z^{-2} + 72z^{-3}$

This example allows one to conclude that the RNS can be used for implementing signal processing systems using finite-field wavelets. One of the important properties of the finite-field wavelets is the ability of encoding information, which is intensified by several applied processing channels in the RNS. In this case, the processing goes in parallel without increasing the encoding time.

The stated examples show that the proposed image processing approach in the RNS strengthens the cryptographic properties of finite-field wavelets by increasing the number of parallel and independent data processing channels. The use of finite-

Fig. 9.21 (**a**) Original Lena image; (**b**) reconstructed image after the processing in the RNS; (**c**) reconstructed image with a single error synthesizing filter bank

field wavelets in the RNS offers the challenge of designing new image compression and encoding algorithms and creating new noncryptographic and cryptographic information protection systems.

9.5 Conclusions

In this chapter principles of digital image processing using RNS were presented. Some applications of digital imaging were reviewed in detail: edge detection, sharpening, smoothing, and wavelet processing. There is shown how image processing filters may be implemented using RNS in order to improve performance and power consumption parameters. Techniques considered in this chapter may be used in machine vision, medicine imaging, satellite image processing, and other fields.

It should be noted that the image processing using RNS is relatively little studied direction. In the theory of digital image processing there are many other algorithms which could be adapted for implementation in RNS. Solution of this problem is interesting for further research in this area.

References

1. F.Y. Shih, *Image Processing and Pattern Recognition: Fundamentals and Techniques* (Wiley-IEEE Press, Hoboken, 2010)
2. R.C. Gonzalez, R.E. Woods, S.L. Eddins, *Digital Image Processing Using MATLAB*, 2nd edn., (Gatesmark Publishing, 2009), www.gatesmark.com
3. M. Honda, M. Kameyama, T. Higuchi, Residue arithmetic based multiple-valued VLSI image processor, in *Proceedings of Twenty-Second International Symposium on Multiple-Valued Logic*, Sendai, Japan, 1992, pp. 330–336

4. A.C. Bovik, *Handbook of Image and Video Processing*, 2nd edn. (Academic Press, Boston, 2005)
5. A. Ammar, A.S.S. El-Kabbany, M.I. Youssef, A. Emam, A novel data hiding technique using residue number system, in *Proceedings of the Twentieth National Radio Science Conference*, Cairo, Egypt, 2003, pp. 1–12
6. Atta-ur-Rahman, M.T. Naseem, I.M. Qureshi, M.Z. Muzaffar, Reversible watermarking using Residue Number System, in *Proceedings of 7th International Conference on Information Assurance and Security*, Malacca, Malaysia, 2011, pp. 162–166
7. S.K. Singh, V.P. Gopi, P. Palanisamy, Image security using DES and RNS with reversible watermarking, in *Proceedings of 2014 International Conference on Electronics and Communication Systems*, Coimbatore, India, 2014, pp. 1–5
8. M. Gomathisankaran, X. Yuan, P. Kamongi, Ensure privacy and security in the process of medical image analysis, in *Proceedings of 2013 IEEE International Conference on Granular Computing*, Denton, TX, USA, 2013, pp. 120–125
9. D.K. Taleshmekaeil, A. Mousavi, The use of residue number system for improving the digital image processing, in *Proceedings of 10th International Conference on Signal Processing*, Tabriz, Iran, 2010, pp. 775–780
10. D. Younes, P. Steffan, Efficient image processing application using Residue Number System, in *Proceedings of the 20th International Conference on Mixed Design of Integrated Circuits and Systems*, Brno, Czech Republic, 2013, pp. 468–472
11. Q. Huynh-Thu, M. Ghanbari, Scope of validity of PSNR in image/video quality assessment. Electron. Lett. **44**(13), 800–801 (2008)
12. Z. Wang, Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**(4), 600–612 (2004)
13. E. Vassalos, D. Bakalis, H.T. Vergos, RNS assisted image filtering and edge detection, in *Proceedings of 18th International Conference on Digital Signal Processing*, Patras, Greece, 2013, pp. 1–6
14. A. Tatsaki, T. Stouraitis, C. Goutis, Image coder based on residue number system for progressive transmission. Electron. Lett. **31**(6), 442–443 (1995)
15. M. Vetterli, J. Kovacevic, *Wavelets and Subband Coding* (CreateSpace Independent Publishing Platform, San Francisco, CA, 2013)
16. J. Ramirez, U. Meyer-Base, F. Taylor, A. Garcia, A. Lloris, Design and implementation of high-performance RNS wavelet processors using custom IC technologies. J. VLSI Sig. Proc. **34**, 227–237 (2003)
17. J. Ramirez, A. Garcia, U. Meyer-Base, F. Taylor, A. Lloris, Implementation of RNS-based distributed arithmetic discrete wavelet transform architectures using field-programmable logic. J. VLSI Sig. Proc. **33**, 171–190 (2003)
18. F. Fekri, R.M. Mersereau, R.W. Schafer, Theory of wavelet transform over finite fields, in *Proceedings of 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Atlanta, GA, USA, 1999, pp. 1213–1216
19. F. Delgosha, F. Fekri, Stream cipher using finite-field wavelets, in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, Atlanta, GA, USA, 2005, pp. 689–692
20. F. Fekri, S.W. McLaughlin, R.M. Mersereau, R.W. Schafer, Block error correcting codes using finite-field wavelet transforms. IEEE Trans. Signal Process. **54**(3), 991–1004 (2006)
21. P.P. Vaidyanathan, *Multirate Systems and Filter Banks* (Prentice Hall, Englewood Cliffs, NJ, 1992)
22. P. See-May, P.P. Vaidyanathan, Paraunitary filter banks over finite fields. IEEE Trans. Signal Process. **45**(6), 1443–1457 (1997)
23. N.I. Chervyakov, P.A. Lyakhov, M.G. Babenko, Digital filtering of images in a residue number system using finite-field wavelets. Autom. Control. Comput. Sci. **48**(3), 180–189 (2014)

Chapter 10
Logarithmic Number System and Its Application in FIR Filter Design

Vassilis Paliouras

10.1 Introduction

Data representation is an important parameter in the design of digital signal processors as it affects their performance and hardware complexity. Data representation influences the choice of algorithms and the hardware architecture, thus it has an impact on data storage requirements, processing performance, complexity, and power dissipation [1–3].

The Logarithmic Number System (LNS) can be used to efficiently represent data in special-purpose processors. The efficiency of LNS stems from the properties of the logarithm exploited to reduce the basic arithmetic operations of multiplication, division, square roots, and squares to simple binary addition, subtraction, and right and left shifts, respectively [4]. Similarly, the computation of general roots and powers as well as of reciprocals are substantially simplified.

Building on the reduced complexity of certain arithmetic operations, digital signal processor implementations can benefit from substantial architectural-level simplifications. Furthermore, beyond operation strength reduction, superior LNS dynamic range and numerical behavior, compared to conventional representations, are additional reasons that make LNS attractive. In order to fully utilize LNS potential, procedures are required to optimally define LNS design parameters and derive LNS solutions, matched to application specifications and requirements.

The remainder of the chapter is organized as follows: A brief review is offered in Sect. 10.2 and LNS design procedures are discussed in Sect. 10.3. LNS basics are discussed in Sect. 10.4. Section 10.5 focuses on base optimization, while Sect. 10.6 presents techniques for the low-power implementation of LNS addition/subtraction.

V. Paliouras (✉)
University of Patras, Patras, Greece
e-mail: paliuras@ece.upatras.gr

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_10

In Sect. 10.7 various filter structures are discussed, while a hybrid linear-LNS scheme is reviewed in Sect. 10.8. Section 10.9 demonstrates the impact of the proposed design methodology to other LNS schemes. Finally, conclusions are discussed in Sect. 10.10.

10.2 Literature Review

LNS roundoff error behavior resembles that of floating-point arithmetic. In fact, LNS-based systems have been proposed that exhibit characteristics similar to 32-bit single-precision floating-point representation [5, 6]. Specifically, Coleman et al. have reported that the European Logarithmic Processor (ELM) compares favorably to 32 bit floating-point implementations [6]. LNS provides an additional degree of freedom in the exploration of VLSI hardware design space, namely the choice of logarithmic base: The choice of the LNS base allows the designer to fine-tune resolution *vs.* dynamic range trade-offs.

As expected, LNS benefits come at a cost: the operations of addition and subtraction are rather awkward to perform in LNS as large look-up tables or other approximation circuitry are required. While for short word lengths simple techniques based on LUTs suffice, involved approximation techniques become necessary for longer word lengths. Several authors have proposed solutions to reduce complexity of awkward LNS operations. Mahalingam and Ranganathan improve Mitchell's Algorithm (MA) in terms of the accuracy of the logarithmic operations [7], while Johansson et al. use a method based on sums of bit-products to implement the basic logarithmic functions [8]. Arnold et al. suggest the use of co-transformations for the reduction of the look-up tables [9]. Dimitrov et al. have proposed an extension of LNS in which several bases are used [10]. In this context and to address the complicated required conversion to LNS, Muscedere et al. have studied techniques for the conversion of binary to a multidigit-multidimensional logarithmic number system by using look-up tables [11]. Ismail and Coleman have presented a co-transformation procedure and an improved interpolation method that reduce the size of look-up tables facilitating their easy synthesis in logic [12]. Fu et al. deal with LNS arithmetic optimizations on FPGAs [13]. Arnold and Collange propose Complex LNS (CLNS) as a generalization of LNS, which represents complex values in log-polar form [14].

For several practical applications, the benefits of LNS are found to be more important than its inherent disadvantages. In particular, several authors have shown that LNS reduces power dissipation in a variety of signal processing-related applications, ranging from hearing-aid devices [15] and subband coding [16] to video processing [17] and error control [18]. Moreover, logarithmic techniques have been employed in turbo code decoding for wireless communications applications. In particular, logarithmic representation has been used in the implementation of the symbol-by-symbol Log-MAP (Logarithmic Maximum A Posteriori) algorithm for iterative decoders [19, 20]. Peng and Chen have adopted LNS for the implementation of an FFT-based Log-Sum-Product-decoding-Algorithm (Log-SPA) used in decoding of nonbinary Low-Density Parity Check (LDPC) codes [21].

The properties of logarithmic representation that render it efficient for reducing power dissipation have been studied [22–25] and it has been demonstrated that the proper choice of the parameters of the representation can reduce the switching activity, while guaranteeing the quality of the output signal. In this context, the quality of the output is evaluated in terms of signal-to-noise ratio (SNR). In particular, the impact of the selection of the base b of the logarithm has been investigated as a means to explore trade-offs between precision and dynamic range given a particular word length. Paliouras and Stouraitis [22, 23] address the low-power LNS properties from a representational viewpoint and do not focus on power dissipation estimation data obtained by circuit simulations. The low-power characteristics of LNS addition/subtraction and multiplication have been quantitatively studied and compared to equivalent linear binary two's-complement fixed-point operations [24], where it has been demonstrated that there are practical cases, in which an optimized LNS representation can replace a linear representation of longer word length, without signal quality degradation.

10.3 A Design Methodology for LNS Systems

The adoption of LNS in a digital signal processor offers a multitude of design parameters that can be controlled in order to fine-tune behavior, performance, and complexity to optimally meet design constraints and application requirements. Due to the multitude of design parameters, a systematic exploration of the design space is required, as outlined in Fig. 10.1, extending [26]. Algorithm selection taking into account operation strength reduction due to LNS is an important step, as most solutions are biased towards the use of algorithms based on operations simpler to implement using linear expressions; e.g., the minimization of the number of multiplications is commonly sought, contrasted to the number of additions.

Extending [25] and [24], optimal selection of LNS design parameters refers to the determination of the word length and the logarithmic base, which influence the

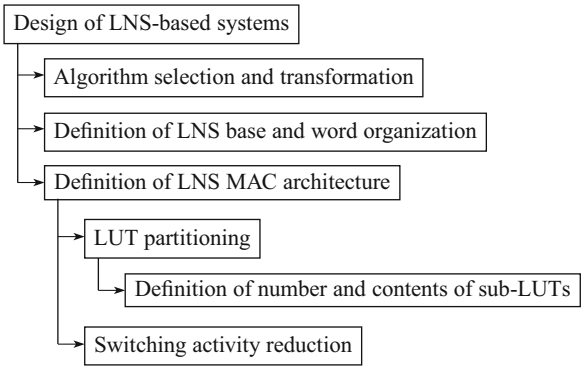

Fig. 10.1 A design methodology for LNS-based systems

organization and complexity of employed look-up tables. The particular framework has been illustrated in [26] for the case of certain FIR filter structures; it has been used to explore the area–time–power design space of a low-pass FIR filter for several configurations of MAC units. Partitioning has been highlighted as a technique to limit the exponential growth of the size of LUTs with increasing word length. Partitioning optimization defines the number, size, and contents of small look-up tables that constitute the memory which stores the addition and subtraction function. Several LNS architectures for addition, subtraction, and multi-operand operations have been proposed in the literature using interpolation, linear, or polynomial approximation aiming at reducing the memory requirements, particularly for larger word lengths [27, 28]. Furthermore mathematical decompositions and transformations of the basic operations, exploiting the particular characteristics of the functions, have been used to further simplify approximation [12, 27–31].

LNS arithmetic units that implement more awkward operations such as addition and subtraction have structural characteristics that can be exploited to reduce power dissipated, namely mutually exclusive sub-units, which can be used selectively, and imbalanced delay paths. The particular structural properties can be exploited by simple low-power design techniques both based on table look-up or more sophisticated techniques. Extension to other architectures is demonstrated using an interpolation-based subtractor as an illustrative example. Partitioned LUT circuits provide high speed; more sophisticated techniques can be used to reduce size [12, 13, 28–30, 32, 33].

10.4 Basics of LNS Representation

The basic idea in LNS is to use logarithms to represent data. Since the logarithm of a negative number is not real, LNS signed-number representation stores the sign information as a separate bit s_X, to be used in combination with the logarithm of the magnitude of the number. Furthermore, due to the singularity of logarithm at point zero, an additional single-bit flag z_X is used to denote that a number is zero. Formally in LNS, a number X is represented as the triplet

$$\mathscr{X}_{\mathrm{LNS}} = (z_x, s_x, x), \tag{10.1}$$

where z_x is asserted when X is zero, s_{x} is the sign of X, and $x = \log_b(|X|)$, if X is not zero, with b being the base of the logarithm, also called base of the representation. The choice of b plays a crucial role in the representational capabilities of the triplet in (10.1), as well as the computational complexity of the processing and forward and inverse conversion circuitry.

Due to the basic properties of the logarithm, the multiplication of $\mathscr{X}_{\mathrm{LNS}}$ and $\mathscr{Y}_{\mathrm{LNS}}$ is reduced to the computation of the triplet $\mathscr{Z}_{\mathrm{LNS}}$,

$$\mathscr{Z}_{\mathrm{LNS}} = (z_z, s_z, z), \tag{10.2}$$

where $z_Z = z_X \bigvee z_Y$, $s_Z = s_X \bigoplus s_Y$, and $z = x + y$. Similarly division reduces to binary subtraction, while the computation of roots, powers, and reciprocals is substantially simplified.

The derivation of the logarithm a of the sum $\mathscr{A}$ of two LNS triplets is more involved, as it relies on the computation of

$$a = \max\{x, y\} + \log_b \left(1 + b^{-|x-y|}\right) \tag{10.3}$$

$$= \max\{x, y\} + \phi_a(d), \tag{10.4}$$

where $\phi_a(d) = \log_b(1 + b^{-d})$ and

$$d = |x - y|. \tag{10.5}$$

Similarly the derivation of the logarithm c of the difference of two LNS numbers requires the computation of

$$c = \max\{x, y\} + \log_b \left(1 - b^{-|x-y|}\right) \tag{10.6}$$

$$= \max\{x, y\} + \phi_s(d), \tag{10.7}$$

where $\phi_s(d) = \log_b(1 - b^{-d})$.

The logarithm in an LNS triplet is expressed as a binary word. Assume that a two's-complement (TC) word is used to represent the logarithm x, composed of a k-bit integral part and an l-bit fractional part. The range D_{LNS} of x is given by

$$D_{\text{LNS}} = \left[-b^{2^{k-1}-2^{-l}}, -b^{2^{-l}}\right] \bigcup \{0\} \bigcup \left[b^{2^{-l}}, b^{2^{k-1}-2^{-l}}\right]. \tag{10.8}$$

Range D_{LNS} is compared with range $D_{\text{TC}} = \left[-2^{i-1},\ 2^{i-1} - 2^{-f}\right]$ of a linear TC representation of i integral bits and f fractional bits. In general, LNS offers a superior range, over the linear TC representation of comparable word lengths. The basic organization of an LNS adder/subtractor is shown in Fig. 10.3. Two parallel subtractions

$$s_1 = x - y \tag{10.9}$$

$$s_2 = y - x \tag{10.10}$$

are followed by a multiplexer, which selects d according to

$$d = |x - y| = \begin{cases} s_1, & s_1 > 0 \\ s_2, & \text{otherwise.} \end{cases} \tag{10.11}$$

The sign of either (10.9) or (10.10) serves as a select signal for the multiplexer. The same signal is used to select the maximum of x and y, required for the computation of (10.4) and (10.7).

There are two main approaches to implement the evaluation of functions, namely the hardware implementation of an approximation algorithm or the off-line precomputation and storage of all required values in a look-up table [34]. The former approach is generally adopted for high-precision applications, while the latter approach is generally preferable for smaller word lengths, i.e., in relatively low-precision applications where the size of the required look-up tables is moderate. Both approaches have been extensively studied in the context of elementary function approximation. Let x denote the base-b logarithm of X and x_2 denote the base-2 logarithm of X. Since $x = \log_b |X| = \log_b 2^{x_2} = x_2 (\log_b 2)$, the conversion between a base-b LNS and a base-2 LNS requires scaling by a constant factor. Several authors have studied hardware implementations of converters to/from base-2 LNS [11, 35, 36].

10.5 Impact of the LNS Base

The base of the LNS representation allows for an additional degree of freedom in the exploitation of range *vs.* precision trade-offs, which is not available in linear representations. This section reviews optimization techniques for the determination of the base of LNS.

10.5.1 Base Optimization

The choice $b = 2$ is common and it allows LNS to be considered as an extreme case of a base-2 floating-point representation [37]. However several works depart from the particular choice [22, 23, 38–40] and in this way achieve various improvements. Among them, Kwa et al. [40] use bases b, $b < 1$ to study the usable LNS dynamic range requirements for speech and sinusoidal signals. Furthermore, by choosing particular values of $b < 1$, significant memory reductions are also achieved.

The issue of equivalence between LNS and linear arithmetic under several assumptions is discussed in [22]. Specifically, a closed-form criterion for representational equivalence has been proposed, which guarantees equivalent accuracy and dynamic range between an LNS of base b, a k-bit integral part and an l-bit fractional part, and a linear fixed-point system of n bits [22].

In [38] the model of [22] is used to derive LNS bases and reduce memory requirements. The dynamic range and the average representational accuracy offered by an n-bit linear system are matched by a base-b LNS, when k and l are computed—

as described by Theorem 1 of [22]—via the formulas

$$l(n,b) = \left\lceil -\log_2 \log_b \left(1 + \frac{\psi(2^n) + \gamma}{2^n - 1}\right)\right\rceil \tag{10.12}$$

$$k(n,b) = \left\lceil \log_2 \log_b \left((2^n - 1) + 2^{-l(n,b)}\right)\right\rceil, \tag{10.13}$$

where $b > 1$, $\psi(x) = \frac{d}{dx} \ln \Gamma(x)$, and γ is the Euler constant. To numerically evaluate (10.12) and (10.13), it is noted that $\psi(2^n)\psi(2^n) = \sum_{i=1}^{2^n-1} \frac{1}{i} - \gamma$ and $\gamma \approx 0.577216$.

For word lengths similar to the IEEE 754 floating-point single-precision format, numerical approximations of the functions $f_a(d)$ and $f_s(d)$ have been proposed [41]. However, for shorter word lengths, e.g., less than 16 bits, functions $f_a(d)$ and $f_s(d)$ can be practically evaluated using look-up tables. The argument d, $d > 0$, is upper bounded by the effective-zero values [42] e_a and e_s that correspond to $f_a(d)$ and $f_s(d)$, respectively. Specifically, for values of d, $d \geq e_a$, f_a is identified to zero, since its value is less than the minimum quantity representable with l fractional bits. Two cases are distinguished depending on the value of the base, namely $b > 1$ and $b < 1$.

The number $w_{\text{add}}(b)$ of memory words that need to be stored in the LNS addition LUT can be computed as [22]:

$$w_{\text{add}}(b) = \left\lceil \frac{e_a - 2^{-l(n,b)}}{2^{-l(n,b)}} \right\rceil = \left\lceil -2^{l(n,b)} \log_b (b^{2^{-l(n,b)}} - 1) - 1 \right\rceil, \tag{10.14}$$

where $l(n,b)$ depends on the value of b, as implied by (10.12), $e_a = -\log_b(2^{-l} - 1)$, and $e_s = \log_b(1 - b^{-2^{-l}})$. To correctly compute the number of words $w_{\text{add}}(b)$ using (10.14), it should hold that $e_a(b,l) > 2^{-l}$. The definition of $w_{\text{sub}}(b)$ for the LNS subtraction function $f_s(d)$ is similar, leading to

$$w_{\text{sub}}(b) = \left\lceil -2^{l(n,b)} \log_b \left(1 - b^{-2^{-l(n,b)}}\right) - 1 \right\rceil. \tag{10.15}$$

Following a minimization procedure [38], bases of the form

$$b_i = \left(1 + \frac{\gamma + \psi(2^n)}{2^n - 1}\right)^{2^i} \tag{10.16}$$

for $i \geq 0$, are found to minimize the number of stored words. Eq. (10.16) introduces a closed formula for optimal bases $b_i > 1$. A similar analysis for the case $b < 1$ derives to

$$b_i = \left(1 - \frac{\psi(2^n) + \gamma}{2^n - 1}\right)^{2^i}. \tag{10.17}$$

for word lengths $n > 2$.

Table 10.1 Number of memory words $w_{\text{add}}(2)$ and $w_{\text{sub}}(2)$ for $b = 2$, and $w_{\text{add}}(b)$, $w_{\text{add}}(b)$, achieved for equivalent n-bit linear systems and the values $b = b_{\text{opt}}$

	LNS subtraction		LNS addition		
n	$w_{\text{sub}}(2)$	$w_{\text{sub}}(b_{\text{opt}})$	$w_{\text{add}}(2)$	$w_{\text{add}}(b_{\text{opt}})$	b_{opt}
8	177	153	176	152	0.9526
9	418	321	418	320	0.9735
10	964	667	963	666	0.9854
11	2183	1374	2182	1373	0.9920
12	4879	2819	4879	2818	0.9957
13	10,781	5763	10,780	5762	0.9977
14	23,611	11,746	23,610	11,747	0.9987

Table 10.1 reports the number of stored words required for $f_s(d)$ and $f_a(d)$, for several values of n. Base optimization techniques do not impose any overhead in an LNS-based architecture; however, they can substantially reduce the words that need to be stored in an LUT, for LNS addition and subtraction, depending on the word length required by the application.

10.5.2 Optimization of Representation for FIR Filters

Some works utilize the average representational error as a measure to define equivalence between LNS and linear representation [22, 38]. For the case of FIR filters, the output SNR emerges as an appropriate equivalence criterion [23, 24].

The output SNR of an LNS FIR filter has been both theoretically and experimentally studied in conjunction with the LNS word organization in [39] and [36]. Chandra provides an expression for the ratio of output error variance to output signal variance of a logarithmic FIR filter implementation due to roundoff [39]. Chandra assumes the use of the logarithmic base $b = 2$.

In [26], the approach has been extended to general bases; i.e., other than two. The extension to more general bases is required as in this way the base can be treated as a design parameter, the value of which should be optimally determined taking into account both hardware implementation issues the SNR as a measure of output quality. Error ϵ due to roundoff in the logarithmic representation and assuming rounding of addition outputs is given by [39]:

$$\epsilon = K_x - L_x, \tag{10.18}$$

where $L_x = \log_b |x|$ and K_x is the quantized value of L_x, rounded to l fractional bits. Assuming a base b, the relative error in a linear expression is

$$e = \frac{b^{K_x} - b^{L_x}}{b^{L_x}} = b^{\epsilon} - 1, \tag{10.19}$$

hence,

$$\epsilon = \log_b(1 + e). \tag{10.20}$$

Since ϵ is due to rounding, it can be assumed as a random variable, uniformly distribution in the interval $[-\tau, \tau]$, where $\tau = 2^{-l-1}$; i.e., the probability density function of e is

$$f_\epsilon(\epsilon) = \begin{cases} \frac{1}{2\tau}, \epsilon \in [-\tau, \tau] \\ 0, \quad \text{else} \end{cases}. \tag{10.21}$$

Using (10.20) and (10.21) and applying variable transformation [43], the probability density function (pdf) of e is obtained as follows:

$$f_e(e) = \frac{1}{2\tau \log b} \frac{1}{1 + e}. \tag{10.22}$$

The expected value $m_e = E[e]$ is obtained from (10.22), as

$$m_e = E[e] = \frac{b^\tau - b^{-\tau}}{2\tau \log b} - 1. \tag{10.23}$$

The MSE of the error e in the linear domain is given by

$$E[e^2] = \frac{(b^\tau - b^{-\tau})(b^\tau + b^{-\tau} - 4)}{4\tau \log b} + 1. \tag{10.24}$$

Finally, the linear roundoff-noise variance σ_e^2 is given by

$$\sigma_e^2 = E[e^2] - m_e^2. \tag{10.25}$$

Resembling [39] for the case of successive additions, the roundoff-noise variance to signal variance ratio, $\frac{\sigma_f^2}{\sigma_y^2}$, at the output of an N-th order filter is given by

$$\frac{\sigma_f^2}{\sigma_y^2} = \frac{\sigma_e^2 \sum_{k=0}^{N-1} P_{kk} h_k^2}{\sum_{k=0}^{N-1} h_k^2}, \tag{10.26}$$

where σ_e^2 is given by (10.25), h_k are the coefficients of the filter, and $P_{00} = N - 1$, $P_{kk} = N - k$, for $k > 0$. Equation (10.26) relates SNR with the filter coefficients, the base b, and the fractional word length l, since $\tau = 2^{-l-1}$. However, it does not take into account coefficient quantization and overflow/underflow errors. Therefore no limit on the number of integral bits is considered. Figure 10.2 depicts the behavior of SNR, computed as

$$\text{SNR} = -10 \log_{10} \frac{\sigma_f^2}{\sigma_y^2} \tag{10.27}$$

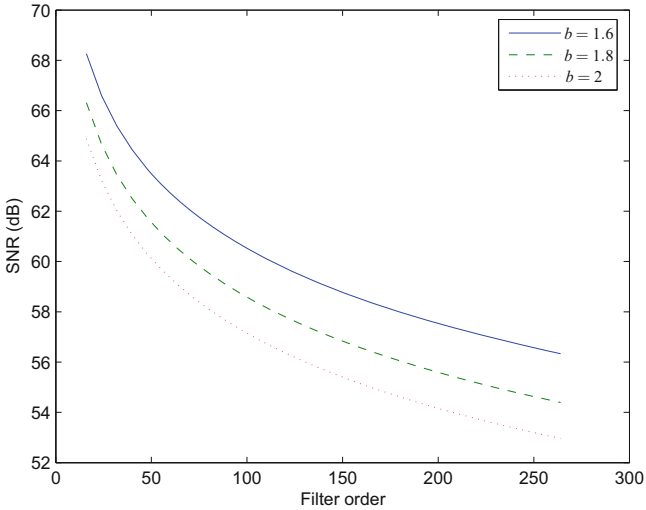

Fig. 10.2 SNR as a function of filter order for various values of b and $l = 10$

as a function of the filter order N, for three different values of b, and a fractional word length of $l = 10$.

The minimum number of the integral bits needed to achieve the theoretical SNR for a given number of fractional bits is discussed in [36]. In particular, Kurokawa et al. report that when the number of integral bits is too small, theoretical and experimental results disagree due to overflow or underflow [36]. They also report that the number of integral and fractional bits can be changed without having an effect on output SNR, provided that the total word length remains unchanged and an appropriate base is used. Specifically, base b^2 is used if the fractional bits increase by one; $\sqrt{b}$ if the fractional bits decrease by 1.

In this context, Basetas et al. [24] have demonstrated that a base b of value different than the common choice of two (i.e., $b \neq 2$) may provide better balance between accuracy and dynamic range, thus offering a higher output SNR. The experimentally determined SNR and corresponding LNS word organization are found to be in close agreement with anticipated results. Initially LNS word organizations that provide at least better SNR than fixed-point counterparts are required. The organization of the LNS word is determined by deciding on the total word length and the number k, of integral, and l, of fractional bits. A basic simulation-based methodology comprises two steps:

1. A linear FIR filter model is simulated utilizing an n-bit fixed-point two's-complement (TC) representation and the corresponding output SNR is calculated. The output of a double-precision floating-point FIR model is used as the reference signal.

2. An LNS FIR filter model is implemented and its behavior is studied for several LNS word organizations, i.e., determine the numbers k integer and l fractional bits required and the corresponding base.

In order to guarantee an unbiased comparison between fixed-point and LNS FIR filters, the employed models use identical data types with the hardware and perform on them bit-true operations, so that numerical results obtained by the hardware and the model identify. The procedure is used to determine equivalent data representations for both fixed-point and the LNS-based systems. Subsequently, circuits based on the derived representations are simulated in order to perform quantitative comparisons in terms of area, power, and delay.

10.6 Design of LNS Circuits

In this section design of LNS hardware for addition and subtraction is briefly reviewed. Targeting to relatively low word lengths, techniques based on look-up tables are found to be simple and efficient. Optimized table partitioning can be employed to achieve performance improvement and low-power operation. The look-up table is implemented as a collection of LUTs and is the most complex part of the LNS adder/subtractor.

Complexity and power dissipation reduction is sought by partitioning the LUTs into smaller LUTs, called sub-LUTs, as shown in Fig. 10.3. It is noted that only one of the sub-LUTs is accessed and, therefore, active per operation. To guarantee that no dynamic power is dissipated in the inactive sub-LUTs, the sub-LUT addresses are latched and remain constant throughout a particular operation; i.e., only the address of the active sub-LUT is updated. Several design choices are possible, such as the choice of using either latches or D flip-flops (DFFs) to freeze the addresses of inactive sub-LUTs, or the choice to select the active sub-LUT either based on the most significant bits (MSB) or on the least significant bits (LSB) of d in (10.11). Further power dissipation reduction and throughput rate increase at the multiply accumulate units are achieved by using retiming, as well as at the algorithmic optimization level by optimally selecting the parameters of the LNS representation.

Complexity reduction in LNS processors by partitioning of the LUTs has been successfully applied in the past [44]. The organization of the LNS adder/subtractor comprises N sub-LUTs per operation, as shown in Fig. 10.3. The upper sub-LUT system corresponds to function $\phi_a(d)$ required for LNS addition, i.e., addition of operands having the same sign, while the lower sub-LUT system is used for LNS subtraction, i.e., addition of operands of different signs.

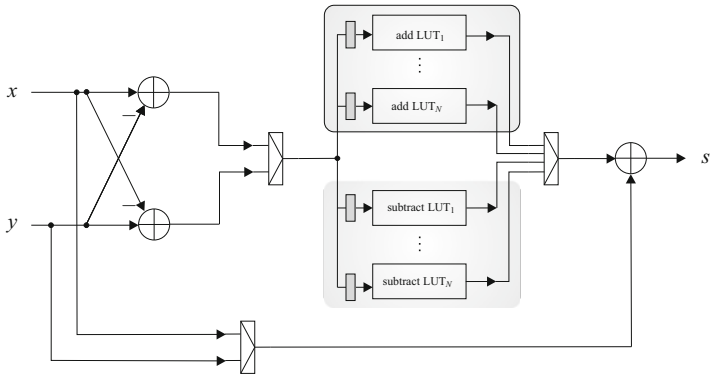

Fig. 10.3 Organization of an LNS adder/subtractor

10.6.1 Organization of Look-up Table Subsystem

The range of values of the functions $\phi_a(d)$ and $\phi_s(d)$ required to be stored is defined by the *essential zero*, X_{eff}, beyond which the absolute value of $\phi_a(d)$ or $\phi_s(d)$ is lower than the resolution of the representation. As $\phi_a(x)$ monotonically decreases, and by requiring that $\phi_a(X_{\text{eff}}) \leq 2^{-l}$, X_{eff} is obtained as

$$X_{\text{eff}} = -\log_b \left(b^{2^{-l}} - 1 \right). \tag{10.28}$$

Therefore the total number of words, i.e., values of $\phi_a(x)$, required to be stored is

$$W(b, l) = \left\lceil \frac{X_{\text{eff}}}{2^{-l}} \right\rceil. \tag{10.29}$$

The number of bits required for addressing the memory subsystem is given by

$$\text{ab}(b, l) = \lfloor \log_2 W(b, l) \rfloor + 1. \tag{10.30}$$

Assume that the m most significant bits of the memory subsystem address are used for selecting the sub-LUT to be activated for a particular addition. Then the number of words stored in each sub-LUT is given by

$$w(b, l, m) = 2^{\text{ab}(b,l)-m}. \tag{10.31}$$

Since $W(b, l)$ is not necessarily an integral multiple of w, the number of words w_l stored in the last sub-LUT is given by

$$w_l(b, l, m) = W(b, l) - (N(b, l, m) - 1)w(b, l, m), \tag{10.32}$$

where $N(b, l, m)$ is the number of sub-LUTs. It is interesting to note that using the m most significant bits of the address to select the sub-LUT to activate does not imply that the number of sub-LUTs is 2^m. Due to $\phi_a(d)$ being essentially zero for $d > X_{\text{eff}}$, the number N of required sub-LUTs is given by

$$N(b, l, m) = \left\lfloor \frac{X_{\text{eff}}}{2^{\text{ab}(b,l)-m} 2^{-l}} \right\rfloor + 1. \tag{10.33}$$

The first value of $\phi_a(k)$ stored in the kth sub-LUT assuming the selection of sub-LUTs based on MSB of d corresponds to the input values $x(k)$

$$x(k) = k 2^{\text{ab}(b,l)-m} 2^{-l}. \tag{10.34}$$

Therefore the word length of values stored in the kth sub-LUT is

$$\text{bpwLUT}(k) = l + \lfloor \log_2 \phi_a(x(k)) \rfloor + 1. \tag{10.35}$$

It is noted that in the case of using LSB of d in (10.11) to select sub-LUT, since the monotonicity of the stored values cannot be exploited, the word length of values is essentially constant; i.e.,

$$\text{bpwLUT}_{\text{LSB}}(k) = \text{bpwLUT}(1). \tag{10.36}$$

As an illustrative example consider Fig. 10.4. The kth sub-LUT, $1 \leq k \leq 2^m$, stores values $\phi_a(d)$, with $(k-1)2^{n-m} \leq d \cdot 2^l < k \cdot 2^{n-m}$. Bits $d_{n-1} \ldots d_{n-1-m}$ are used to select the requested value between the outputs of the p sub-LUTs. Since $\phi_a(d)$ is monotonically decreasing, it holds that

$$\phi_a(d) \leq \phi_a(2^{n-1-l}), \tag{10.37}$$

for $d \geq 2^{n-1-l}$. Therefore differences among the values stored in LUT2 are limited to their less significant part, therefore the less significant part is the only one that needs to be stored for each value. Hence fewer bits per entry are required to be stored in LUT2 than in LUT1. Sub-LUTs that correspond to the upper parts of the interval need to store data words of reduced length, since stored values share a common most significant part as shown in Fig. 10.4a.

The possibility to determine the active sub-LUT using the LSBs of d is of interest, as LSBs are available early in the computation of d, thus allowing the fast generation of selection signals. However a partitioning scheme based on LSBs does not facilitate memory compression since consecutive function samples are stored in different sub-LUTs, as shown in Fig. 10.4b. Therefore no common most significant part of the sub-LUT contents can be omitted, resulting in increased storage with respect to MSB-based sub-LUT selection. Total storage requirements, expressed in number of stored bits, are given by

$$B_{\text{tot}} = \sum_{k=0}^{N-2} \text{bpwLUT}(k) w(b, l, m) + \text{bpwLUT}(N-1) w_l(b, l, m). \tag{10.38}$$

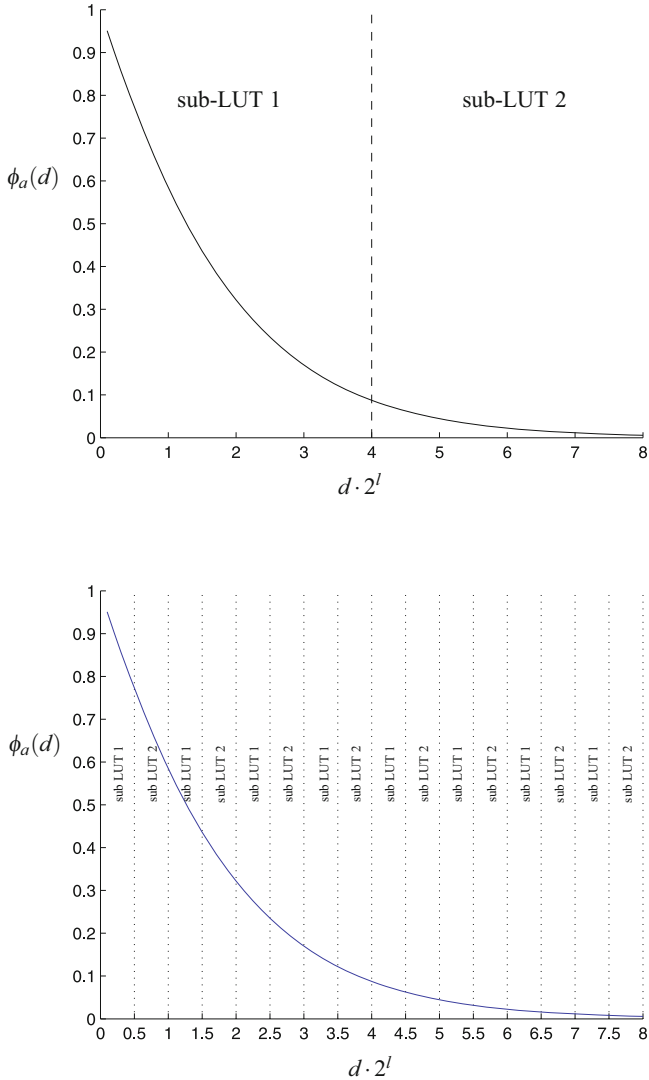

Fig. 10.4 Partitioning of the addition function $\phi_a(d)$ storage using two different sub-LUT schemes. (**a**) Active sub-LUT is selected by the MSB of d. (**b**) Active sub-LUT is selected by the LSB of d

The complexity imposed by latching the sub-LUT addresses depends on the number of sub-LUTs N, as follows:

$$A_{\text{in}} = N(\text{ab}(b, l) - m). \tag{10.39}$$

Finally, to select the output of N sub-LUTs, a network of $(N-1)$ multiplexers are required. An upper bound on the complexity of this network is

$$A_{\text{mux}} = (N-1)(l + \lfloor \log_2 \phi_a(0) \rfloor + 1). \tag{10.40}$$

Further complexity reduction is possible for the MSB-based selection by exploiting the fact that the actual word length of the data in the kth sub-LUTs is given by (10.35), which is up-bounded by $l + \lfloor \log_2 \phi_a(0) \rfloor + 1$, due to the monotonicity of $\phi_a(x)$. A similar analysis can be carried out for the case of $\phi_s(x)$.

The above analysis reveals that the partitioning of the storage into sub-LUTs with latched inputs introduces an area cost for latching the sub-LUT addresses and multiplexing the sub-LUT outputs. The particular complexity increases linearly with the number N of sub-LUTs. Assuming that the multiplexer network is organized as a tree, a delay cost that increases logarithmically with N is also imposed. Therefore, a detailed design-space exploration is required to determine the proper number of sub-LUTs required to achieve given design specifications. To facilitate such an exploration, an objective function

$$A(b, t, m) = B_{\text{tot}} a_{\text{mem}} + A_{\text{in}} a_{\text{lat}} + A_{\text{mux}} a_{\text{mux}} \tag{10.41}$$

is formed, where a_{mem}, a_{lat}, and a_{mux} denote the area cost of storing a bit in a look-up table, of a latch, and an one-bit two-to-one multiplexer, respectively. Such a model provides a starting point for the determination of m, i.e., how many of the address bits are used for sub-LUT selection, given the base and the word organization. While a_{lat} and a_{mux} can be found in the corresponding standard-cell library databook, a_{mem} depends on the results and effectiveness of the employed logic synthesis algorithms.

10.6.2 Sub-LUT Selection Strategies

Figure 10.5 depicts an architecture using a one-bit LSB selection for LUT partitioning. The inputs to the sub-LUTs are latched to prevent unnecessary switching. Sign s and LSB d_0 form a two-bit word which is subsequently decoded to generate enable signals for the latches at sub-LUT inputs. A single sub-LUT is activated for a particular computation. LSB d_0 should reach the latches fast enough, considering the additional delay of computing $|x - y|$, to avoid timing violations.

Figure 10.6 depicts a DFF-based architecture operating with a latch-based gated clock. Gated clocks achieve further power savings and also the problem of setup and hold time violations is easier to resolve, since glitches are avoided [45]. A quantitative comparison of the two techniques is reported in [26], where the LUTs have been implemented as combinational logic, synthesized in a UMC 90 nm 1.0 V CMOS standard-cell library, by using the Synopsys Design Compiler, IC Compiler, and Prime Time (http://www.synopsys.com). Power dissipation results assume the maximum possible clock frequency as dictated by circuitry delay, in case of DFFs

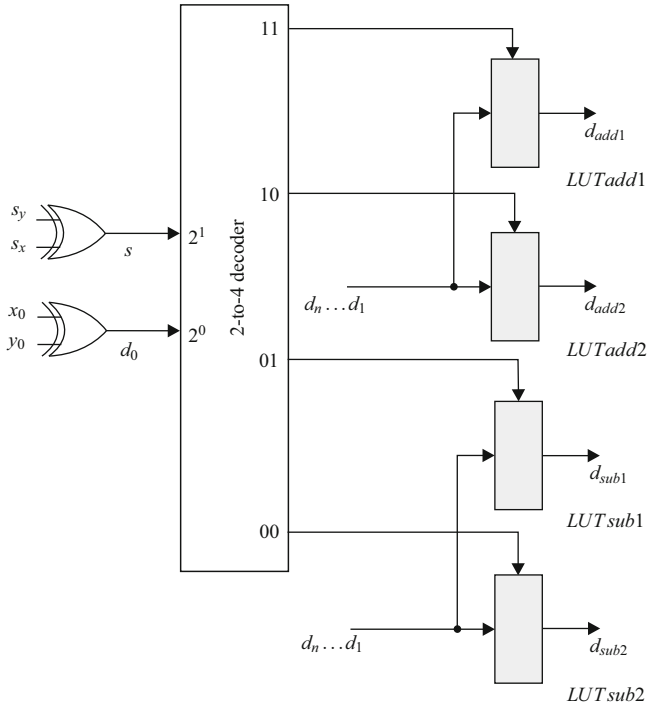

Fig. 10.5 Four-latch organization using LSB d_0 to generate the look-up table selection signals

and the correspondingly maximum possible data rate in all other cases. A two-step procedure has been adopted during synthesis and optimization: initially circuits are optimized for lowest area and subsequently are optimized under delay constraints. Figure 10.7 depicts the results. Specifically, Fig. 10.7a shows that MSB-based architectures achieve lower area complexity than the LSB-based counterpart. This is because the required LUTs become simpler. Figure 10.7b reveals that the MSB selection of d in (10.11) combined with the use of DFFs leads to LNS circuits of lower delay and lower power×delay product than the corresponding combinational or latch-based LNS, for a word length of 12 bits.

Since the utilization of the MSB for LUT selection is not efficient for a latch-based design due to additional hardware used to introduce the required delay to fast paths of the circuit, a solution based on D flip-flops is preferable. Moreover, the flip-flop-based selection is better supported by commercial EDA design flows.

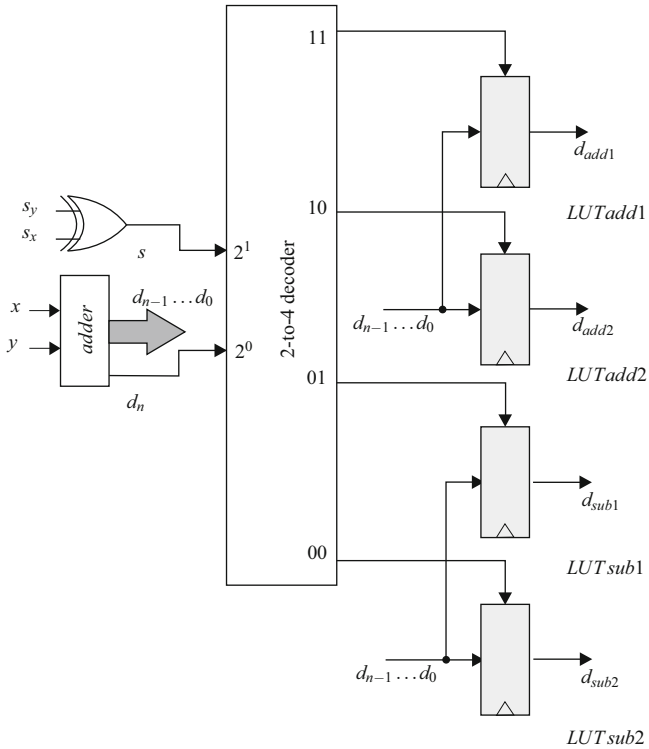

Fig. 10.6 D flip-flop (DFF) organization using the MSB d_n to generate the look-up table selection signals

10.7 LNS and FIR Filter Architectures

The design of low-power low-complexity FIR filters has been studied by several researchers [46, 47]. This section discusses the impact of LNS on the implementation of FIR filters. The performance of FIR filter structures is quantitatively studied, to demonstrate the benefits of employing LNS in FIR filter implementations.

10.7.1 LNS MAC Architectures

An FIR filter is described by

$$Y(n) = \sum_{i=0}^{N-1} C_i X(n-i), \tag{10.42}$$

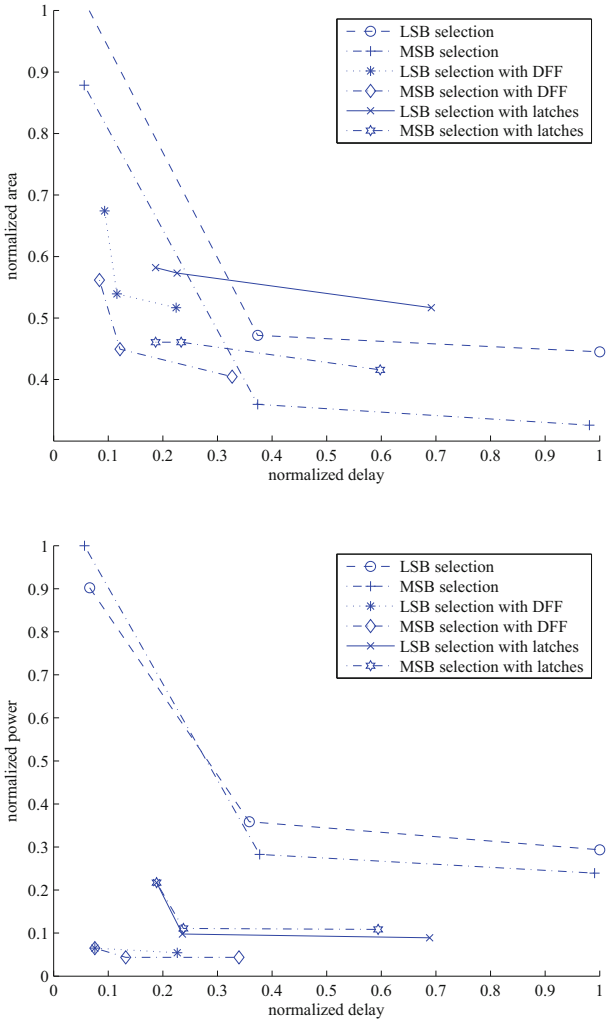

Fig. 10.7 Normalized area and normalized power plots as a function of the normalized delay for 12-bit LNS adder and various address latching schemes. (**a**) Normalized area as a function of the normalized delay for 12-bit LNS adders. (**b**) Normalized power as a function of the normalized delay for 12-bit LNS adders

where C_i are the filter coefficients, $X(n)$ is the input sequence, and $Y(n)$ is the output sequence. The basic structure of the single-MAC unit is shown in Fig. 10.8. Symbols $\otimes$ and $\oplus$ denote a multiplier and an adder, respectively, while D denotes a delay unit, implemented as a register.

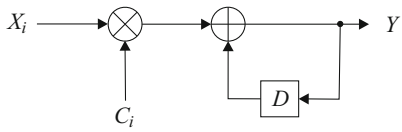

Fig. 10.8 Organization of a single-MAC architecture

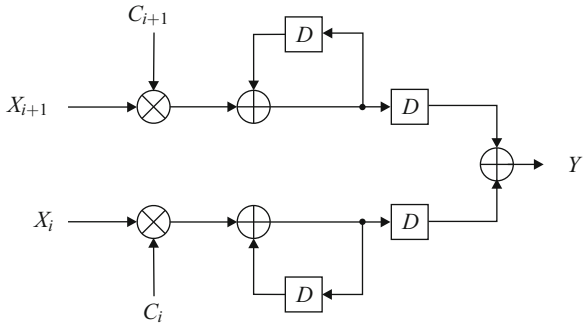

Fig. 10.9 Organization of two-MAC architecture

Computation (10.42) can be partitioned into the computation of two terms, subsequently added as follows:

$$Y(n) = \sum_{j=0}^{\lfloor \frac{N}{2} \rfloor - 1} X(n-2j)C_{2j} + \sum_{j=0}^{\lfloor \frac{N}{2} \rfloor - 1} X(n-2j-1)C_{2j+1}. \tag{10.43}$$

Each term can be allocated to a MAC of a two-MAC architecture, as shown in Fig. 10.9. In general, in order to derive a P-MAC architecture, computation (10.42) is decomposed as

$$Y(n) = \sum_{p=0}^{P-1} \sum_{j=0}^{\lfloor \frac{N}{P} \rfloor - 1} X(n-Pj-p)C_{Pj+p}. \tag{10.44}$$

The pth MAC unit computes $S_p = \sum_{j=0}^{\lfloor \frac{N}{P} \rfloor - 1} X(n-Pj-p)C_{Pj+p}$. Subsequently, all S_p are summed by an addition tree, as shown in Fig. 10.10.

Figure 10.11 depicts the LNS equivalent of a single-MAC architecture, where the binary multiplier has been reduced to an adder, and the binary adder is mapped to an LNS adder/subtractor. The LNS adder/subtractor is augmented with saturation circuitry and exploits a zero flag to avoid unnecessary activation of LUT partitions and further reduce power dissipation. The paths to the final adder in Fig. 10.11 are not balanced, therefore the particular adder exhibits high switching activity. Switching activity further depends on the logic depth of the LUT implementation. Switching activity can be reduced by retiming the circuit so that the register located at the feedback path is replaced by registers placed at the inputs of the final adder, as shown in Fig. 10.12. Power dissipation simulations in [25] have shown that the retimed circuit is more efficient.

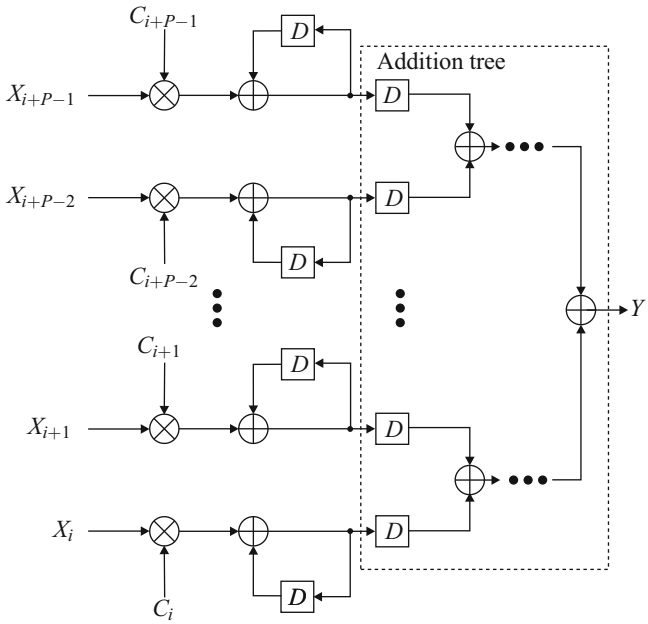

Fig. 10.10 Organization of a *P*-MAC architecture

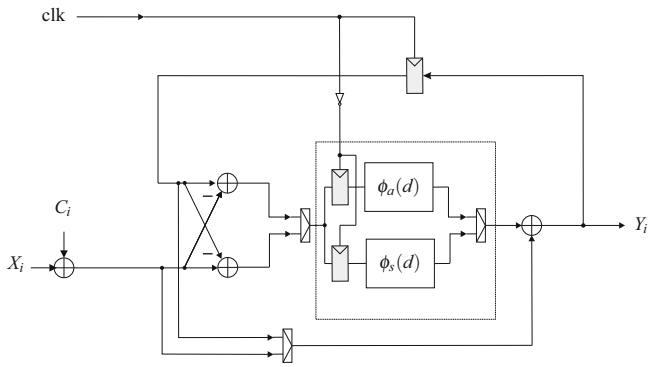

Fig. 10.11 An LNS MAC unit

A test case comprising a 50th-order FIR low-pass filter is reported in [26], with a cut-off frequency of 0.3 rad/s. A zero-mean uncorrelated Gaussian random sequence is used as stimulus. The maximum supported clock frequency is assumed for each simulation, as dictated by the circuit delay. The experiment uses 1000 input data samples. The parameters of the LNS representation (base of the logarithm and word organization) have been selected to achieve at least the same SNR as their linear equivalent [24]. In particular for the 12- and 14-bit LNS FIR implementations the LNS bases used are 1.8 and 1.9, respectively. Equivalent in terms of SNR binary

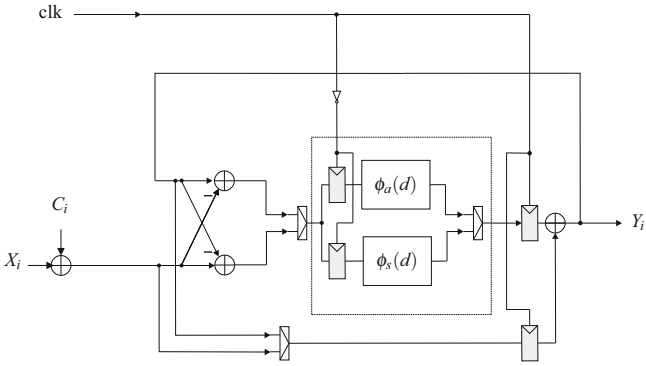

Fig. 10.12 A retimed LNS MAC unit

structures of 13- and 15-bit word length are compared to 12- and 14-bit LNS-based systems. The LNS word length does not include the zero and values sign bit as described in (10.1).

10.7.2 LNS FIR Filter Implementations

In this subsection the impact of LNS MAC units on the implementation of FIR filter structures is reviewed. The employed LNS MACs adopt the MSB-based architectures for LUT partitioning and use DFFs for sub-LUT address latching. Area-delay and power-delay complexity of LNS filters is quantified. The choice of the number of sub-LUTs emerges as a major design choice, which defines the area, delay, and power dissipation. Experimental results reveal that, depending on area-time constraints, a different number of sub-LUTs provide minimum area requirements for a given delay constraint.

Furthermore the power-delay performance for various filter configurations is quantified. Indicative quantitative results are shown in Fig. 10.13, where the Time/sample on the horizontal axis refers to the time required to process one sample of the input sequence, while the vertical axis denotes power P_y that has been consumed for processing per sample, according to

$$P_y = P_{\text{average}}/N_{\text{samples}}, \tag{10.45}$$

where P_{average} denotes the average power consumed during the simulation and $N_{\text{samples}} = 1000$ is the number of samples used for the experiment. Figure 10.13 demonstrates that the Time/sample values decrease proportionally with the number of MAC units. This is because the computational load of processing a particular input sequence is distributed to several parallel MACs units.

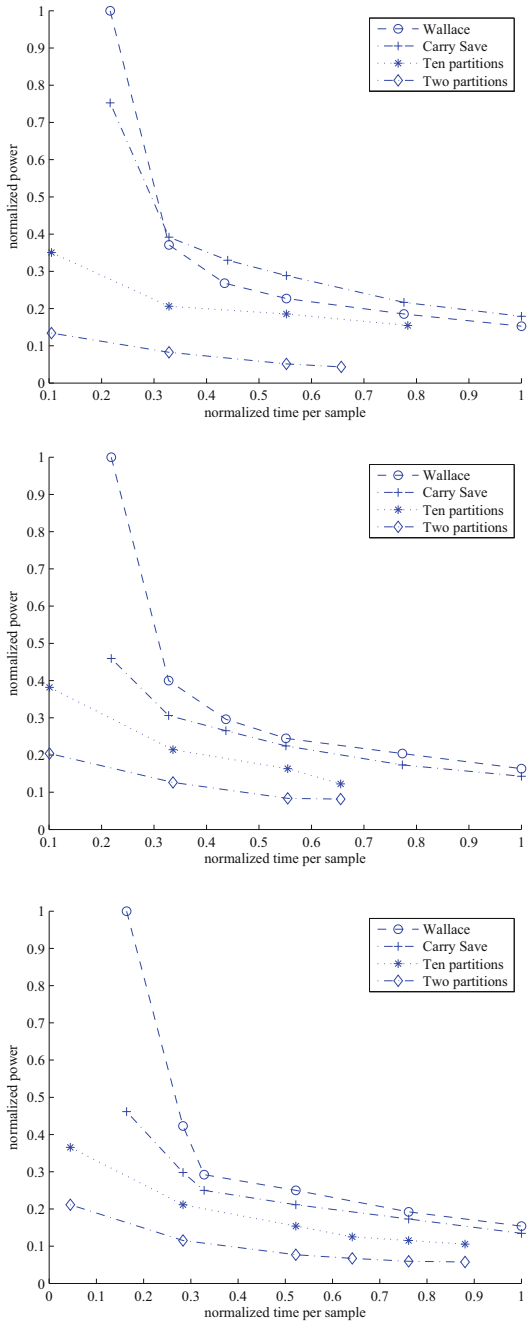

Fig. 10.13 Normalized power as a function of normalized delay for the cases of one-, two-, and four-MAC FIR filter structures. (**a**) Single-MAC FIR structure. (**b**) Two-MAC FIR structure. (**c**) Four-MAC FIR structure

Figure 10.13 depicts the power requirements of several filter implementations that comprise one, two, or four MAC units assuming 12-bit word length. For each of the three cases, LNS adder/subtractors are designed using different numbers of sub-LUTs for the implementation of the addition and subtraction functions $\phi_a(x)$ and $\phi_s(x)$. Several circuits are derived in every case and optimized using different delay constraints. The experimental results depicted in Fig. 10.13 reveal that for lower speed requirements, the most effective solution is a single-MAC architecture. To achieve higher speeds, the number of MAC units should be increased. Depending on the architecture, it is found that a 2-LUT LNS adder/subtractor achieves lower power requirements for a given delay. Further increasing the number of sub-LUTs, increases the power required by the supporting circuitry, canceling the benefits due to smaller sub-LUTs.

In Fig. 10.13 power-delay points of equivalent binary structures, namely Wallace-tree area optimized and carry-save (CSA) structures, are also included. By using two different types of binary multipliers it is shown that the proposed LNS structures are compared in a fair way to the corresponding most competitive binary structures [48]. It can be seen that the four-MAC LNS-based architectures exhibit lower power consumption. A savings factor S is used for comparisons, defined as $S = \frac{P_{c_1} - P_{c_2}}{P_{c_1}}$, with P_c denoting power dissipated by circuit c. Then the lowest-power circuit namely the 2-sub-LUT LNS, achieves 88% power savings when compared to both corresponding Wallace-tree and CSA structures. The 4-sub-LUT architecture is the next most power efficient circuit with 82% power savings. For two-MAC implementations, the 2-sub-LUT lowest-power circuit demonstrates 80 and 55% savings with respect to the Wallace-tree and CSA binary structures, respectively, while the slowest one exhibits 63 and 59% savings when compared to the corresponding slowest binary implementations. For the four-MAC implementations, using the most power efficient two-sub-LUT structures the savings range from 51% for the slowest circuit up to 80% for the fastest one, compared to the corresponding Wallace-tree circuits. Experimental results show that the best choices for the number of sub-LUTs are the 2-sub-LUT and the 4-sub-LUT LNS, for the case of 12 and 14 bits, respectively. This result is consistent with that observation that as the word length increases, gains achieved by partitioning the larger LUTs become more significant compared to the case of smaller LUTs. Furthermore when compared to equivalent binary structures, it is shown that for the word lengths investigated the best LNS-based implementations exhibit lower power dissipation.

Area *vs.* delay results are depicted in Fig. 10.14, for a word length of 12 bits. The x axis represents the circuit latency. Several instances are synthesized under increasing values of delay constraint. As expected, for larger values of allowed delay, area is decreased. The area of the 2-MAC LNS implementation is roughly three times the area of the 1-MAC architecture, and the 4-MAC structure is seven times larger than the 1-MAC architecture. Such area complexities are expected, as they match the number of LNS adders which account for most of the area. However, power scales linearly with the number of multipliers. This is because power dissipation is dominated by the multiply add units, which in turn are dominated by the adder/subtractor. During normal operation, the final adder tree is inactive and its inputs remain latched, therefore practically no dynamic power is dissipated.

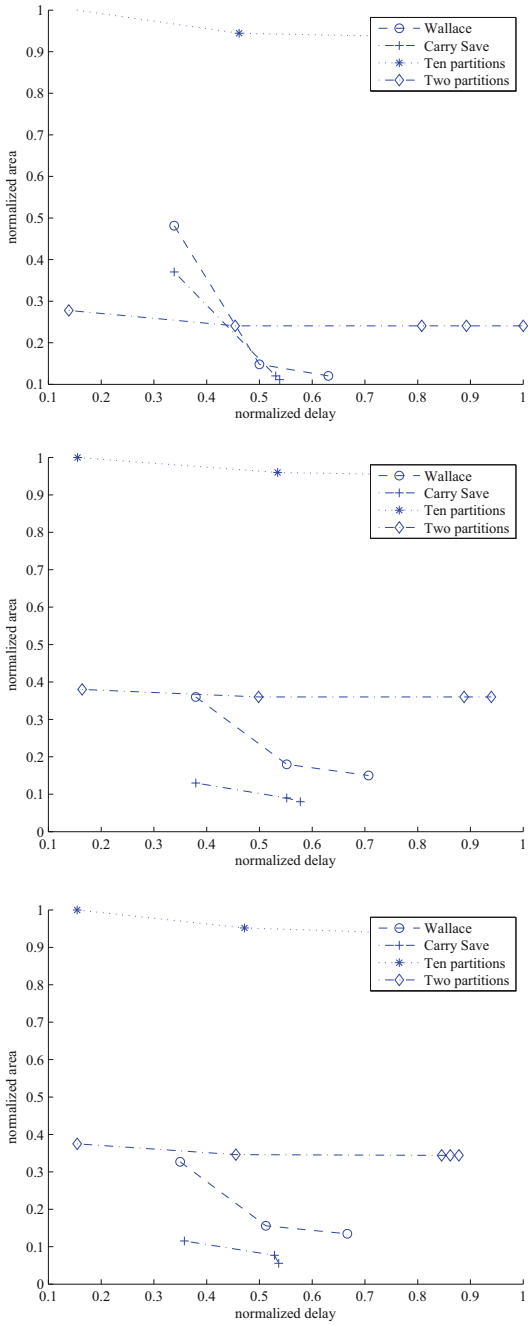

Fig. 10.14 Normalized area as a function of normalized delay for the one-, two-, and four-MAC FIR filter structures. (**a**) Single-MAC FIR structure. (**b**) Two-MAC FIR structure. (**c**) Four-MAC FIR structure

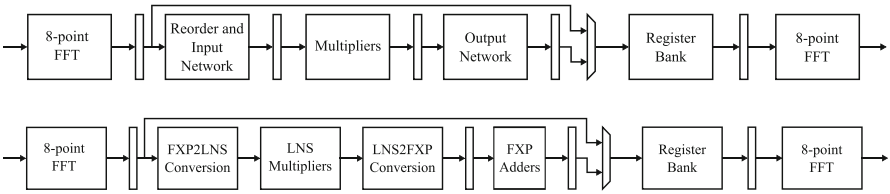

Fig. 10.15 Radix-8 row–column 64-point FFT implemented (**a**) in the linear domain and (**b**) using a hybrid linear-LNS structure

10.8 LNS and FFT

The FFT has found numerous applications in a variety of domains. Among other uses, it is employed in digital filtering applications to compute convolution. Brokalakis and Paliouras [49] have used an optimized LNS representation, in terms of word length, to implement FFT hardware targeting wireless modems that employ OFDM. A hybrid approach was used as LNS was employed to reduce the complexity of multiplications by the twiddle factors; however, FFT butterflies are implemented in the linear domain. Figure 10.15 depicts a linear in comparison to a hybrid linear-LNS FFT architecture. Specifically, the linear row–column radix-8 FFT algorithm for 64-point sequences, as implemented by Maharatna et al. [50], is translated to the LNS domain. The word lengths employed were optimized using a simulation-based approach which relied on the bit-error-rate (BER) achieved at certain noise levels by the architecture, as a measure of equivalence of the LNS to the linear implementation.

Dominant in the complexity of the linear architecture is the cost of multiplications by twiddle factors, which requires an elaborated scheme of interconnections, shared resources, and storage. The adoption of the LNS in the particular application leads to substantial improvements, as it facilitates re-thinking the twiddle factor multiplication problem. Specifically the use of multiple low-cost adders to implement parallel multiplications in LNS resolves datapath hazards existing in the linear counterpart, cutting the cost of overall FFT hardware as well as the complexity of the interconnection networks, control logic, and storage. The cost of forward and inverse conversion is included in the overall evaluation of the solution. Due to the reduced word lengths required, simple conversions are employed having a moderate impact on complexity. An area reduction of 5% is achieved for a 180-nm technology. However, power dissipated is reduced by 23%. Further experiments using a 90-nm technology reveal a power dissipation reduction of the order of 42% due to this hybrid method, as detailed in Table 10.2. Area and power data refer to placed and routed designs obtained using the synopsis tool chain.

Table 10.2 Comparative area and power for a linear and hybrid LNS 64-point FFT

	Linear	Hybrid
Power	1.27	0.73
Area	73,135	71,431

10.9 Extension to Other LNS Schemes

While the simple partitioned-LUT-based LNS adder/subtractor has been used as the basic building block, to investigate LNS FIR filters, the employed design techniques are applicable to other LNS architectures as well.

Specifically, most non-LUT-based LNS adder/subtractor architectures comprise separate sub-units at least for the computation of addition and subtraction, which can be activated selectively, thus resembling the low-power partitioning and selective use of sub-LUTs, as discussed earlier. Furthermore, LNS adder/subtractors comprising paths of substantially different delay, i.e., imbalanced delay paths, may also benefit from the use of retimed MAC units. Imbalanced delay paths occur due to the basic concept of implementing an approximation followed by some post-processing, which stems from the definition of the fundamental LNS operations for addition and subtraction operation in (10.3) and (10.7), respectively. Such paths introduce glitches leading to excessive switching activity that dissipates power. This problem can be addressed by appropriately retiming LNS multiply accumulate units closely resembling the techniques described here.

LNS adders/subtractors comprise several different subsystems and unbalanced paths; i.e., paths of unequal length. Coleman describes a technique that simplifies the structure of tables involved in logarithmic arithmetic. As described in [6] co-transformation is based on storing the values of function $F(r)$,

$$F(r) = \log_2 (1 - 2^r) ,$$

with $r = j - i, j \leq i$. The computation derives an intermediate result denoted r_2, as

$$r_2 = j - i + F(k_1) - F(r_1), \tag{10.46}$$

where k_1 and r_1 are computed as follows:

$$r_1 = (((j - i) \text{ div } m_1) - 1)m_1 = j + k_1 - i \tag{10.47}$$

$$k_1 = i - j + r_1 = -((j - i) \text{ mod } m_1) + m_1. \tag{10.48}$$

The result r_2 is approximated as $j - i$, for the case $r < -1$, while for the case $-1 \leq r < -m_1$, r_2 is computed using (10.46). In both cases, the final result is subsequently obtained through a sub-unit which performs interpolation. When $-m_1 < r < 0$, the final result is obtained as $k_2 = F(k_1)$. Details of the method are discussed in [6].

In [26] it is demonstrated that the concepts reviewed in this chapter can be applied to co-transformation-based LNS circuits; a co-transformation-based subtractor modified to implement the selective activation of required-only sub-units has been implemented, specifically a 12-bit logarithmic-base-2 interpolation-based subtractor using co-transformation is studied. The use of the value of r to decide which of the sub-units should be activated per subtraction; i.e., the use of selective activation decreases power dissipation from 0.55 to 0.45 mW. Power dissipation is further decreased to 0.38 mW by partitioning the largest of the tables used by the interpolator, using the DFF-based sub-unit selection.

10.10 Conclusions

This chapter quantitatively shows that the adoption of LNS can lead to very efficient circuits for digital filtering applications when appropriately selecting the logarithmic base and the word length. An LNS-based system using the proposed adder/subtractors offers substantial power dissipation savings at no performance penalty. Partitioning of the LUTs generates parts in the circuit that can be independently activated thus reducing power dissipation. Furthermore power has been reduced by latching the inputs to the LUTs. In addition gated clocks have been used to further reduce power consumption. The choice of the number of sub-LUTs is an important design parameter that can be exploited for the exploration of the area, time, power design space. Furthermore, the application of retiming is particularly useful in avoiding unnecessary switching activity, due to unbalanced delay paths in LNS arithmetic circuits.

Furthermore base-2 LNS filter SNR models are extended for the case of logarithmic base $b \neq 2$, to facilitate the exploration of a larger design space. By properly defining word length, base, circuit architecture, and LUT organization, the LNS-based MACs can outperform the corresponding TC ones in both power dissipation and performance, for specific practical word lengths.

The design techniques and quantitative performance analysis of LNS MAC units and filter implementations outlined in this chapter show that LNS can offer a viable solution for low-power signal processing systems with moderate word length requirements.

References

1. T. Stouraitis, V. Paliouras, Considering the alternatives in low-power design. IEEE Circuits Devices **17**(4), 23–29 (2001)
2. P.E. Landman, J.M. Rabaey, Architectural power analysis: the dual bit type method. IEEE Trans. VLSI Syst. **3**(2), 173–187 (1995)
3. K.-H. Chen, T.-D. Chiueh, A low-power digit-based reconfigurable FIR filter. IEEE Trans. Circuits Syst. II: Express Briefs **53**(8), 617–621 (2006)

4. E. Swartzlander, A. Alexopoulos, The sign/logarithm number system. IEEE Trans. Comput. **24**(12), 1238–1242 (1975)
5. M.G. Arnold, T.A. Bailey, J.R. Cowles, M.D. Winkel, Applying features of the IEEE 754 to sign/logarithm arithmetic. IEEE Trans. Comput. **41**, 1040–1050 (1992)
6. J. Coleman, C. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, N. Benschop, The European logarithmic microprocessor. IEEE Trans. Comput. **57**(4), 532–546 (2008)
7. V. Mahalingam, N. Ranganathan, Improving accuracy in Mitchell's logarithmic multiplication using operand decomposition. IEEE Trans. Comput. **55**(12), 1523–1535 (2006)
8. K. Johansson, O. Gustafsson, L. Wanhammar, Implementation of elementary functions for logarithmic number systems. IET Comput. Digit. Tech. **2**(4), 295–304 (2008) [Online]. Available: http://link.aip.org/link/?CDT/2/295/1
9. M.G. Arnold, T.A. Bailey, J.R. Cowles, M.D. Winkel, Arithmetic co-transformations in the real and complex Logarithmic Number Systems. IEEE Trans. Comput. **47**(7), 777–786 (1998)
10. V.S. Dimitrov, G.A. Jullien, W.C. Miller, Theory and applications of the double-base number system. IEEE Trans. Comput. **48**(10), 1098–1106 (1999)
11. R. Muscedere, V. Dimitrov, G. Jullien, W. Miller, Efficient techniques for binary-to-multidigit multidimensional logarithmic number system conversion using range-addressable look-up tables. IEEE Trans. Comput. **54**(3), 257–271 (2005)
12. R.C. Ismail, J.N. Coleman, ROM-less LNS, in *IEEE Symposium on Computer Arithmetic* (2011), pp. 43–51
13. H. Fu, O. Mencer, W. Luk, FPGA designs with optimized logarithmic arithmetic. IEEE Trans. Comput. **59**(7), 1000–1006 (2010)
14. M. Arnold, S. Collange, A Real/Complex logarithmic number system ALU. IEEE Trans. Comput. **60**(2), 202–213 (2011)
15. R.E. Morley Jr., G.L. Engel, T.J. Sullivan, S.M. Natarajan, VLSI based design of a battery-operated digital hearing aid, in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (1988), pp. 2512–2515
16. J.R. Sacha, M.J. Irwin, Number representation for reducing switched capacitance in subband coding, in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (1998), pp. 3125–3128
17. M.G. Arnold, Reduced power consumption for MPEG decoding with LNS, in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 02)* (2002), pp. 65–67
18. B. Kang, N. Vijaykrishnan, M.J. Irwin, T. Theocharides, Power-efficient implementation of turbo decoder in SDR system, in *Proceedings of the IEEE International SOC Conference* (2004), pp. 119–122
19. P. Robertson, E. Villebrun, P. Hoeher, A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain, in *Proceedings IEEE International Conference on Communications*, June 1995, pp. 1009–1013
20. H. Wang, H. Yang, D. Yang, Improved log-MAP decoding algorithm for turbo-like codes. Commun. Lett. IEEE **10**(3), 186–188 (2006)
21. R. Peng, R.-R. Chen, Application of nonbinary LDPC codes for communication over fading channels using higher order modulations, in *IEEE Global Telecommunications Conference, GLOBECOM '06*, December 2006, pp. 1–5
22. V. Paliouras, T. Stouraitis, Low-power properties of the Logarithmic Number System, in *Proceedings of 15th Symposium on Computer Arithmetic (ARITH15)*, June 2001, pp. 229–236
23. V. Paliouras, T. Stouraitis, Logarithmic number system for low-power arithmetic, in *Proceedings of International Workshop - Power and Timing Modeling, Optimization and Simulation (PATMOS 2000)*. Lecture Notes in Computer Science, vol. 1918 (2000), pp. 285–294
24. C. Basetas, I. Kouretas, V. Paliouras, Low-power digital filtering based on the logarithmic number system, in *Proceedings of 17th Workshop on Power and Timing Modeling, Optimization and Simulation*, Lecture Notes in Computer Science, vol. 4644 (2007), pp. 546–555
25. I. Kouretas, C. Basetas, V. Paliouras, Low-power Logarithmic Number System addition/subtraction and their impact on digital filters, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'08)*, pp. 692–695 (2008)

26. I. Kouretas, C. Basetas, V. Paliouras, Low-power logarithmic number system addition/subtraction and their impact on digital filters. IEEE Trans. Comput. **62**(11), 2196–2209 (2013)
27. H. Henkel, Improved addition for the logarithmic number system. IEEE Trans. Acoust. Speech Signal Process. **37**(2), 301–303 (1989)
28. D. Lewis, L. Yu, Algorithm design for a 30 bit integrated logarithmic processor, in *Proceedings of the 9th Symposium on Computer Arithmetic*, pp. 192–199 (1989)
29. J. Coleman, Simplification of table structure in logarithmic arithmetic. Electron. Lett. **31**(22), 1905–1906 (1995)
30. V. Paliouras, T. Stouraitis, A novel algorithm for accurate logarithmic number system subtraction, in *Proceedings of the 1996 IEEE Symposium on Circuits and Systems (ISCAS'96)*, vol. 4, May 1996, pp. 268–271
31. I. Orginos, V. Paliouras, T. Stouraitis, A novel algorithm for multi-operand Logarithmic Number System addition and subtraction using polynomial approximation, in *Proceedings of the 1995 IEEE International Symposium on Circuits and Systems (ISCAS'95)* (1995), pp. III.1992–III.1995
32. S. Collange, J. Detrey, F. de Dinechin, Floating-point or LNS: choosing the right arithmetic on an application basis, in *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)* (2006), pp. 197–203
33. P.D. Vouzis, S. Collange, M.G. Arnold, Cotransformation provides area and accuracy improvement in an HDL library for LNS subtraction, in *Proceedings of the 10th Euromicro Conference on Digital System Design (DSD'07)* (2007), pp. 85–93
34. J.-M. Muller, *Elementary Functions – Algorithms and Implementation* (Birkhäuser, Boston 1997)
35. S. Paul, N. Jayakumar, S. Khatri, A fast hardware approach for approximate, efficient logarithm and antilogarithm computations. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **17**(2), 269–277 (2009)
36. J. Kurokawa, T. Payne, S. Lee, Error analysis of recursive digital filters implemented with logarithmic number systems. IEEE Trans. Acoust. Speech Signal Process. **28**(6), 706–715 (1980)
37. I. Koren, *Computer Arithmetic Algorithms* (Prentice-Hall, Englewood Cliffs, NJ, 1993)
38. V. Paliouras, Optimization of LNS operations for embedded signal processing applications, in *IEEE International Symposium on Circuits and Systems (ISCAS), 2002)*, vol. 2 (2002), pp. II-744–II-747
39. D. Chandra, Error analysis of FIR filters implemented using logarithmic arithmetic. IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process. **45**(6), 744–747 (1998)
40. S.W. Kwa, G.L. Engel, R.E. Morley, Quantization noise analysis of sign/logarithm data encoders when excited by speech or sinusoidal inputs. IEEE Trans. Signal Process. **48**(12), 3578–3581 (2000)
41. D.M. Lewis, Interleaved memory function interpolators with application to an accurate LNS arithmetic unit. IEEE Trans. Comput. **43**(8), 974–982 (1994)
42. T. Stouraitis, Logarithmic Number System: theory, analysis and design. Ph.D. dissertation, University of Florida (1986)
43. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 3rd ed. (McGraw-Hill, New York, 1991)
44. F. Taylor, R. Gill, J. Joseph, J. Radke, A 20 bit Logarithmic Number System processor. IEEE Trans. Comput. **37**(5), 190–199 (1988)
45. M. Keating, D. Flynn, R. Aitken, A. Gibbons, K. Shi, *Low Power Methodology Manual: For System-on-Chip Design* (Springer Publishing Company, Incorporated, New York, 2007)
46. C.-H. Chang, J. Chen, A. Vinod, Information theoretic approach to complexity reduction of FIR filter design. IEEE Trans. Circuits Syst. – Part I **55**(8), 2310–2321 (2008)
47. M. Aktan, A. Yurdakul, G. Dundar, An algorithm for the design of low-power hardware-efficient FIR filters. IEEE Trans. Circuits Syst. – Part I **55**(6), 1536–1545 (2008)
48. T.K. Callaway, E.E. Swartzlander Jr., Power-delay characteristics of CMOS multipliers, in *Proceedings of the 13th Symposium on Computer Arithmetic (ARITH13)*, July 1997, pp. 26–32

49. A. Brokalakis, V. Paliouras, Using the arithmetic representation properties of data to reduce the area and power consumption of FFT circuits for wireless OFDM systems, in *2011 IEEE Workshop on Signal Processing Systems (SiPS)*, October 2011, pp. 7–12
50. K. Maharatna, E. Grass, U. Jagdhold, A 64-point Fourier transform chip for high-speed wireless LAN application using OFDM. IEEE J. Solid-State Circuits **39**(3), 484–493 (2004)

Chapter 11
Double-Base Number System and Its Application in FIR Filter Design

Jiajia Chen and Chip-Hong Chang

11.1 Introduction

In the past decade, there has been great advancement in the theories, optimization algorithms, and architecture designs of DSP that drive the technological innovations of multimedia and telecommunication applications. One example is the complexity reduction of digital filters for radio signal transmission and processing in RADAR systems. Images constructed from reflected waves suffer from signal quality degradations in terms of poor contrast, angular loss, and motion induced distortion. The raw radar data needs to be filtered in order to separate the signals, interferences, and noises so that they can be individually analyzed. The output data is then subjected to nonlinear transformation to reconstruct the original domain. Filtering and waveform shaping functions are also the main performance stumbling blocks immediately after the analog-to-digital watershed in RF systems. As technology matures, Application-Specific Integrated Circuit (ASIC) has become more cost-effective to address some of these generic digital signal processing problems. ASIC implementation makes software defined radio (SDR) possible by processing radio frequency (RF) and analog front-end signals in digital domain to eliminate the sophisticated compensation methods needed to provide the agility to power management and simplify post-processing. Still, significant design challenges have been posed on the design of channelizers and frequency-selective filters of SDR due

J. Chen (✉)
Engineering Product Development, Singapore University of Technology and Design, Singapore 487372, Singapore
e-mail: jiajia_chen@sutd.edu.sg

C.-H. Chang
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798, Singapore
e-mail: echchang@ntu.edu.sg

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_11

to the additional constraints on area and power requirements when the bandlimited real-time applications are targeted for migration from the fixed station to the mobile terminals [1, 2]. To achieve an all-digital or a digital-dominant system architecture, the associated kernel functions like convolution, correlation, and linear transformations must be dedicatedly designed.

Finite Impulse Response (FIR) filter is an important class of linear time-invariant systems that is frequently found in DSP system. It can be seen as a computational process or algorithm by which an input sequence of numbers is transformed into a different output sequence of numbers [3]. The operations used to obtain the output signal from the input signal can be completely characterized by their unit-sample (or impulse) response and analyzed by the convolution of two discrete sequences. The arithmetic computations involved in this process are mainly multiplications and additions. Depending on the attenuation characteristics of filter response, the number and complexity of these operators may grow to an extent that the throughput rate, area, and power constraints are difficult to be met with tight design turnaround time [4]. Fortunately, many kernel functions, such as linear transformations, convolution, and correlation, associated with signal conditioning and real-time multimedia signal processing can be modeled as a generic multiple constant multiplication (MCM) problem [5, 6]. When there are multiple multiplications of a variable by different constants, the power hungry multipliers can be decomposed into shifters and a reduced number of less computational intensive adders. Although multiplications by constants appear only in a small fraction of general computation benchmarks, mainly for the purpose of scaling, its frequency of occurrence has been abundant in application-specific computations.

To reduce the long design turnaround time accrued with the desired optimality and complexity of FIR filters in domain-specific applications, algorithms for the design of low-complexity MCM block in FIR filters have been extensively studied [7–14]. Design of dedicated hardware optimized for a fixed coefficient set can then be automated to meet the stringent goals of sampling rate and power budget but at reduced flexibility and scalability. For applications that require the flexibility of multi-services on single expandable platform, the FIR filters need to be made reconfigurable and their design is more complex than fixed coefficient filters [15–19]. Fast reconfigurable Field Programmable Gate Arrays (FPGAs), such as the recent Virtex FPGA family [20], can be a good implementation platform for programmable filters. Automation of both ASIC and FPGA implementations require combinatorial optimization with drastically different emphasis from software compilers for general purpose DSPs. This chapter reviews the existing design methodologies for programmable FIR filter to explore the use of unconventional double-base number system for the design of high-speed, low-complexity, and reconfigurable FIR filters.

11.2 Number Systems in FIR Filter Design

11.2.1 Transpose Direct Form FIR Filter

Practical frequency-selective filters are generally implemented with finite impulse response (FIR) filters in direct or transposed direct form structures. Direct form structure is the most intuitive implementation of the following time domain filter transfer function.

$$y(n) = \sum_{i=0}^{N-1} h(i) \times x(n-i) \tag{11.1}$$

where $x(n)$ and $y(n)$ are the input and output signals, respectively, at the nth sampling instance. N is the total number of taps of the filter and $h(i)$ is the coefficient value at the ith tap.

A direct form structure is a simple tapped delay line in which each delayed versions of the input signal is multiplied by the tap coefficients, and the results are accumulated to form the filter output, as shown in Fig. 11.1. Each block of "Z^{-1}" represents one delay element, or a pipelined stage of the calculation. An FIR "tap" is basically a coefficient multiplier and delay element pair.

Generally, for an N-tap direct form FIR structure, N multiplications and $(N-1)$ additions are required. The critical delay path of direct form structure can be further sped up by retiming. By interchanging the input and output, and reversing all signal flows in a signal-flow graph of single input single output system, the transfer function is unchanged [21]. Based on the transposition theorem, the Z^{-1} delay elements can be repositioned to obtain the transposed direct form structure as shown in Fig. 11.2.

Although the results from [22] demonstrated some area advantage of direct form architecture, transposed direct form has the advantage in throughput rate. Since the

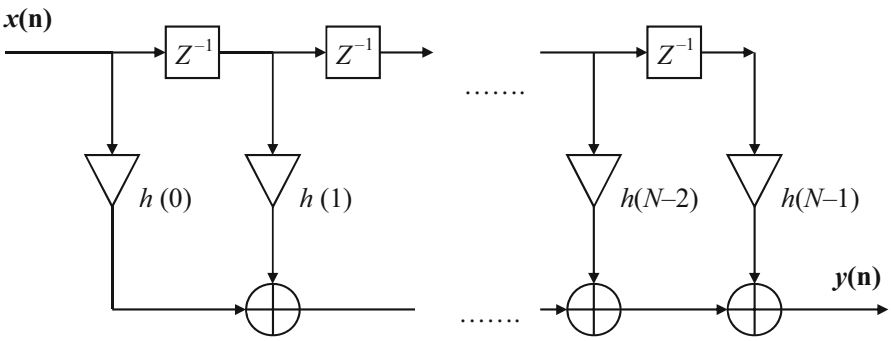

Fig. 11.1 Direct form FIR filter structure

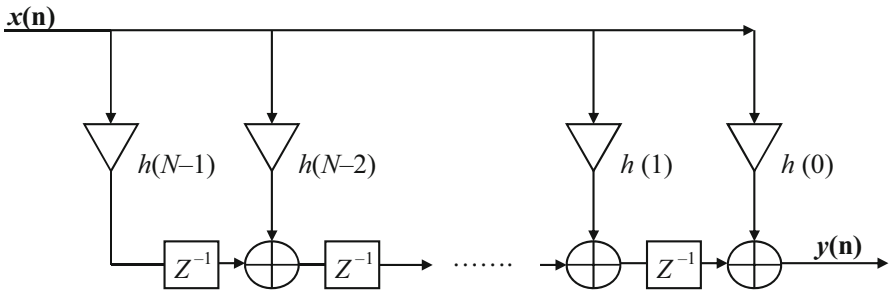

Fig. 11.2 Transpose direct form of FIR filter

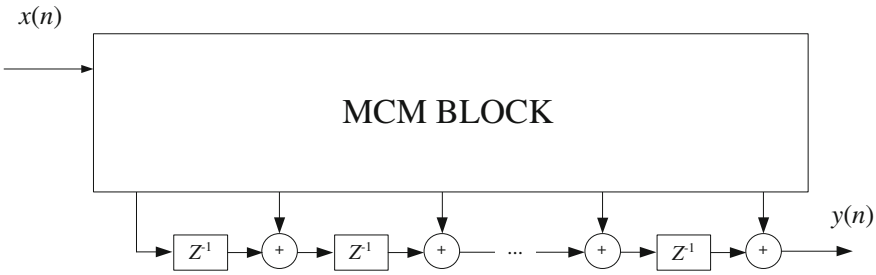

Fig. 11.3 Multiple constant multiplication block of FIR filter

critical path delay is independent of the number of taps, transposed direct form can achieve high throughput without adding extra pipeline registers and has been the preferred architecture for high-speed, high order FIR filter implementation.

11.2.2 Multiple Constant Multiplication Block

The MCM block in transposed direct form FIR filter is shown in Fig. 11.3. Reducing the area, critical path delay, and power of MCM block by substituting the compute intensive multiplications by a network of adder/subtractors and shifters with reduced number of logic operators and logic depth form the basis of various design algorithms. Existing algorithms for optimizing the design of MCM block can be broadly classified into two main categories. The quality of their solutions varies depending on the subexpression representation, the number of non-output fundamentals, dimension of subexpression shifts (in time and spatial distance), and the primary optimization criterion.

The graph-dependence (GD) algorithms [5, 7, 12, 23] are based on graph data structure. They do not rely on specific integer number representation but the numerical values of the coefficients and partial sums to reduce the number of elementary arithmetic operations required to realize the MCM block. The concept of primitive

operator was first introduced by Bull and Horrocks [5]. The implementation of MCM block is fully specified by a reduced adder graph whereby the partial sums called the fundamentals are symbolically encapsulated in the vertices of the graph and are multiplied by a power-of-two factor annotated on the edges. GD algorithms involve the synthesis of a set of connected graphs from the source that represents integer "1" to the sinks that represent the filter coefficients obtained by summing the fundamentals or shifted fundamentals. The fundamentals are generated one at a time depending on the previously generated fundamentals. Such logical graph not only reflects the arithmetic relationship between operators but also the delay of any path in the graph in terms of the delay of a single operator if all operators are assumed to be of equal complexity. The most recently proposed IMA algorithm [12] has iteratively solves the MCM problem from a strongly connected graph formed by adding new vertex from the least cost surviving hypergraph in the preceding iteration until no new vertex is found. As the visible and hidden edges of an elementary hypergraph are interchangeable by swapping their values encoded in its hyperedge, previously removed vertex can be revived if it is a predecessor of the least cost surviving hyperedge as the hypergraph evolves.

The other category of algorithms is based on Common Subexpression Elimination (CSE), which has historically been adopted in computer-aided design in high-level synthesis and has now become a very popular dedicated approach to realizing VLSI efficient FIR filters in recent years [6, 8, 10, 11, 24–26]. The main idea behind all these algorithms is the maximization of the sharing of common subexpressions, or the elimination of replicated subexpressions in the MCM block by pattern matching. Figure 11.4a shows an example of eliminating the common subexpression, $10\overline{1}$, which occurs three times in the coefficients set. This subexpression is eliminated by replacing the pattern $10\overline{1}$ with a symbol S_1. Together with other signed digits, higher weight common subexpressions can be detected and eliminated as shown in Fig. 11.4b. The objective function and the representation of filter coefficients play an important role in the quality of final solutions synthesized by CSE algorithms.

11.2.3 Time-Multiplexed MCM Block

If the FIR filter coefficients are programmable, the multiplier block can be made multiplierless with multiplex-shift-add network and control logic as shown in Fig. 11.5. Ifthe resources used in the multiplier block are allowed to be shared in time, for instance, each output of multiplier block is obtained at one clock cycle by multiplying a variable with one of the predetermined constants, this problem can be formulated as a time-multiplexed multiple constants multiplication (TMCM) [17] as shown in Fig. 11.5. In Fig. 11.5, N is the number of filter taps, $h(i)$ is the ith fixed filter coefficient, and $h(i, t)$ is the ith programmable coefficient at time t. $x[n]$ and $y[n]$ are the nth input and output samples, respectively. TMCM block can be optimized by sharing the operators that are not use concurrently. In most

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & S_1 & 0 & 0 & 0 & S_1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & S_1 & 0 & -1 & 0 & 0 \end{bmatrix}$$

(b)

Fig. 11.4 Example of common subexpression elimination (**a**) before and (**b**) after 10-1 is eliminated

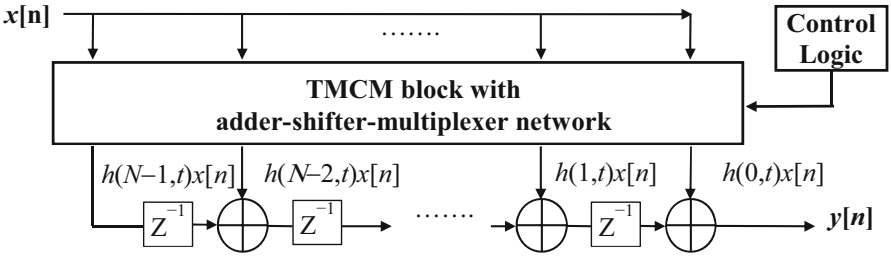

Fig. 11.5 TMCM block of transposed direct form FIR filter with programmable coefficients

standard cell library, a multiplexer occupies around half of the combinational area of an adder of similar bit width. The algorithms in [15–17, 27] optimize the TMCM block design by multiplexing the inputs to adders. These algorithms generally minimize the cost of TMCM block by identifying common subexpressions over all the possible coefficient values in conventional number representation in order to maximally reuse the adders. While CSE techniques are able to harness the sparseness of canonical signed digit (CSD) representation for the complexity reduction of multiplier block in fixed coefficient FIR filters due to its limited number of known coefficients, the enormous search space of unspecified time-varying coefficients in TMCM renders all such derivatives of existing heuristics ineffective and inefficient. The most general form of programmable filter requires a whole gamut of coefficient values of a given precision, including those coefficients that inevitably require more adders and shifters to implement in CSD. Sharing of the subexpressions in these coefficients may not always be possible or profitable due to the temporal dependency of these subexpressions in different programmable coefficient sets across all the filter taps. This motivates the research into more

succinct representation of filter coefficients for the design of hardware efficient TMCM block with reasonable search complexity.

11.2.4 Conventional Number Systems for FIR Filter Design

Different number systems are available for the representation of fixed point coefficients. Each will invoke a different topology of arithmetic operators for the same optimization problem. Although it is merely a choice of numerical representation, alternative representations possess theoretic features that can be manipulated differently for simplifying the amount of shift and add operations involved in the multiplication of a variable with a set of known constants [28]. The algorithmic performance of a design methodology in terms of storage and computation time as well as the quality of its generated solutions is largely affected by certain features of the selected number system, such as the canonicity and sparsity of the representation, the probability of recurrent substrings, and the ease of their identification. Conventionally, binary system is commonly adopted to represent integer constants due to its simplicity and the direct realization of Boolean expressions with basic logic gates. Negative integers are typically represented in two's complement form to avoid the complicated sign and overflow detection. In hardware implementation, addition and subtraction in two's complement form have comparable cost and the total number of hardware resources can be reduced if the use of subtractors can be easily inferred from the positional notation at design time. To employ both adders and subtractors to reduce the hardware cost, radix-2 signed digit, also called binary signed digit representation, is usually used to represent the signed fixed point coefficients in design automation algorithms. In this section, some of the frequently used number representations in digital system design are reviewed.

Binary representation is a way of expressing numbers using two symbols only. The symbols are usually written as 0 and 1, and are called bits. Although numbers written in binary notation are generally much longer than their decimal equivalents, binary notation is employed in digital electronics because digital computers represent numbers in terms of the presence or absence of an electrical pulse by 1 and 0, respectively. More specifically, an integer x can be written in a positional notation with a radix of two as follows:

$$x = \sum_{i=0}^{W-1} b_i 2^i \tag{11.2}$$

where $b_i \in \{0, 1\}$. i is the position of the digits with b_0 as the least significant bit (LSB) and W is the word length of the representation.

Owing to its straightforward implementation using logic gates, the binary system is used internally by all modern computers. However, the first problem of this simple representation is the identification of polarity or sign of negative numbers.

Some offset bias has to be added to shift the dynamic range of the operations so that all operands involved are unipolar. To implement the multiplierless MCM block with binary representation, only additions can be used. Hence the total operator cost is higher than if subtractors can also be used [7]. The second problem of binary number is that the representation uses more nonzero digits than the signed digit representations for the same integer number. Although the binary representation of a number is unique and having greater potential of forming common subexpressions due to its two-symbol characteristic according to [28, 29], the advantage of it is often offset by the higher number of nonzero power-of-two terms over signed digit representations. Another important drawback is that using binary number representation for filter coefficients cannot guarantee minimum logic depth of the multiplier block. From the latest research conducted by [28] to evaluate the adder cost and logic depth of the multiplier block using different number representations, it is concluded that binary representation can produce a very similar performance in adder cost as signed digit representations, but with a conspicuously higher logic depth. The postulations made in [29] for the merit of binary coefficient for CSE are controversial, as pointed out in [17]. For this reason, binary representation is less popular than signed digit (SD) representation for CSE.

SD representation of numbers indicates that the digits can be prefixed with a minus sign to indicate that they are negatively weighted. An SD representation [30] of radix r is a positional number system with a symmetrical digit set, i.e., $a = (a_{n-1} \ldots, a_2, a_1, a_0)$ with $a_i \in \{0, \pm 1, \pm 2 \ldots \pm (r-1)\}$. With $r = 2$, radix-2 signed digit representation consists of three symbols, $\{0, \pm 1\}$. It is widely used in digital arithmetic, such as modified Booth coding. By using three instead of two symbols, both positive and negative integers can be represented in radix-2 SD system as a sum of Signed-Power-of-Two (SPT) terms as shown below:

$$x = \sum_{i=0}^{W-1} s_i 2^i \tag{11.3}$$

where $s_i \in \{0, \pm 1\}$

Unlike binary number, the SD representation of a constant is not unique. Different valid representations of the same coefficient differ in terms of the total number of the nonzero digits and their positions. Without considering subexpression sharing, the representations that have more nonzero digits for the same integer constant cost more adders for the multiplierless implementation of coefficient multiplier. Therefore, SD representations with the minimal number of signed digits are of specific interest. This category of SD representations is called the Minimal Signed Digit (MSD) representation.

MSD representations [31] are crucial in reducing the hardware cost and power consumption of multiply-and-accumulate operations, from the point of view that these representations guarantee the use of the least number of signed digits to represent an integer constant. Any arbitrary n-bit two's complement number can be represented in MSD with no more than $(n + 1)/2$ nonzero digits. On average,

MSD representation uses 33 % fewer nonzero digits than the normal binary number system [32]. The finite space of integers with n power-of-two terms has also been discussed in [33]. It was also shown in [32] that the expected number of nonzero digits in an n-bit MSD number in the range $(-1, 1)$ tends asymptotically to $n/3 + 1/9$. Among the MSD representations, the CSD form [32] is a specific subset, which possesses the following two main features. First, it has the minimal number of nonzero digits. Second, two nonzero digits are never adjacent in a CSD number, which distinguishes it from the other MSD forms. It is these two features that make CSD a unique representation. Any integer number can have one and only one CSD representation.

The hardware complexity of final implementation for a given filter specifications depends not only on the cost function used to direct the coefficients synthesis process, but also the number system chosen to represent the filter coefficients [34]. Due to the necessity to fix the positional notation in order to search for good common subexpressions, CSD [8, 10, 11, 17, 23, 24, 26] and MSD [34, 35] representations have been widely adopted. As CSD is a unique representation while MSD may appear in more than one form for the same integer coefficient, CSD has the advantage of lower search complexity for common subexpressions but MSD is capable of producing better quality solution.

11.3 Double-Base Number System and Programmable FIR Filter Design

11.3.1 Double-Base Number System

Besides the MSD and CSD representations used in FIR filter design, other unconventional number systems with mathematically appealing properties have become so fundamentally embedded into digital design that they present new opportunity to improve the solution quality if they can be incorporated into the design methodology [36]. Double-base number system (DBNS) is one typical example that has been advantageously exploited in several DSP applications and public key cryptography [36]. Although DBNS as a specialized tool for high-performance computing has been considered recently, its application to digital FIR filter design has not been adequately explored and elucidated in a book chapter. This section presents the research effort in exploring DBNS for FIR filter design.

A signed integer coefficient c of a digital filter can be expressed as a sum of n products of two integers in DBNS. A DBNS is defined by two integer bases b_1 and b_2 and assumes the following general form [37]:

$$c = \sum_{i=1}^{n} d_{\alpha,\beta} \times 2^{\alpha} \times 3^{\beta} \tag{11.4}$$

where $d_{\alpha,\beta} \in \{0, 1\}$, and α and β are the exponents of 2 and 3 of the double-base (or 2-integer) term, respectively.

To reduce the number of adders required to implement the product of a programmable coefficient c and an input x, the α power of 2 in the double-base product term can be simply realized by α binary shifts. Thus, the number of adders required to realize the ith partial product of cx will be solely determined by the other factor 3^{β}. In DBNS, three being the smallest integer that has good divisibility is chosen as the other base. Any integer that is not divisible by 3 has a remainder of either 1 ($2^0 3^0$) or 2 ($2^1 3^0$), which can be represented by a simple double-base product. Moreover, CSD representations of $3 = 10\overline{1}$ and $-3 = \overline{1}01$ are the two most frequently encountered common subexpressions in the shift-add network of FIR filters. Making the second base to be 3 will increase the sparsity of the representation and the number of common exponents for all integers within the dynamic range of the filter coefficients.

11.3.2 Canonical Double-Base Number System for FIR Filter Design

Owing to the many possible ways to factor an integer into a sum of double-base products, DBNS representation is highly redundant. For example, the integer 127 has 783 different DBNS representations, of which only three ($2^2 3^3 + 2^4 3^0 + 2^0 3^1$, $2^5 3^1 + 2^0 3^3 + 2^2 3^0$, and $2^2 3^3 + 2^1 3^2 + 2^0 3^0$) have the least number of product terms. To reduce the cost of implementing the product of any w-bit integer coefficient c and an input x, the number of double-base products has to be minimized. A subset of DBNS representations that has the minimum number of double-base products is known as the Canonical Double Based Number System (CDBNS) [37], as shown in Fig. 11.6.

It appears that the complexity of finding the CDBNSs for all the integers of a given word length grows rapidly with the word length. The greedy algorithm in [37] reported that the correct CDBNSs are generated only in 80 % of the 1000 randomly chosen 215-bit integers. In programmable filter design, the word length of filter coefficients rarely exceeds 24 bits. What is more important is to obtain the correct CDBNS for all the coefficients of a desired precision. Hence, a more efficient CDBNS search algorithm is proposed in [18] for all the integer coefficients that can

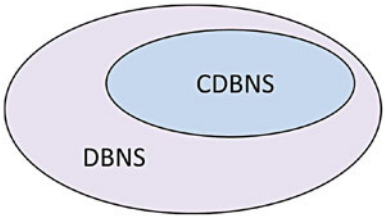

Fig. 11.6 Canonical double based number representation

```
CDBNS_generate(w) {
    Initialize P; // storage for α, β and 2^α 3^β
    Initialize D; // storage for sums of k different double base products
    Initialize CDBNS_array; // storage for CDBNSs of C
    C = set of all w-bit coefficients;
    n = min_terms(w); // lookup for min. no. double base products
    α_max = maximum integer value such that 2^(α_max) < 2^w;
    β_max = maximum integer value such that 3^(β_max) < 2^w;
    for (x from 1 to α_max)
    for (y from 1 to β_max)
        while (2^x 3^y <2^w) add α = x, β = y and 2^x 3^y into P;
        while (C is not empty)
            for (k from 1 to n) {
            D = SODBP(P, k);
            if (c = exist(C, D)) {
                remove c from C;
                update(CDBNS_array, α, β, c); }
            }
    return CDBNS_array;
}
```

Fig. 11.7 CDBNS search algorithm

be represented by a 24-bit binary word. The correctness of the generated CDBNS is guaranteed with reasonable computational complexity. The pseudo code of the algorithm is shown in Fig. 11.7.

The function **CDBNS_generate**(w) returns an array of CDBNSs for all the w-bit coefficients in C. The function **min_terms**(w) returns the minimum number of double-base products needed to represent all the w-bit numbers. The array P stores the exponents for the two bases and the product 2^x3^y that are smaller than 2^w. The function **SODBP**(P, k) returns all the integers that can be obtained by summing k unique double-base products from P. These integers are stored in the array D. The function **exist**(C, D) checks the integers in D against the unprocessed coefficients in C. If there is a match, the matched coefficient c will be removed from C and its CDBNS representation along with the exponents α and β for each of its double-base products is stored in CDBNS_array. The key advantage of this algorithm is that the filter coefficients are checked immediately upon the accumulation from one till n possible double-base products progressively. The sparsest sum of double-base products for a coefficient will be found before all the sums of n products are generated. This way it will guarantee that the CDBNS generated is always correct without having to perform an exhaustive search for the double-base numbers generated from all possible combinations of α and β values. By searching from only

Table 11.1 Frequencies of occurrence of power-of-three factors in CDBNS

w	β									
	1	2	3	4	5	6	7	8	9	10
8	143	69	77	39	12	0	0	0	0	0
	42.1 %	20.3 %	22.6 %	11.5 %	3.5 %	0 %	0 %	0 %	0 %	0 %
10	582	335	349	231	143	70	0	0	0	0
	34.0 %	19.6 %	20.4 %	13.5 %	8.4 %	4.1 %	0 %	0 %	0 %	0 %
12	2322	1485	1493	1060	800	673	258	0	0	0
	28.7 %	18.4 %	18.5 %	13.1 %	9.9 %	8.3 %	3.2 %	0 %	0 %	0 %
14	9345	5998	5950	4519	3799	3080	2326	1762	0	0
	25.4 %	16.3 %	16.2 %	12.3 %	10.3 %	8.4 %	6.3 %	4.8 %	0 %	0 %
16	37,250	25,650	22,903	18,164	15,235	13,837	10,527	9867	6007	2425
	23.0 %	15.8 %	14.1 %	11.2 %	9.4 %	8.5 %	6.5 %	6.1 %	3.7 %	1.5 %

the entries of P, the computational complexity of this algorithm has been reduced to $O(n^2)$. The CDBNS_array can be generated very fast considering that there are at most five double-base products in the CDBNS of any 16-bit number, i.e., $n = 5$ for $w = 16$. From the CDBNSs generated by this search algorithm, the number of occurrences of different 3^β factors in the CDBNSs of all the w-bit integers can be analyzed. This is shown in Table 11.1 for $w = 8$ to 16 and $\beta = 1$ to 10. For each exponent β, its percentage of occurrences over the total number of 3^β factors (for all possible values of β) that can be found in all the CDBNSs of w-bit integers is also listed. Figure 11.8 shows the number of occurrences of the same exponent β in different double-base products of the same coefficient for the 12-bit and 16-bit coefficients. These statistical information are valuable for the development of programmable filter design algorithm.

11.3.3 CDBNS Based Design Method and Examples

As shown in Fig. 11.5, the multiplier block of programmable FIR filter produces N products corresponding to $h(i, t) \times x[n]$ for $i = 0, 1, \ldots, N-1$. To reduce the adder and multiplexer costs of the multiplier block, CDBNS with high sparsity is used to represent the programmable coefficients. Each product can be implemented by a coefficient multiplier consisting of a Common Subexpression Generator (CSG), a Double-Base Product Selector (DBPS) block, and a programmable shifter block, as shown in Fig. 11.9. The CSG is a common shift-add network shared by all the N coefficient multipliers. It generates all the distinct subexpressions for the composition of the distinct 3^β factors of all double-base products in the CDBNS array of w-bit coefficients. RAG-n [7] or any other effective CSE algorithm can be used to reduce the adder cost for the implementation of this block. The DBPS block consists of a bank of n multiplexers with their inputs feeding from the CSG.

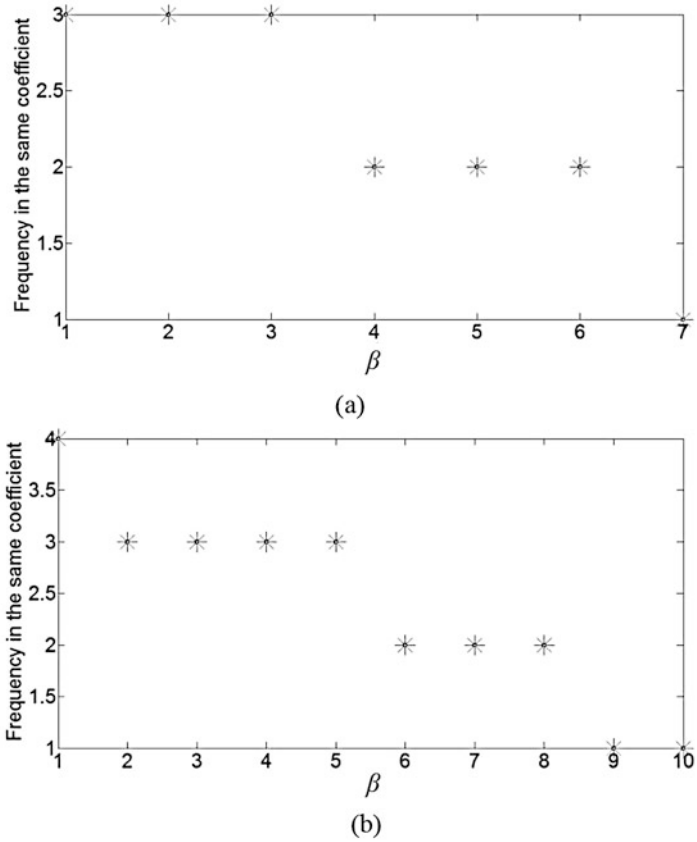

Fig. 11.8 Frequencies of occurrence of power-of-three factors within the coefficient for all the (**a**) 12-bit coefficients and (**b**) 16-bit coefficients

The 3^{β} outputs of the CSG are routed to the inputs of these multiplexers according to its frequency of occurrences in the CDBNSs within the coefficient (see Fig. 11.7). The same 3^{β} factor will appear in multiple multiplexers if it occurs more than once in the CDBNS of the same coefficient. The final shifters shift the outputs of these multiplexers to generate the double-base products $2^{\alpha}3^{\beta}$ for the CDBNS of $h(i, t)$. The control signals to the multiplexers and the programmable shifters are generated by a w-bit LUT.

The design procedure is summarized as follows:

Step 1: Generate the CDBNS array for all the w-bit coefficients using the search algorithm.

Step 2: Implement the CSG by applying the RAG-n algorithm to generate the products of x and all the distinct 3^{β} factors of CDBNS array.

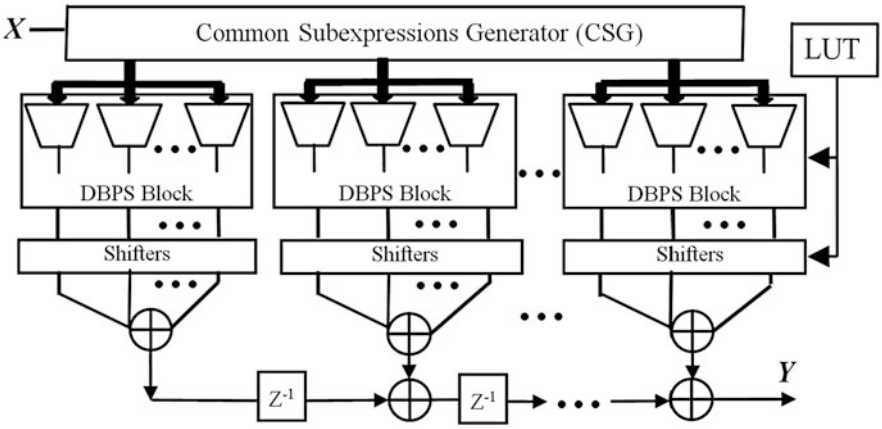

Fig. 11.9 Programmable coefficient multiplier design

Step 3: Design and optimize the multiplexer network for DBPS with the help of the frequencies of occurrence of the 3^{β} factors within the same coefficient and store the control inputs to the multiplexers in an LUT addressable by the value of the coefficient.

Step 4: Add 0 as one of the inputs to the multiplexers. It will be selected if the coefficient is 0.

Step 5: Design n $\alpha_{\max}$-bit programmable shifters, where $\alpha_{\max}$ is the maximum value of e in all the $2^{\alpha}3^{\beta}$ products of CDBNS array. Extract the shifter input α for each $2^{\alpha}3^{\beta}$ product and store it in the LUT addressable by the value of the coefficient.

Step 6: Sum the n shifter outputs by an adder tree.

The first design example using CDBNS is a programmable FIR filter with 8-bit coefficients. 8-bit input signal x is assumed for the estimation of the bit widths of arithmetic operators. The solution generated by CDBNS is shown in Fig. 11.10. Only the architecture of the common CSG of all the coefficient multipliers with the DBPS and shifters of one tap is shown in Fig. 11.10, since the latter two blocks are identical for every coefficient multiplier.

From Fig. 11.10, the implementation comprises four adders in the CSG block, two 4–1 multiplexers and one 3–1 multiplexer in the DBPS block and two adders after the shifter block to sum up the three double-base products. To compare the hardware complexity of CDBNS solution against the binary implementations of the same filter shown in Figure 5 of [15] and Figure 3 of [16], the area cost metric of [27] is adopted. In this model, the complexity of an arithmetic operator or a multiplexer is considered to be proportional to its bit width k. Thus the cost of an operator is ak, where a is a constant of proportionality depending on its structure. Based on the mapping of typical adder, subtractor and multiplexer in a CMOS 0.18 μm standard cell library, $a_{\text{adder}} = 67$, $a_{\text{subtractor}} = 75$ and $a_{p-1\ \text{multiplexer}} = 14p$ [27]. For single coefficient multiplier, the area cost of CDBNS design is 7880 units, which

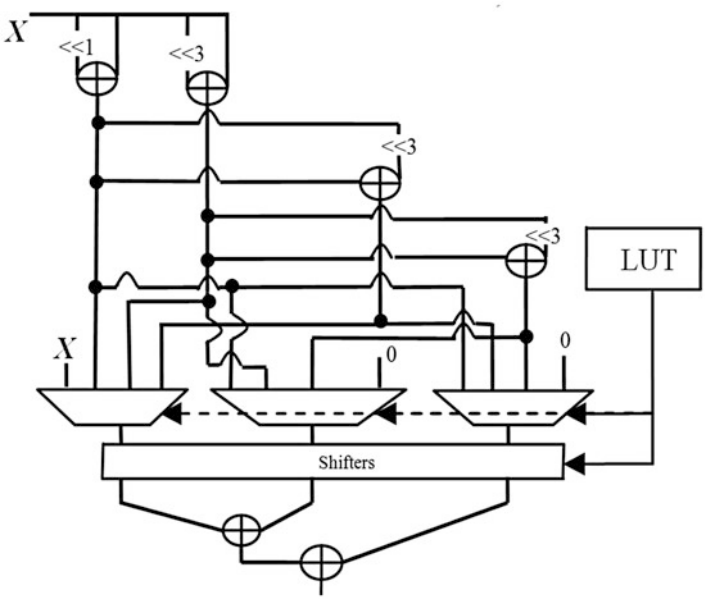

Fig. 11.10 Architecture of a general coefficient multiplier tap for an 8-bit programmable FIR filter designed by CDBNS

is about 25 % lower than 10,554 units of [15] and 4 % lower than 8203 units of [16]. Due to the sparsity of CDBNS, four 3^{β} factors are sufficient to represent all the 8-bit coefficients, which result in the use of only four adders in CSG for all the coefficient multipliers as opposed to nine adders for the generation of all common subexpressions in the precomputer block of [15]. The design of [16] uses only three adders to generate three common subexpressions of 3, 5, and 7 but these common subexpressions can only generate all the 3-bit coefficients. Extension of the 3-bit subexpressions to 8-bit coefficients requires substantial multiplexing costs. Two wide 8-to-1 multiplexers, one 4-to-1 multiplexer, and one 2-to-1 multiplexer are required, which are more costly than the three 4-to-1 multiplexers of DBPS. As the precomputer block is shared by all the coefficient multipliers, its saving is trivial compared to its increase in multiplexer cost which will be duplicated in every coefficient. Consequently, the saving by CDBNS design against that of [16] grows with N, from 4 % for $N = 1$–24.4 % for $N = 100$, as shown in Fig. 11.11.

A 12-bit programmable FIR filter design generated by CDBNS algorithm is shown in Fig. 11.12 as a second design example to demonstrate its ability to handle higher coefficient bit width. This design has an area cost of 11,966 units, which is about 52 % higher than the 8-bit programmable FIR filter.

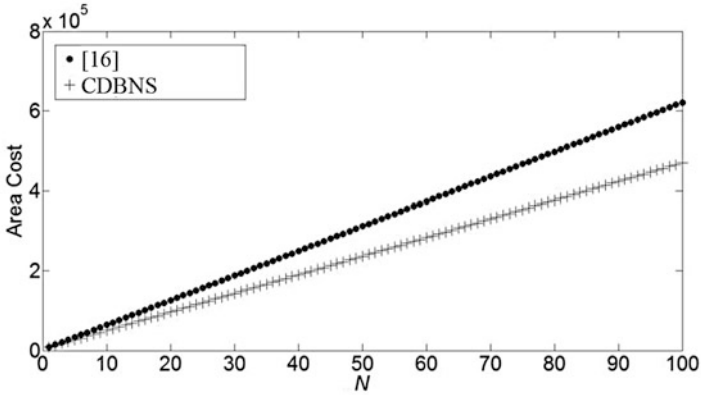

Fig. 11.11 Comparison of area costs with [16] for $N = 1$–100

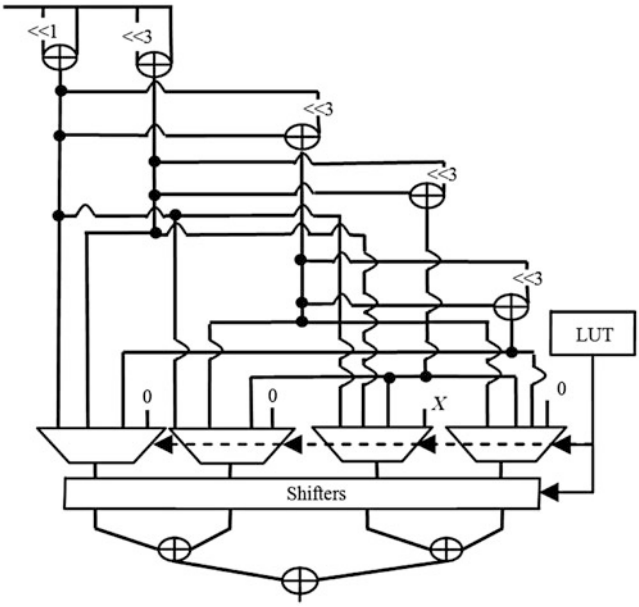

Fig. 11.12 Architecture of a general coefficient multiplier tap in a 12-bit programmable FIR filter designed by CDBNS

11.3.4 Extended Double-Base Number System

As discussed in the previous section, the DBNS with the least number of nonzero double-base terms is called CDBNS. It should be noted that unlike CSD, the integer representation in canonic CDBNS is not unique. Due to the sparsity of CDBNS, any integer within a given range can be generated with fewer additions of

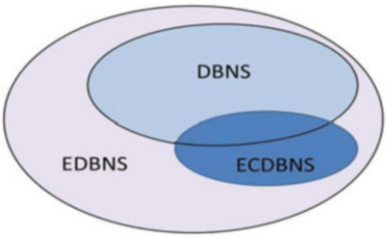

Fig. 11.13 Extended double-base number system

double-base terms. It will therefore be beneficial to explore another integer for the second base b_2 provided that each distinct double-base term can still be generated using only one adder. The subexpression "101" with two adjacent nonzero power-of-two terms in binary and CSD representations and its negation corresponding to the decimal integers 5 and −5 are also frequently encountered. To incorporate this additional common subexpression, the authors of [19] proposed to extend the DBNS to include the next prime integer 5 as a choice for the second base. To differentiate it from the accustomed form of DBNS defined in [36] with 2 and 3 as the only base numbers, this form of DBNS is called the Extended Double Based Number System (EDBNS) [19], whose subset is DNBS when the second base is fixed as 3, as shown in Fig. 11.13.

EDBNS representation for any positive integer c is defined as:

$$c = \sum_{t=1}^{T} 2^{\alpha_t} b_t^{\beta_t} \tag{11.5}$$

where $b_t \in \{3, 5\}$, α_t and β_t are, respectively, the non-negative exponents of 2 and b_t of the tth nonzero double-base term, and T is the total number of nonzero double-base terms. Negative coefficient can be expressed in sign-magnitude form with its magnitude expressed in this EDBNS and its sign used to configure its structural adder as adder or subtractor to achieve the same effect as using signed EDBNS with configurable adder/subtractor.

It can be shown by exhaustive enumeration that every positive integer in the range (0, 256) can be expressed in EDBNS with three or less double-base terms. The numbers of occurrence of EDBNS representations with one, two, and three double-base terms over the entire range are 26, 147, and 82, respectively. On average, only 2.21 double-base terms are required to represent any integer in the range [0, 255]. This is 44.75 % and 17.23 % lower than the binary and CSD representations, respectively.

Similar to the definition of CDBNS, the minimum EDBNS representation of an integer c is an extended CDBNS (ECDBNS) representation of c with the minimum number of nonzero double-base terms $T_{\min}$. The ECDBNS can be constructed as an abstraction of the maximum sharings of two adjacent power-of-two terms in binary and CSD representations over the range of w-bit integers representable by the EDBNS. Let w be the word length of the unsigned binary representation of

the largest magnitude coefficient of a programmable FIR filter. The CSE problem for the TMCM block can then be recast into the problem of searching for a minimum number of distinct power-of-b integers, i.e., b^{β}, where $b \in \{3, 5\}$ and β is a positive integer, to generate the ECDBNS representations for all the integers in the range $(0, 2^w)$.

In general, the highest number of double-base terms $T_{\max}$ for a valid EDBNS representation of any w-bit integer c is $T_{\max} = c$, which is the case when the powers of both bases of all the terms are zero. If there is no restriction imposed in the number of terms T, there are many possible ways to express a w-bit positive integer in DBNS or EDBNS. The ECDBNS representation of each integer in the range $(0, 2^w)$ has different number of terms T. Let $T_{\min}(w)$ be the maximum number of double-base terms among the minimum EDBNS representations of all the w-bit integers. To minimize the number of adders required to add up the double-base terms, $T = T_{\min}(w)$.

Let $F_{\min}(T, w)$ be the smallest set of positive power-of-b ($b \in \{3, 5\}$) integers that appear in any terms of the EDBNS representations of w-bit integers with T or less double-base terms. To obtain $F_{\min}$, all possible power-of-b sets that can be used to represent all the w-bit integers in EDBNS with T or less terms are sought. Among them, the set that has the minimum number of power-of-b integers is selected as $F_{\min}$. If there are two or more sets that have the same minimum number of power-of-b integers, the one with more power-of-3 integers is selected as $F_{\min}$ to avoid complicating the search and comparison. Table 11.2 shows $T_{\min}(w)$ and $F_{\min}(T_{\min}, w)$ obtained by this method for $w = 8$, 12, and 16.

Obviously, with $T = T_{\min}(w)$, the number of adders required to add up the double-base terms of all the w-bit coefficients can be minimized by the minimum EDBNS. However, this may also result in a larger set of distinct power-of-b integers to guarantee that the ECDBNS exists for all the w-bit integers. When adopting EDBNS to design programmable FIR filter, a good trade-off is made by the following rule proposed in [19].

The ECDBNS representations of all the w-bit integers are first generated to obtain $F_{\min}(T_{\min}, w)$. Then, T is incremented to $T_{\min}(w) + 1$ and the quasi-minimum EDBNS representations of T terms are generated. If $|F_{\min}(T, w)| < |F_{\min}(T_{\min}, w)|$, where $|F|$ denotes the cardinality of F, then T is further incremented and the process continues until $|F_{\min}(T-1, w)| = |F_{\min}(T, w)|$.

The cardinality of $F_{\min}$ may decrease when T increases above $T_{\min}$. As the cardinality of $F_{\min}$ stops to shrink with increasing T, the reduction in the quantity and

Table 11.2 $F_{\min}$ with the $T = T_{\min}$ for 8-bit, 12-bit, and 16-bit coefficients

w		8	12	16
$T_{\min}$		3	4	4
$F_{\min}$	$b = 3$	3, 9, 27	3, 9, 27, 81	3, 9, 27, 81, 243, 729, 2187, 6561
	$b = 5$	5	5	5, 25, 125

```
EDBNS(w) {
    C = set of all w-bit coefficients;
    exp3_max = ⌊w/log2 3⌋ // max exponent of power-of-3 integer < 2^w
    exp5_max = ⌊w/log2 5⌋ // max exponent of power-of-5 integer < 2^w
    T = Tmin(w); // min. no. double base terms for w-bit integers
    F_current = Fmin(T, w); // current min. set of power-of-b integers
    F_next = ∅; // next F_min
    S = ∅; // initialize set of EDBNS representations found
    EDBNS_array = ∅; // Initialize storage for EDBNS representations of C
    P = insert(P, 0, 0, 1); // Initial entry for storage of double base terms
    while (F_next ≠ F_current) {
        for (i from 1 to w−1) {
                for (j from 1 to exp3_max)
                    while (2^i 3^j < 2^w) P = insert(P, i, j, 2^i 3^j);
                for (j from 1 to exp5_max)
                    while (2^i 5^j < 2^w) P = insert(P, i, j, 2^i 5^j); }
        while (C is not empty) {
            for (i from 1 to T) {
            S = Gen_EDBNS(P, i);
                I = EDBNS2Int(S) ∩ C;
                if (I is not empty) {
                    EDBNS_array = insert(EDBNS_array, I);
                C = C − I; } } }
        F_current = F_next;
        T = T+1;
        F_next = Fmin(T, w); }
    return EDBNS_array;
}
```

Fig. 11.14 Search algorithm for the quasi-minimum EDBNS [19]

bit width of adders required to realize the distinct power-of-b terms in $F_{\min}$ ceases. Further increment of T will only increase the redundancies of one or more EDBNS representations for all w-bit integers. Hence, the search for EDBNS representations can stop as it will not lead to more efficient hardware implementation. The pseudo code for the EDBNS search algorithm is presented in Fig. 11.14.

The function **EDBNS**(w) returns an array of EDBNS representations for all the w-bit integers in C. The function **Tmin**(w) returns $T_{\min}(w)$ for w-bit integers. The function **Fmin**(T, w) returns the smallest set of power-of-b integers that can be used to represent all the w-bit integers with T or less terms in EDBNS. The function **insert**(P, i, j, t) appends the exponents, i and j, of the two bases and the resultant double-base term t into the array P. The function **Gen_EDBNS**(P, i) generates the EDBNS representations with i unique double-base terms from P. If each of the i double-base terms is obtained in any order, there will be $P_i^i = i!$ permutations of i terms from the same EDBNS representation of an integer. Replicated EDBNS representations due to the permutation of double-base terms are avoided by controlling the loop indices in **Gen_EDBNS**. The function

EDBNS2Int(S) converts the EDBNS representations in the array S into a set of integers. The function **insert**(*EDBNS_array*, I) adds the subset I of coefficients expressed in EDBNS into the array *EDBNS_array*.

This search algorithm reduces the search complexity by seeking only the EDBNS representations in the reduced space from $T_{\min}$ to a value of T that satisfies the criterion of $|F_{\min}(T-1,\ w)| = |\ F_{\min}(T_{\min},\ w)|$. Efficient EDBNS representations with bigger T but smaller $F_{\min}$ will never be missed, which guarantees that the few most efficient EDBNS representations are generated out of many redundant ones without an exhaustive search for all possible combinations. The EDBNS representations sought by this algorithm represent a very small subset of all the EDBNS representations of w-bit integers. For example, for $w = 8$, only two sets of EDBNS representations with $T = 3$ and 4 double-base terms are sought and generated, from which the set with the more succinct representations is selected. Table 11.3 illustrates the frequencies of occurrence of power-of-b integers in 8-bit, 12-bit, and 16-bit integers, respectively, when $T = T_{\min}$. The percentage of occurrence of each power-of-b integer b^{β} over all the power-of-b integers of EDBNS is also listed.

11.3.5 Extended Double-Base Number System for Programmable FIR Filter Design

With EDBNS and its search algorithm presented in 1.3.4, an efficient architecture for the implementation of TMCM is derived as shown in Fig. 11.15. It consists of a power-of-b generator (POBG), N blocks of power-of-b selector (POBS), and N blocks of double-base coefficient generator (DBCG), where b is the second base number in EDBNS and N is the number of taps.

Table 11.3 The occurrences of b^{β} with $T = T_{\min}$ for 8-, 12-, and 16-bit integers

w	b	$\beta = 1$	$\beta = 2$	$\beta = 3$	$\beta = 4$	$\beta = 5$	$\beta = 6$	$\beta = 7$	$\beta = 8$
8	3	377	79	77	0	0	0	0	0
		58.91 %	12.34 %	12.03 %	0 %	0 %	0 %	0 %	0 %
	5	107	0	0	0	0	0	0	0
		16.72 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
12	3	7338	1696	1893	0	0	0	0	0
		50.67 %	11.71 %	13.07 %	0 %	0 %	0 %	0 %	0 %
	5	1825	1729	0	0	0	0	0	0
		12.60 %	11.94 %	0 %	0 %	0 %	0 %	0 %	0 %
16	3	104,644	17,606	17,244	12,746	12,583	10,768	10,902	10,790
		42.83 %	7.21 %	7.06 %	5.22 %	5.15 %	4.41 %	4.46 %	4.42 %
	5	15,184	16,756	15,075	0	0	0	0	0
		6.22 %	6.86 %	6.17 %	0 %	0 %	0 %	0 %	0 %

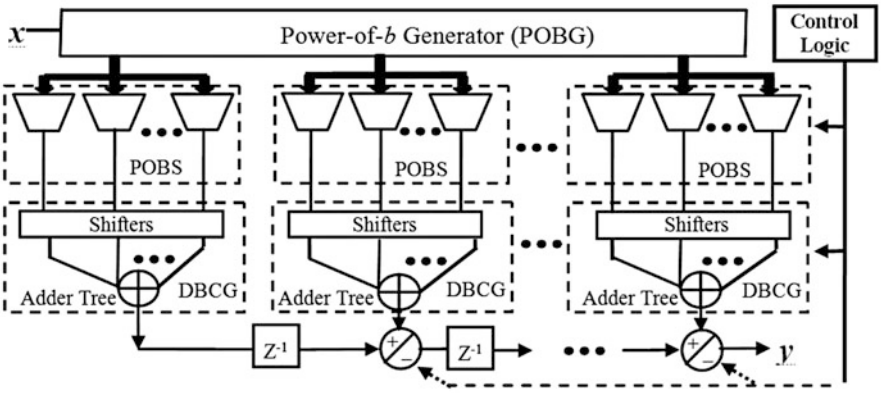

Fig. 11.15 Transposed form FIR filter with programmable coefficients

The POBG block uses the input sample $x[n]$ to produce the set of unique power-of-b integers obtained from the quasi-minimum EDBNS search algorithm presented in Sect. 11.3.4 for all the w-bit coefficients. The set of coefficients $h[i]$, $i = 0, 1, \ldots, N-1$ to be convoluted with the input sample $x[n]$ establishes the control logic of the N POBS blocks to select the desired power-of-b terms from the POBG block. Each POBS block consists of T multiplexers and each multiplexer produces either a value of 0, 1 or a power-of-b integer multiple of $x[n]$. In each successive DBCG block, the selected $b_t^{\beta_t}$ terms are shifted by α_t bit positions according to the quasi-minimum EDBNS representation of $h[i]$. These $2^{\alpha_t} b_t^{\beta_t}$ terms are summed in parallel to produce a product of $h[i]$ and $x[n]$. The outputs of the N DBCG blocks are then delayed and accumulated in the final structural adder block to produce the output $y[n]$.

11.3.5.1 Optimization of POBG Block

Only one POBG block is needed as the set $F_{\min}(w)$ of power-of-b integers b^{β}, $b \in \{3, 5\}$, generated by the algorithm in Fig. 11.14 can be reused by all the N taps. The power-of-b integers in $F_{\min}$ are generated in ascending order of their exponents β in the EDBNS search algorithm of Fig. 11.14. Since $3 = 2 + 1$, the multiplication of an input variable x by 3 can be calculated by $3x = x \ll 1 + x$ using only one adder. Similarly, the product of x and 5 can be calculated by $5x = (2^2 + 1)x = x \ll 2 + x$. In general, the product of an input variable x and every element b^{β} of $F_{\min}$ can be generated by the following recursions for $\beta > 1$:

$$3^{\beta} x = 3^{\beta-1} (2 + 1) x = 3^{\beta-1} x << 1 + 3^{\beta-1} x \tag{11.6}$$

$$5^{\beta} x = 5^{\beta-1} \left(2^2 + 1\right) x = 5^{\beta-1} x << 2 + 5^{\beta-1} x \tag{11.7}$$

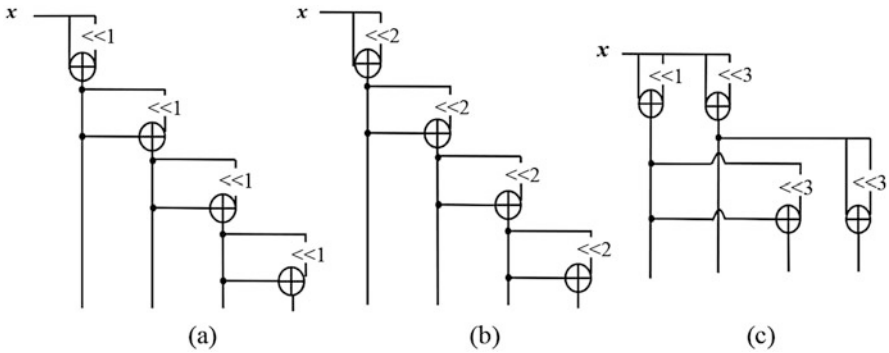

Fig. 11.16 POBG block implementation (**a**) $F_{\min} = \{3, 9, 27, 81\}$ without logic depth optimization (**b**) $F_{\min} = \{5, 25, 125, 625\}$ without logic depth optimization (**c**) $F_{\min} = \{3, 9, 27, 81\}$ with logic depth optimization

Using the above recursion, each successive power-of-*b* integer multiplier needs only one adder to implement. As the shift amount is fixed, it can be hardwired without incurring additional logic. Figure 11.16a, b shows the implementations of the POBG for $F_{\min} = \{3, 9, 27, 81\}$ and $F_{\min} = \{5, 25, 125, 625\}$, respectively. Since all the elements in $F_{\min}$ are odd integers, they are not divisible by any power-of-two integers and cannot be generated from any other elements directly without requiring at least one adder. Therefore, the adder cost for this method of implementation is the optimum.

From, Table 11.3, it is evident that the number of elements for $b = 3$ outweighs that for $b = 5$ in $F_{\min}$. Thus the critical path of POBG block is dominated by the power-of-three integer multipliers. Without incurring additional adder and logic, the logic depth of the POBG can be further reduced for $b = 3$ by generating 3^2x by $x \ll 3 + x$ in parallel with $3x$ in the first adder step. Subsequent products of x and the odd and even power-of-three multipliers 3^β for $\beta > 2$ can also be generated in pair concurrently from their respective preceding pair of odd and even power-of-three multipliers in each adder step by

$$3^\beta x = 3^{\beta-2}x << 3 + 3^{\beta-2}x \tag{11.8}$$

Figure 11.16c shows the reduced logic depth minimum adder cost implementation for $F_{\min} = \{3, 9, 27, 81\}$. This method of implementation reduces the logic depth of the POBG for $F_{\min}$ of $w = 16$ in Table 11.3 by half from 8 to 4.

11.3.5.2 Minimization of POBS by EDBNS Reduction Properties and Exponential Diophantine Equation

Each of the N POBS blocks consists of a bank of T multiplexers with their inputs feeding from the POBG. The products of every power-of-b integers and the input signal x from the POBG are routed to the inputs of these multiplexers according to their frequencies of occurrence in the EDBNS representation of the coefficient. The same partial product may appear in the inputs of several multiplexers of the same POBS block if it occurs more than once in the EDBNS representation of the same coefficient. The implementation cost of a POBS block is determined by the number of multiplexers and the complexity of each multiplexer. The complexity of a multiplexer is approximately proportional to the number and bit width of its inputs. Therefore, one way to reduce the hardware cost of POBS block is to minimize the total number of input lines to the T multiplexers. Although T has been constrained as a whole in the generation of EDBNS representations for all the w-bit coefficients, the EDBNS representations of some coefficients may have fewer than T terms. Therefore, some elements in $F_{\min}$ do not have to be connected to the inputs of all the T multiplexers. Further reduction in the total number of inputs to the multiplexers of a POBS block can be achieved with the help of the following two reduction rules [37] for the DBNS. The second property is also valid for $b = 5$ of EDBNS.

Property 1 Sum of two consecutive double-base terms with the same α.

$$2^{\alpha}3^{\beta} + 2^{\alpha}3^{\beta+1} = 2^{\alpha+2}3^{\beta} \tag{11.9}$$

Property 2 Sum of two double-base terms with the same β.

$$2^{\alpha}3^{\beta} + 2^{\alpha+1}3^{\beta} = 2^{\alpha}3^{\beta+1} \tag{11.10}$$

$$2^{\alpha}5^{\beta} + 2^{\alpha+2}5^{\beta} = 2^{\alpha}5^{\beta+1} \tag{11.11}$$

Obviously, every application of Property 1 reduces the number of double-base terms by one, and the same situation happens for Property 2 provided that the merged higher order term $b^{\beta+1}$ exists in $F_{\min}$. These two properties can be applied recursively until the number of double-base terms in an EDBNS cannot be reduced further. This process can be generalized and modeled as an Exponential Diophantine Equation (EDE) [38].

$$\begin{aligned} &2^{\alpha_1}3^{\beta_1} + 2^{\alpha_2}3^{\beta_2} + \cdots + 2^{\alpha_k}3^{\beta_k} + 2^{\alpha'_1}5^{\beta'_1} + \cdots + 2^{\alpha'_{k'}}5^{\beta'_{k'}} \\ &= 2^{i_1}3^{j_1} + 2^{i_2}3^{j_2} + \cdots + 2^{i_l}3^{j_l} + 2^{i'_1}5^{j'_1} + \cdots + 2^{i'_{l'}}5^{j'_{l'}} \end{aligned} \tag{11.12}$$

where $l + l' < k + k'$.

If (11.12) can be solved without introducing any new element into $F_{\min}$, the existing EDBNS for the coefficient, which is the left-hand side (LHS) of (11.12) will be replaced by the right-hand side (RHS) of (11.12) with less number of terms.

```
POBS_minimize(EDBNS_array, w) {
    EDE_LHS = ∅; EDE_RHS = ∅;// Initialize LHS and RHS of EDE
    for (i from 1 to 2^w–1) {
        j = nTerms(EDBNS_array, i);/ number of double base terms of i
        nPOT = 0; //Initialize the number of shifted power-of-3 terms
        nPOF = 0; //Initialize the number of shifted power-of-5 terms
        while (nPOT + nPOF < j) {
            if (nPOT < j – 1) {
                    nPOT = nPOT + 1; // increment nPOT
            } else {
                    nPOF = nPOF + 1; // increment nPOF
                    nPOT = 0; // reset nPOT }
            LHS = get(EDBNS_array, i);
          RHS = Solve_EDE(LHS, nPOT, nPOF);
            if (RHS ≠ ∅) {
                    replace(EDBNS_array, i, RHS);
                    j = nTerms(EDBNS_array, i);
                    if (nPOF ≠ 0) break; // exit while loop
                    nPOF = nPOF + 1; // increment nPOF
                    nPOT = 0; // reset nPOT  } } }
    return EDBNS_array;
}
```

Fig. 11.17 POBS reduction algorithm

With the aid of *Properties 1* and *2* and the EDE solutions proved in Section 3 of [38], we can match the solutions of (11.12) against the EDBNS representations for all the w-bit coefficients. Once a solution is found, the EDBNS will be replaced by the solution with a reduced number of terms. Eventually, some coefficients are represented with a reduced number of terms even though T for the overall w-bit coefficient sets remains unchanged. The unused input lines to the multiplexers of the corresponding POBS block can then be removed. The reduction process is summarized by the pseudo code in Fig. 11.17.

The functions **nTerms**(*EDBNS_array*, *i*) and **get**(*EDBNS_array*, *i*) return the number of double based terms and the EDBNS expression, respectively, for the integer at the *i*th row of EDBNS_array. The function **Solve_EDE**(*LHS*, *nPOT*, *nPOF*) solves the EDE with *nPOT* binary shifted power-of-three terms and *nPOF* binary shifted power-of-five terms by applying *Properties 1* and *2*. The function **replace**(*EDBNS_array*, *i*, *RHS*) replaces the current EDBNS at the *i*th row of EDBNS_array by the solution *RHS* of EDE. The current minimum number of terms *j* and the expression *LHS* on the left-hand side of EDE is always updated with the latest solution found so that subsequent solution for the same integer, if found, will always have a smaller number of terms. If *LHS* is already the minimum EDBNS, the function **Solve_EDE** will always return a null value. For each integer *i*, the EDE is solved by incrementing *nPOT* before *nPOF*. If the EDE is solved before the increment of *nPOF*, the program continues to search for a better solution by incrementing *nPOF* and resetting *nPOT* until either a solution is found or the sum

of *nPOT* and *nPOF* reaches the number of terms of the current EDBNS of i. As an example, for the 8-bit design with $T = T_{\min} = 3$ and $F_{\min}$ set of {3, 9, 27, 5}, 136 out of the 255 integers can be implemented using two terms only. Another 26 integers can be implemented using only one term. Only 93 integers have to be expressed using three terms, which cannot be further reduced. Thus, the input of each multiplexer in the POBS block is reduced from six to four compared to the previous works [15, 16], where all factors from the precomputer block are connected to the subsequent multiplexers as inputs.

11.3.5.3 Overall Design Flow Using EDBNS and Example

The shifters in the DBCG block shift the T outputs from the POBS to generate the double-base terms $2^{\alpha_t} b_t^{\beta_t}$ for $t = 1, 2, \ldots, T$, and the subsequent adder tree sums up the T terms to produce the coefficient multiplier output $h[i]x[n]$ of each filter tap. Given T and $F_{\min}$, the EDBNS representations of all the w-bit coefficients that result in the minimum number of different α_t for each of the T POBS outputs are selected. Thus, each shifter needs to realize only k different amounts of shift, where k is determined by the minimum number of distinct exponents α_t that can appear in the tth double-base term. Each POBS output is hardwire-shifted by k different numbers of bits and fed into the k data inputs of a multiplexer. One of these k shifted versions will be selected from each multiplexer to compose the EDBNS representation of the programmed coefficient.

The control signals to the multiplexers and programmable shifters are generated by an external look up table (LUT). Because the EDBNS representation of any even integers can be obtained by a double-base scalar multiplication of a 2^{α} factor and an odd integer, i.e., $c = 2^{\alpha} \times c'$ where c is an even number and c' is a fundamental [7], the control information for the even integers are stored together with their fundamentals in the LUT, plus a 2^{α} factor stored for each even number. This will reduce the LUT size by almost half. The complete design procedure is summarized as follows:

Step 1: Compute $T_{\min}$ and $F_{\min}$ for all the w-bit coefficients. Generate the EDBNS array for all the w-bit coefficients using the search algorithm presented in Sect. 11.3.4.

Step 2: Implement the POBG block by producing all the power-of-b integers in $F_{\min}$ using the method described in Sect. 11.3.5.1.

Step 3: Design the POBS block with T multiplexers. Each power-of-b integer from the POBG block is first connected to an input of k different multiplexers, where k is the maximum number of times that the power-of-b value can appear in the EDBNS representation of any coefficient. Then, minimize the number of input lines to the multiplexers of POBS block by the algorithm presented in Fig. 11.17.

Step 4: Design T programmable shifters for the DBCG block. Extract the amount of shifts α for each power-of-b integer and store it in the LUT addressable by the fundamentals.

Step 5: Sum the T double-base terms in the DBCG by a carry save adder (CSA) tree to reduce the delay.

Design Example Consider the design of a programmable FIR filter with 8-bit coefficients. According to Table 11.3, $T_{\min} = 3$. By fixing $T = T_{\min} = 3$, $F_{\min} = \{3, 9, 27, 5\}$ is obtained. By relaxing T to $T_{\min} + 1 = 4$, $F_{\min} = \{3\}$ is obtained. It is obvious that the cardinality of $F_{\min}$ cannot be further reduced by incrementing T. Two sets of EDBNS representations for all the 8-bit coefficients can be generated using $F_{\min} = \{3\}$ for $T = 4$ and $F_{\min} = \{3, 9, 27, 5\}$ for $T = 3$. The former EDBNS expresses all possible 8-bit coefficient values using only one power-of-b integer in the POBG but four double-base terms are to be added in each DBCG block. The latter EDBNS uses four power-of-b integers in the POBG to express all possible 8-bit coefficient values but only three double-base terms need to be added in each DBCG block.

Using the EDBNS with $F_{\min} = \{3, 9, 27, 5\}$ and $T = 3$ as an example, the POBG block is designed as shown in Fig. 11.18, where $3x$, $5x$, and $9x$ are produced using only one adder each, while $27x$ is generated from $3x$ by $3x \ll 3 + 3x$ with one additional adder. The number of adders is 4 and the logic depth is only 2. Since $T = 3$, only three multiplexers are needed in each POBS block. To reduce the input lines to the multiplexers, the EDEs for the EDBNS of all the 8-bit coefficients are solved by using the algorithm presented in Fig. 11.17. The number of double-base terms is reduced for the EDBNS representation of each coefficient that has an EDE solution. For example, 147 is initially expressed as $2^7 3^0 + 2^4 3^0 + 2^0 3^1$. The EDE $2^7 3^0 + 2^4 3^0 + 2^0 3^1 = 2^{\alpha_1} 3^{\beta_1} + 2^{\alpha_2} 3^{\beta_2}$ has a solution of $\alpha_1 = 0$, $\alpha_2 = 4$, $\beta_1 = 1$, and $\beta_2 = 2$. Hence, the EDBNS representation for 147 is reduced to $2^0 3^1 + 2^4 3^2$ and one term has been saved. Upon minimizing the members of EDBNS array, each multiplexer of POBS block needs only four instead of six data inputs, including 0 (for 0 valued coefficient) and x (for $b^0 x$). Finally, each multiplexer output is connected to a programmable shifter in the DBCG to generate a double-base term, and the three double-base terms are added up by an adder tree of two adders. Together with the tap-delay registers and configurable structural adder/subtractors, the final design of this programmable filter is shown in Fig. 11.18.

11.4 Logic Synthesis Results and Comparison with Programmable FIR Filters Designed by Binary Number Systems

11.4.1 ASIC Synthesis Results

More than 100 programmable filters with the number of filter taps ranging from 10 to 100 in step size of 10 and coefficient word lengths of 8, 12, and 16 bits are designed by the EDBNS method and two existing programmable FIR filter design

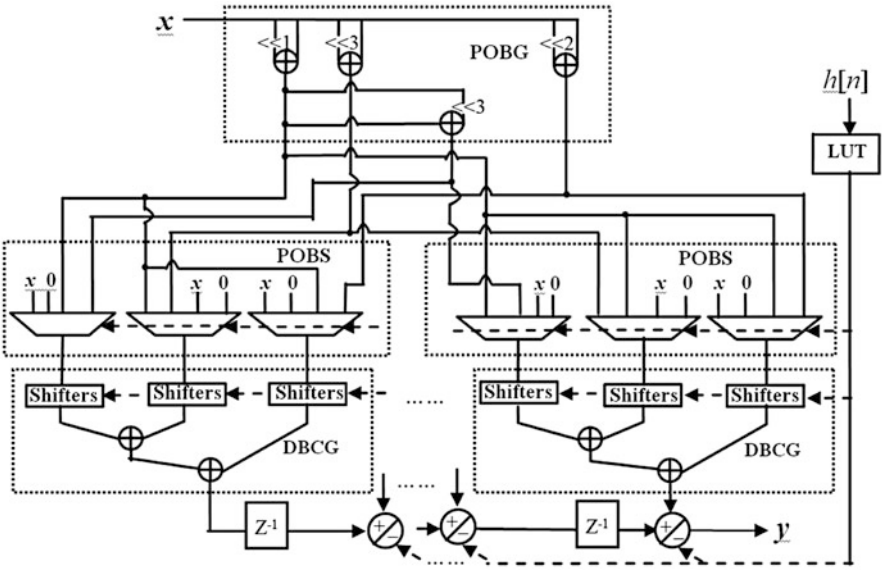

Fig. 11.18 8-bit programmable FIR filter designed by EDBNS algorithm

methods [15, 16], which are based on binary number system. The LUT and control logic circuit are independent of the filter taps and represent a small fraction of the total cost. An input word length of 8 bits is assumed. All the programmable filter architectures are described in VHDL codes and synthesized by Mentor Graphics LeonardoSpectrum using 0.18 μm standard cell library. As the number of structural adders and the registers in the tap-delay line are identical in all three design methods, only the synthesis results of the TMCM blocks of these programmable filters are compared.

More than one set of quasi-minimum EDBNS representations with different T and $F_{\min}$ is generated by the search algorithm for a given coefficient word length. Table 11.4 shows the synthesized areas in equivalent gate counts of the TMCM designs by EDBNS based algorithm with different values of T for 8-bit, 12-bit, and 16-bit coefficients.

On average, the gate count for $T = 4$ and $F_{\min} = \{3\}$ is about the same as that for $T = 3$ and $F_{\min} = \{3, 9, 27, 5\}$ for programmable filters with 8-bit coefficients. For 12-bit coefficients, the designs implemented by the EDBNS with $T = 5$ and $F_{\min} = \{3, 9, 27\}$ have lower gate count when the number of filter taps is smaller than 10, whereas the designs implemented by the EDBNS with $T = 4$ and $F_{\min} = \{3, 9, 27, 81, 5\}$ are more hardware efficient when the number of taps increased above 10. For 16-bit coefficients, the designs implemented by the EDBNS with $T = 6$ and $F_{\min} = \{3, 9, 27\}$ have the least hardware cost compared with those implemented by the EDBNS with $T = 4$ and $T = 5$. The results indicate that the most efficient TMCM implementation is neither dictated by T nor $F_{\min}$ alone. The complexity

Table 11.4 Gate counts of programmable filters of 1–100 taps designed with quasi-minimum EDBNS of different T for 8-bit, 12-bit, and 16-bit coefficients

	$w=8$		$w=12$		$w=16$		
N	$T=3$	$T=4$	$T=4$	$T=5$	$T=4$	$T=5$	$T=6$
1	406	271	568	497	1433	841	576
10	2526	2402	3556	3785	8517	5418	4640
20	4861	4774	6833	7423	16,264	10,435	9112
30	7176	7129	10,083	11,056	23,939	15,414	13,596
40	9501	9490	13,358	14,708	31,687	20,437	18,095
50	11,850	11,853	16,642	18,348	39,407	25,462	22,569
60	14,157	14,208	19,900	21,973	47,119	30,450	27,055
70	16,482	16,573	23,160	25,610	54,818	35,457	31,533
80	18,831	18,932	26,459	29,253	62,602	40,489	36,020
90	21,159	21,280	29,710	32,876	70,304	45,493	40,502
100	23,483	23,653	32,975	36,527	78,014	50,496	45,000

Table 11.5 Comparison of the gate counts of 8-bit coefficient programmable filters of 10–100 taps designed by [15, 16] and EDBNS

N	[15]	[16]	EDBNS	ΔA [15] (%)	ΔA [16] (%)
10	2814	3709	2402	14.64	35.24
20	5238	7240	4774	8.86	34.06
30	7694	10,798	7129	7.34	33.98
40	10,178	14,358	9490	6.76	33.90
50	12,600	17,889	11,853	5.93	33.74
60	15,062	21,461	14,208	5.67	33.80
70	17,504	25,006	16,573	5.32	33.72
80	19,967	28,558	18,932	5.18	33.71
90	22,403	32,103	21,280	5.01	33.71
100	24,869	35,669	23,653	4.89	33.69

of POBG block is governed by the number of power-of-b integers, which is the cardinality of $F_{\min}$. On the other hand, the multiplexer cost of each POBS block is affected by the word length of power-of-b integers and the number of double-base terms T. Since only one POBG block is required irrespective of the number of filter taps N as opposed to one POBS block for each tap, the total hardware cost will be dominated by the efficient implementation of POBS block more than the efficient implementation of POBG block as N grows. This is observed in the case of 12-bit coefficients when $N > 10$. However, it should be highlighted that a larger T does not necessarily incur higher multiplexer cost in the POBS block. This is because the complexity of POBS block is also dependent on the size and number of power-of-b integers. Those designs with larger T but more efficient POBS blocks have consistently lower overall complexity regardless of N, as evinced by the 16-bit coefficient designs with $T = 6$.

Using the best design for each filter from Table 11.4, we compare the gate counts of programmable filters designed by the EDBNS against those designed by the binary methods [15, 16], in Tables 11.5, 11.6, and 11.7 for 8-bit, 12-bit, and 16-bit coefficients, respectively. The percentage area reduction of EDBNS solutions

Table 11.6 Comparison of the gate counts of 12-bit coefficient programmable filters of 10–100 taps designed by [15, 16] and EDBNS

N	[15]	[16]	EDBNS	ΔA [15] (%)	ΔA [16] (%)
10	4902	6003	3556	27.46	40.76
20	9473	11,872	6833	27.87	42.44
30	14,044	17,719	10,083	28.20	43.09
40	18,615	23,593	13,358	28.24	43.38
50	23,185	29,447	16,642	28.22	43.48
60	27,724	35,301	19,900	28.22	43.63
70	32,276	41,183	23,160	28.24	43.76
80	36,850	47,058	26,459	28.20	43.77
90	41,520	52,912	29,710	28.44	43.85
100	46,091	58,766	32,975	28.46	43.89

Table 11.7 Comparison of the gate counts of 16-bit coefficient programmable filters of 10–100 taps designed by [15, 16] and EDBNS

N	[15]	[16]	EDBNS	ΔA [15] (%)	ΔA [16] (%)
10	6714	8755	4640	30.89	47.00
20	13,132	17,345	9112	30.61	47.47
30	19,487	25,963	13,596	30.23	47.63
40	25,861	34,561	18,095	30.03	47.64
50	32,238	43,159	22,569	29.99	47.71
60	38,593	51,785	27,055	29.90	47.76
70	45,068	60,390	31,533	30.03	47.78
80	51,487	68,994	36,020	30.04	47.79
90	57,847	77,585	40,502	29.98	47.80
100	64,221	86,218	45,000	29.93	47.81

over those of [15, 16] are calculated in the last two columns labeled "ΔA [15]" and "ΔA [16]," respectively. The equivalent gate counts against the number of filter taps N are also plotted in Fig. 11.19 to show the trend of improvements as N increases.

The results show that the TMCM blocks of programmable FIR filters designed by the EDBNS have much lower logic complexity. Its average reductions in gate count over [15] are 6.96, 28.16, and 30.16 % for 8-bit, 12-bit, and 16-bit coefficients, respectively. Its average gate count reductions over [16] are 33.96, 43.21, and 47.64 % for 8-bit, 12-bit, and 16-bit coefficients, respectively. The reduction can be as high as 30.89 % over [15] for $N = 10$ and 47.81 % over [16] for $N = 100$ for the 16-bit coefficient filters. The savings are attributable to the sparsity of the quasi-minimum EDBNS, which effectively eliminates the high redundancies of common subexpressions encapsulated in its power-of-b terms. The adders and multiplexers of POBG and POBS blocks have also been reduced significantly by the EDBNS. As the number of filter taps grows, the net saving of POBS block multiplies, which leads to a more prominent overall area reduction.

The throughput rate is another important criterion especially for programmable filter as it determines the rate of adaptation and how fast the input signal can be sampled. If the critical path delay is longer than the time interval between two samples, the input samples need to be buffered. Owing to the parallel architecture

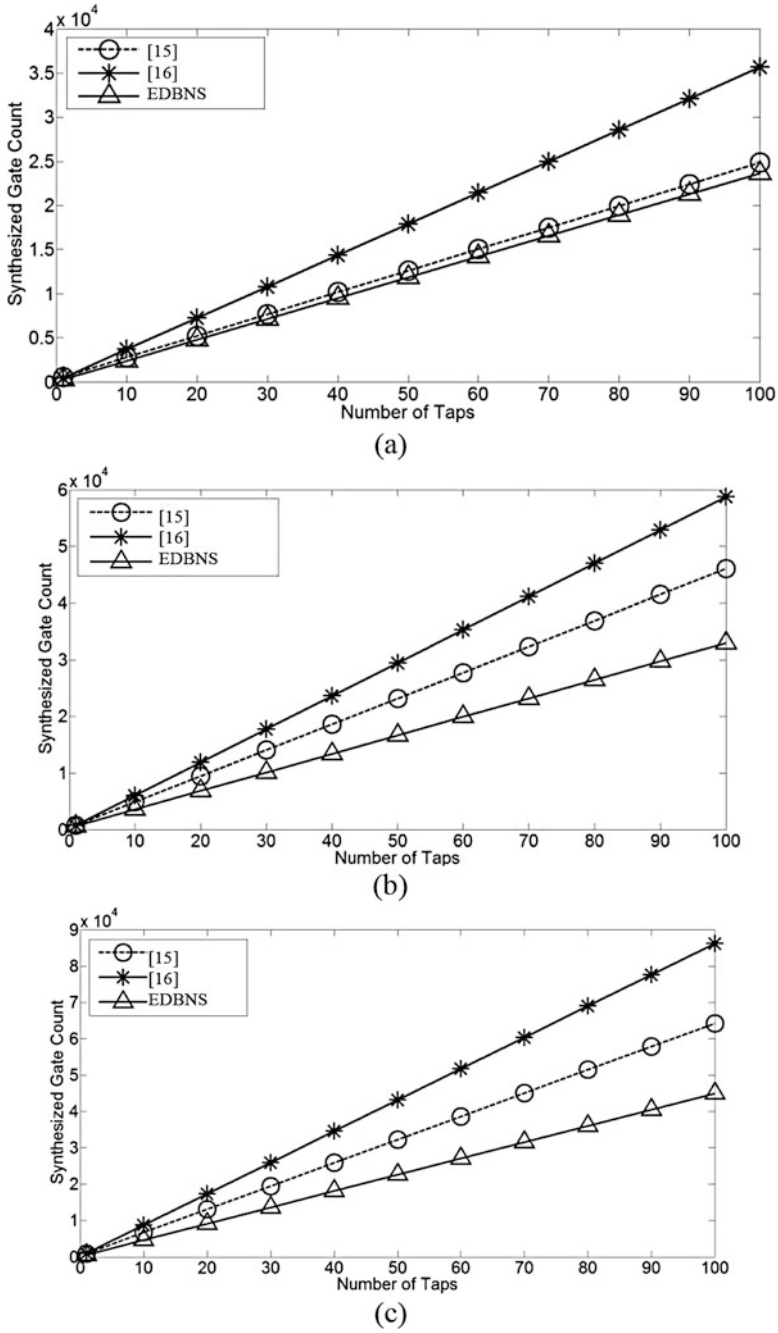

Fig. 11.19 Gate count versus number of taps for (**a**) 8-bit, (**b**) 12-bit, (**c**) 16-bit programmable FIR filters designed by [15, 16] and EDBNS

Table 11.8 Comparison of critical path delays in NS of the w-bit programmable filters designed by [15, 16] and EDBNS

w	8	12	16
[15]	3.37	3.80	4.42
[16]	3.16	3.97	5.03
EDBNS	2.59	4.14	3.57
ΔD [15] (%)	23.15	−8.95	19.23
ΔD [16] (%)	18.04 %	−4.11	29.03

of the N identical POBS blocks in an N-tap filter, the delay is independent of N but dependent on the word length of the coefficients. The critical path delays of the TMCM blocks of 8-bit, 12-bit, and 16-bit coefficient programmable filters designed by the EDBNS [15, 16] are compared in Table 11.8. The percentage reduction in delay over [15, 16] is calculated in the last two rows labeled "ΔD [15]" and "ΔD [16]," respectively.

The POSG block lies in the critical path and its logic depth is a major contributor to the delay of the filter. Thanks to the reduced logic depth implementation of POSG block, although the hardware reduction by the EDBNS over [15] for 8-bit coefficient filters is not as significant, its critical path delay is much shorter. It should be noted that $F_{\min} = \{3, 9, 27\}$ is used for the 16-bit EDBNS while $F_{\min} = \{3, 9, 27, 81, 5\}$ is used for the 12-bit EDBNS. Due to the larger discrete adders used for the generation of $81x$ of $F_{\min}$ in the POBG block, the delay of 12-bit coefficient filter architecture is longer than those of [15, 16] and even 16-bit programmable filter (although the POBG blocks for the 12-bit and 16-bit coefficients have the same adder depth). On average, the critical path delay of TCM block designed by the EDBNS is reduced by 11.14 and 14.32 % in comparison with those designed by [15, 16], respectively.

Taken into consideration the importance of both logic complexity and logic depth, the area-time (AT) complexity, measured in terms of the product of equivalent gate count and critical path delay in ns, are plotted in Fig. 11.20 for the 50-tap filters with coefficient word lengths of 8, 12, and 16 bits. It is evident that the AT complexity of the filters designed by the EDBNS is lower than those of [15, 16] by significant margins of 31.0 and 49.9 %, respectively.

11.4.2 FPGA Synthesis

Comparison of multiplierless programmable and parallel MAC FIR filters is not provided in existing publications with a tacit understanding that adder is cheaper, faster, and more power efficient than multiplier. Nonetheless, an 8-bit 100-tap programmable FIR filter using the MAC approach is designed to compare with the EDBNS. Both designs are mapped to Xilinx Artix 7 FPGA using Xilinx ISE Design Suite 14.7. Excluding the programmable shifters, the EDBNS consumes 7210 logic slices and has a critical path delay of 6.37 ns, which is around 37.8 and 41 % lower than the MAC method. With the inclusion of programmable shifters,

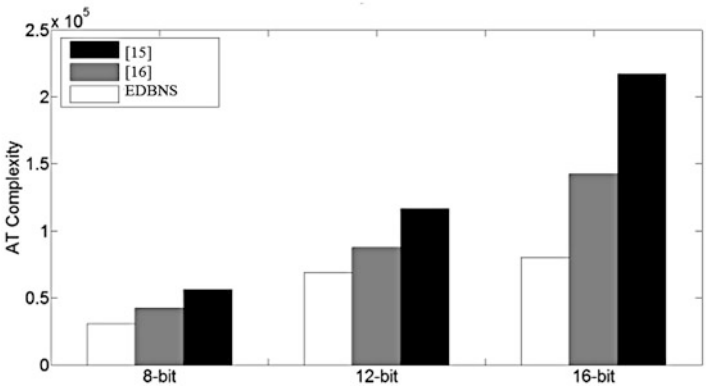

Fig. 11.20 Comparison of AT complexity of 50-tap filters with 8-bit, 12-bit, and 16-bit coefficients designed by [15, 16] and EDBNS

the EDBNS design has a critical path delay of 7.45 ns, which is still considerably faster than the MAC method by 31 % but uses about 19.1 % more logic slices. The cost of programmable shifters could be reduced by further optimization and better mapping. As it stands, the achievable speed of programmable shift-add network makes it a prerogative choice for real-time adaptive system.

11.5 Conclusion

Vector–scalar multiplications with programmable scalars are commonly found in application-specific digital circuits, such as FIR filter. Design methodologies aiming at minimizing their implementation cost have been intensively studied by many researchers. This chapter presents and summarizes the existing approaches to minimize the TMCM block of programmable FIR digital filters based on the canonical CDBNS and its extended form called EDBNS. Search algorithms for the generation of CDBNS and quasi-minimum EDBNS representations are also presented. The obtained CDBNS and EDBNS can be directly mapped to an efficient TMCM architecture consisting of only adders, multiplexers, programmable shifters, and an LUT. Synthesis results show that the designs based on EDBNS can achieve an average logic complexity reduction of up to 47.81 % and critical path delay reduction of up to 14.32 % over the designs obtained by methods based on conventional binary number system.

References

1. R.H. Turner, Functionally diverse programmable logic implementations of digital signal processing algorithms. Ph.D. thesis, Queen's University of Belfast, August 2002
2. T. Hentschel, Channelization for software defined base-stations. J. Ann. Telecommun. **57**(5–6), 386–420 (2002)
3. J.F. Kaiser, *Digital filters: systems analysis by digital computer* (John Wiley and Sons, New York, 1966)
4. J. Chen, C.H. Chang, H. Qian, New power index model for switching power analysis from adder graph of FIR filter, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 2197–2200, Taipei, Taiwan, May 2009
5. D.R. Bull, D.H. Horrocks, Primitive operator digital filters. IEE Proc. G Circuits Devices Syst. **138**(3), 401–412 (1991)
6. M. Potkonjak, M.B. Srivastava, A.P. Chandrakasan, Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination. IEEE Trans. CAD Integr. Circuits Syst. **15**(2), 151–165 (1996)
7. A.G. Dempster, M.D. Macleod, Use of minimum-adder multiplier blocks in FIR digital filters. IEEE Trans. Circuits Syst. II **42**(9), 569–577 (1995)
8. R.I. Hartley, Subexpression sharing in filters using canonic signed digit multipliers. IEEE Trans. Circuits Syst. II **43**(10), 677–688 (1996)
9. Y.C. Lim, S.R. Parker, Discrete coefficient FIR digital filter design based upon an LMS criteria. IEEE Trans. Circuits Syst. **CAS-30**, 723–739 (1983)
10. R. Paško, P. Schaumont, V. Derudder, S. Vernalde, D. Ďuračková, A new algorithm for elimination of common subexpressions. IEEE Trans. CAD Integr. Circuits Syst. **18**(1), 58–68 (1999)
11. C.H. Chang, J. Chen, A.P. Vinod, Information theoretic approach to complexity reduction of FIR filter design. IEEE Trans. Circuits Syst. I **55**(8), 2310–2321 (2008)
12. F. Feng, J. Chen, C.H. Chang, Hypergraph based minimum arborescence algorithm for the optimization and reoptimization of multiple constant multiplications. IEEE Trans. Circuits Syst. I **63**(2), 233–244 (2016)
13. J. Ding, J. Chen, C.H. Chang, A new paradigm of common subexpression elimination by unification of addition and subtraction. IEEE Trans CAD Integr. Circuits Syst. **35**(10), 1605–1617 (2016)
14. J. Chen, J. Tan, C.-H. Chang, F. Feng, A new cost-aware sensitivity-driven algorithm for the design of FIR filters. IEEE Trans Circuits Syst. I **PP**(99), 1–11 (2016)
15. J. Park, W. Jeong, H.M. Meimand, Y. Wang, H. Choo, K. Roy, Computation sharing programmable FIR filter for low-power and high-performance applications. IEEE J. Solid-State Circuits **39**(2), 348–357 (2004)
16. R. Mahesh, A.P. Vinod, Reconfigurable low area complexity filter bank architecture based on frequency response masking for nonuniform channelization in software radio receivers. IEEE Trans. Aerosp. Electron. Syst. **47**(2), 1241–1255 (2011)
17. J. Chen, C.H. Chang, High-level synthesis algorithm for the design of reconfigurable constant multiplier. IEEE Trans. CAD Integr. Circuits Syst. **28**(12), 1844–1856 (2009)
18. J. Chen, C.H. Chang, Design of programmable FIR filters using canonical double based number representation, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1183–1186, Melbourne, Australia, June 2014
19. J. Chen, C.H. Chang, F. Feng, W. Ding, J. Ding, Novel design algorithm for low complexity programmable FIR filters based on extended double base number system. IEEE Trans. Circuits Syst. I **62**(1), 224–233 (2015)
20. R.A. Hawley, B.C. Wong, T.J. Lin, J. Laskowski, H. Samueli, Design techniques for silicon compiler implementations of high-speed FIR digital filters. IEEE J. Solid-State Circuits **31**(5), 656–666 (1996)

21. A.P. Oppenheim, R.W. Schafer, *Discrete-time signal processing* (Prentice-Hall, Englewood Cliffs, NJ, 1989)
22. Z. Ye, R.K. Satzoda, U. Sharma, N. Nazimudeen, C.H. Chang, Performance evaluation of direct form FIR filter with merged arithmetic architecture, in *Proceedings of the 2nd IEEE International Workshop on Electronic Design, Test and Applications*, pp. 407–409, Perth, Australia, Jan 2004
23. F. Xu, C.H. Chang, C.C. Jong, Modified reduced adder graph algorithm for multiplierless FIR filters. Electron. Lett. **41**(6), 302–303 (2005)
24. M.M. Peiro, E.I. Boemo, L. Wanhammar, Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm. IEEE Trans. Circuits Syst. II **49**(3), 196–203 (2002)
25. F. Xu, C.H. Chang, C.C. Jong, Contention resolution algorithms for common subexpression elimination in digital filter design. IEEE Trans. Circuits Syst. II **52**(10), 695–700 (2005)
26. F. Xu, C.H. Chang, C.C. Jong, Design of low-complexity FIR filters based on signed-powers-of-two coefficients with reusable common subexpressions. IEEE Trans. CAD Integr. Circuits Syst. **26**(10), 1898–1907 (2007)
27. P. Tummeltshammer, J. Hoe, M. Püschel, Time-multiplexed multiple-constant multiplication. IEEE Trans. CAD Integr. Circuits Syst. **26**(9), 1551–1563 (2007)
28. L. Aksoy, E. Costa, P. Flores, J. Monteiro, Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. IEEE Trans. CAD Integr. Circuits Syst. **27**(6), 1013–1026 (2008)
29. R. Mahesh, A.P. Vinod, A new common subexpression elimination algorithm for realizing low-complexity higher order digital filters. IEEE Trans. CAD Integr. Circuits Syst. **27**(2), 217–229 (2008)
30. A. Avizienis, Signed-digit representation for fast parallel arithmetic. IRE Trans. Electron. Comput. **10**, 389–400 (1961)
31. S. Arno, F.S. Wheeler, Signed digit representations of minimal hamming weight. IEEE Trans. Comput. **42**(8), 1007–1010 (1993)
32. R.W. Reitwiesner, in Binary arithmetic. *Advances in Computers*, vol. 1. (Academic Press, New York, 1960), pp. 231–308
33. Y.C. Lim, J.B. Evans, B. Liu, Decomposition of binary integers into signed power-of-two terms. IEEE Trans. Circuits Syst. **38**, 667–672 (1991)
34. I.C. Park, H.J. Kang, Digital filter synthesis based on minimal signed digit representation, in *Proceedings of the IEEE Design Automation Conference*, pp. 468–473, Las Vegas, Nevada, June 2001
35. I.C. Park, H.J. Kang, Digital filter synthesis based on an algorithm to generate all minimal signed digit representations. IEEE Trans CAD Integr. Circuits Syst. **21**(12), 1525–1529 (2002)
36. V.S. Dimitrov, G.A. Jullien, R. Muscedere, *Multiple-base number system: theory and applications* (CRC press, Boca Raton, 2012)
37. V.S. Dimitrov, G.A. Jullien, W.C. Liller, Theory and applications of the double-base number system. IEEE Trans. Comput. **48**(10), 1098–1106 (1999)
38. T.N. Shorey, R. Tijdeman, *Exponential diophantine equations* (Cambridge University Press, Cambridge, 1986)

Chapter 12
RNS-Based Public-Key Cryptography (RSA and ECC)

Dimitris Schinianakis and Thanos Stouraitis

12.1 Introduction to Residue Arithmetic

The Residue Number System (RNS) is a data representation system that allows representing an integer number as a set of smaller numbers. RNS is anything but a new idea; it was originally described in terms of a riddle by the Greek mathematician Nicomachus of Gerasa (60–120 CE) in his book "*Introduction to Arithmetic*." In the fifth century, the problem was redescribed by the Chinese Sun Tzu in a book entitled "*Sunzi Suanjing*" (The Mathematical Classic of Sunzi). Sun Tzu described the problem of determining the value of a number X by knowing only the residues after its division with a predefined number of divisors. Sun Tzu devised a methodology for manipulating remainders of an integer after divisions by 3, 5, and 7. This contribution is commemorated today as the Chinese Remainder Theorem (CRT). It was not until the nineteenth century that the theory of residue arithmetic was further explored and new ideas were presented by Carl Friedrich Gauss in his famous "*Disquisitiones Arithmetical*" [41].

Various attempts to explore some of the unique RNS properties were reported from the early 1930s to mid-1950s [41], but perhaps the most remarkable outcome of that period was the research work of Szabo and Tanaka at Lockheed [40]. They worked on a special-purpose correlator, while at the same period a research group from RCA was designing a general-purpose machine. However, these efforts did not

D. Schinianakis (✉)
NOKIA Bell Labs, Werinherstr. 91, Munich, Germany
e-mail: dimitrios.schinianakis@nokia-bell-labs.com

T. Stouraitis
University of Patras, Patras, Greece
KUSTAR, Abu Dhabi, United Arab Emirates

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_12

meet much success at the time, since the technology in 1950s and 1960s could not cope with the unique demands of RNS arithmetic. It is worth mentioning that the research work by Szabo and Tanaka survived only one printing.

Since the mid-1970s, technology and theory started converging gradually and research on the properties and implications of RNS-based designs bloomed in the 1980s and 1990s. However, cryptography was just starting to make its way through the research community and the reasons are quite simple; during the 1980s, requirements for secure communications were not that stringent and technology was struggling to support operations on data of very large word length, which are needed in public-key cryptography [32]. This discouraged researchers from exploring RNS for cryptographic use and it was not until the mid-1990s, when the work of Posch and Posch [29, 30] set the basis for a new research field. Their work, while not explicitly referring to cryptosystem design, proposed a method to embed RNS arithmetic in modular multiplication, which, as we will see later on, is the most time- and area-consuming operation and determines to a large extent the overall cryptosystem performance.

When it comes to modern cryptosystem design, the main RNS application is related to Montgomery modular multiplication (MMM) [26]. The reason is that Montgomery multiplication does not rely on division, which is an inefficient operation in RNS arithmetic. The work by Kawamura et al. in the 2000s is probably a cornerstone for the implementation of competitive RNS-based cryptoprocessors [16, 28]. Since mid-2000s, a considerable amount of research on RNS MMM and efficient cryptosystem designs has been published; the first exploitation of RNS arithmetic in Elliptic Curve Cryptography (ECC) was presented in [35], while RNS has also been applied in RSA cryptography [2, 11, 12, 34]. Lately, studies on the underlying security of RNS Montgomery modular multipliers have also appeared in the literature, contributing significantly to the expansion of RNS-based cryptosystems and their establishment as a viable design option [5, 33].

The ancient arithmetic system is now the basis for a pool of new ideas for performing modular operations on data of large word length. In the following, we analyze the design challenges of RNS arithmetic in RSA and ECC, present design methodologies and important algorithmic advances, and discuss useful and important consequences of RNS cryptosystem-based security.

12.1.1 Mathematical Background

Although a thorough introduction to RNS arithmetic has already been provided in previous chapters, it is important to establish a common mathematical notation followed throughout this chapter.

An RNS base $\mathcal{B}$ comprises a set of L relatively prime base elements $m_1, m_2, \ldots, m_L$ called "*moduli*." Each base is associated with a corresponding range M, given by the product of all moduli, i.e.,

$$M = \prod_{i=1}^{L} m_i. \tag{12.1}$$

Any integer $X \in [0, M-1]$ has a unique RNS representation $\vec{X} = \{x_1, x_2, \ldots, x_L\} = \{\langle X \rangle_{m_1}, \langle X \rangle_{m_2}, \ldots, \langle X \rangle_{m_L}\}$, where $\langle X \rangle_{m_i}$ denotes the operation $X \mod m_i$.

Assuming two integers X, Y, in RNS representation, i.e., $\vec{X} = \{x_1, x_2, \ldots, x_L\}$ and $\vec{Y} = \{y_1, y_2, \ldots, y_L\}$, the operations $\otimes \in (+, -, *)$ can be performed in parallel, as

$$\vec{R} = \vec{X} \otimes \vec{Y} = \{\langle x_1 \otimes y_1 \rangle_{m_1}, \langle x_2 \otimes y_2 \rangle_{m_2}, \ldots, \langle x_L \otimes y_L \rangle_{m_L}\} = \{r_1, r_2, \ldots, r_L\}. \tag{12.2}$$

This equation implies that the ith RNS digit of the result, r_i, is defined only in terms of $\langle x_i \otimes y_i \rangle_{m_i}$, meaning that no information needs to be propagated between residue channels, thus eliminating any delays associated with the carry-related overhead that plagues traditional weighted number systems. The result is high-speed, parallel operations. RNS can represent signed numbers as well; any integer $X \in (-M/2, M/2)$ has an RNS L-tuple representation $\vec{X} = \{x_1, x_2, \ldots, x_L\}$ with $x_i = \langle X \rangle_{m_i}$, if $X > 0$, and $x_i = \langle M - |X| \rangle_{m_i}$, otherwise.

In Fig. 12.1, a general architecture of an RNS processor is depicted. It is apparent that so far our discussion has focused on the RNS core part, but the question that intuitively arises is how to convert an integer to/from RNS representation. In fact, it is exactly these conversions that may define the overall efficiency of an RNS system. Especially for cryptography, the reverse RNS-to-binary conversion is employed implicitly in the form of recursive "*base conversion (BC)*" operations, as we will see in the next sections. As a result, it is vital for the prospect designer to devise efficient

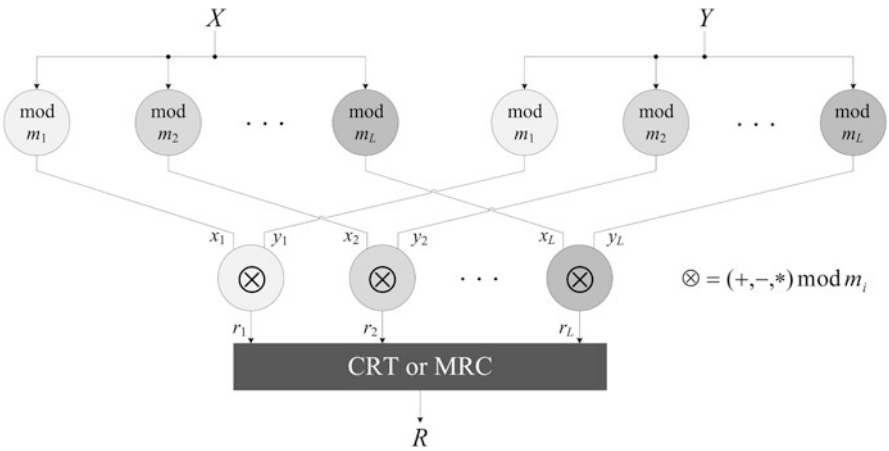

Fig. 12.1 General architecture of an RNS processor

Table 12.1 List of mathematical symbols

Category	Symbol	Explanation
RNS bases	$\mathcal{B}$	An RNS base $\mathcal{B} = \{m_1, m_2, \ldots, m_L\}$
	$\mathcal{B}'$	An RNS base $\mathcal{B}' = \{m'_1, m'_2, \ldots, m'_L\}$
	$\vec{X}$	RNS representation of an integer X in base $\mathcal{B}$
	$\vec{X'}$	RNS representation of an integer X in base $\mathcal{B}'$
	L	Number of moduli
	r	Word length of each modulus
	μ_i	Parameter used in defining moduli of the form $m_i = 2^r - \mu_i$
	h	Word length of parameter μ_i
	m_i	ith modulus of base $\mathcal{B}$; $\forall i \in [1, L]$
	m'_j	jth modulus of base $\mathcal{B}'$; $\forall j \in [1, L]$
	M	$M = \prod_{i=1}^{L} m_i$
	M'	$M' = \prod_{j=1}^{L} m'_j$
	M_i	$M_i = M/m_i$; $\forall i \in [1, L]$
	M'_j	$M'_j = M'/m'_j$; $\forall j \in [1, L]$
	M_i^{-1}	Inverse of M_i modulo m_i; $\forall i \in [1, L]$
	$M_j'^{-1}$	Inverse of M'_j modulo m'_j; $\forall j \in [1, L]$
Montgomery modular multiplication (MMM)	N	Multiplication modulus
	n	Word length of modulus N
	C	Montgomery constant
Operations	$\langle x \rangle_y$	$x \mod y$
	$<<$	Left shift
	$\equiv$	Modulo congruence

conversion strategies. Popular conversion methods are presented in the following sections, while the mathematical notation for this chapter is provided in Table 12.1.

12.1.2 Binary-to-RNS Conversion

The binary-to-RNS conversion is basically a modulo reduction operation $\langle X \rangle_{m_i}$ of the input integer X by all the moduli m_i, $1 \leq i \leq L$ of an RNS base $\mathcal{B}$. The efficiency of conversion is dictated by the form of the moduli. The designer has a lot of options and many moduli sets for efficient modulo reduction have been proposed [4, 39]. Let us review only a few of them:

- The moduli set $\mathcal{B} = \{2^r - 1, 2^r, 2^r + 1\}$. This RNS base is perhaps one of the most popular sets due to the attractive form of the moduli that allows for very easy modulo reduction. Let's see why;

– **Reduction by 2^r**. Let's assume a $2r$-bit integer X with bits $x_{(i)}, 0 < i < 2r-1$, for which we want to calculate $\langle X \rangle_{2^r}$. It holds that

$$\begin{aligned}\langle X \rangle_{2^r} &= \left\langle \sum_{i=0}^{2r-1} x_{(i)} \cdot 2^i \right\rangle_{2^r} = \Big\langle \overbrace{2^{2r-1}x_{(2r-1)} + 2^{2r-2}x_{(2r-2)} + \cdots + 2^r x_{(r)}}^{X_U} + \\ &\quad + \overbrace{2^{r-1}x_{(r-1)} + \cdots + 2^1 x_{(1)} + x_{(0)}}^{X_L} \Big\rangle_{2^r} = \\ &= \langle 2^r \cdot X_U + X_L \rangle_{2^r} = \langle \langle 2^r \cdot X_U \rangle_{2^r} + \langle X_L \rangle_{2^r} \rangle_{2^r} = X_L \end{aligned} \tag{12.3}$$

where X_U, X_L correspond to the upper and lower r-bit words of X, respectively. In other words, the modulo reduction $X \mod 2^r$ amounts to the last r-bits of the integer X.

– **Reduction by $2^r - 1$** By applying the same upper/lower word split like in Eq. (12.3) we get

$$\begin{aligned}\langle X \rangle_{2^r-1} &= \langle 2^r \cdot X_U + X_L \rangle_{2^r-1} = \langle \langle 2^r \cdot X_U \rangle_{2^r-1} + \langle X_L \rangle_{2^r-1} \rangle_{2^r-1} = \\ &= \langle X_U + X_L \rangle_{2^r-1} \end{aligned} \tag{12.4}$$

Thus, the modulo $2^r - 1$ reduction is computed by adding the upper and lower r-bit words of X and then conditionally subtracting $2^r - 1$.

– **Reduction by $2^r + 1$** As with previous cases, it holds that

$$\begin{aligned}\langle X \rangle_{2^r+1} &= \langle 2^r \cdot X_U + X_L \rangle_{2^r+1} = \big\langle \langle 2^r \cdot X_U \rangle_{2^r+1} + \langle X_L \rangle_{2^r+1} \big\rangle_{2^r+1} = \\ &= \langle X_L - X_U \rangle_{2^r+1} \end{aligned} \tag{12.5}$$

In other words, the modulo $2^r + 1$ reduction is computed by subtracting the upper from the lower r-bit word of X and then conditionally adding $2^r + 1$.

Apparently, the two last operations can be easily mechanized via end-around adders, where the output carry determines the condition for addition/subtraction of quantities $2^r - 1$ and $2^r + 1$, respectively. Regarding cryptography, this set is unfortunately not easily exploitable. The problem is that the numbers involved in public-key cryptography are large integers (greater than 192 bits for ECC or 2048 bits for RSA, for secure implementations). As will be explained later, the word length of the corresponding RNS bases should be 2–4 times larger than the word length n of the cryptographic modulus N (see Table 12.1). This simply means that if the 3-modulus set is used, each channel should have a large word length thus, rendering the parallelization benefits of RNS negligible.

- Moduli of the form $m_i = 2^r - \mu_i$. In this moduli set, each modulus m_i is determined only by the corresponding parameter μ_i. This proves useful when it comes to the memory storage requirements of an RNS-based system. Moreover, μ_is are selected to be small integers so that $\mu_i \ll 2^r$ (we assume a word length

h typically 10 bits for a 1024-bit RSA implementation) [16, 28]. Let's analyze how reduction is performed in this case assuming that X is of word length z with $z > 2r$. We split integer X in 3 parts D, E, and F, where D corresponds to the least significant r-bit word, E to the next r-bit word, and F to the most significant $(z-2r)$-bit word. It holds that

$$\begin{aligned}
\langle X\rangle_{m_i} &= \left\langle \overbrace{\sum_{i=0}^{r-1} x_{(i)}2^i}^{D} + 2^r \overbrace{\sum_{i=0}^{r-1} x_{(r+i)}2^{r+i}}^{E} + 2^{2r} \overbrace{\sum_{i=0}^{z-2r-1} x_{(2r+i)}2^{2r+i}}^{F} \right\rangle_{m_i} = \\
&= \left\langle \sum_{i=0}^{r-1} x_{(i)}2^i + \overbrace{\mu_i \sum_{i=0}^{r-1} x_{(r+i)}2^{r+i}}^{\delta} + \overbrace{\mu_i^2 \sum_{i=0}^{z-2r-1} x_{(2r+i)}2^{2r+i}}^{\epsilon} \right\rangle_{m_i} = \\
&= \left\langle \overbrace{\sum_{i=0}^{r-1} x_{(i)}2^i + \sum_{i=0}^{r+h-1} \delta_{(i)}2^i + \sum_{i=0}^{z+2(h-r)-1} \epsilon_{(i)}2^i}^{\zeta} \right\rangle_{m_i} = \left\langle \sum_{i=0}^{r+h} \zeta_{(i)}2^i \right\rangle_{m_i} = \\
&= \left\langle \sum_{i=0}^{r-1} \zeta_{(i)}2^i + 2^r \sum_{i=0}^{h} \zeta_{(r+i)}2^{r+i} \right\rangle_{m_i} = \\
&= \left\langle \overbrace{\sum_{i=0}^{r-1} \zeta_{(i)}2^i + \mu_i \sum_{i=0}^{h} \zeta_{(r+i)}2^{r+i}}^{\kappa} \right\rangle_{m_i} = \left\langle \sum_{i=0}^{r} \kappa_{(i)}2^i \right\rangle_{m_i} = \\
&= \left\langle \sum_{i=0}^{r-1} \kappa_{(i)}2^i + 2^r\kappa_{(r)} \right\rangle_{m_i} = \sum_{i=0}^{r-1} \kappa_{(i)}2^i + \mu_i\kappa_{(r)}
\end{aligned} \tag{12.6}$$

Following the series of Eq. (12.6), we can easily devise a corresponding circuit, as shown in Fig. 12.2. *The reader is urged to verify the equations above and match them with the architecture in Fig. 12.2, as they are instructive on how to perform simplifications for efficient modulo reductions.*

12.1.3 RNS-to-Binary Conversion

The reverse process of converting an integer from RNS to binary representation is much more complex to realize compared to the forward conversion analyzed before. In fact, its efficiency determines the efficiency of the overall RNS processor to a

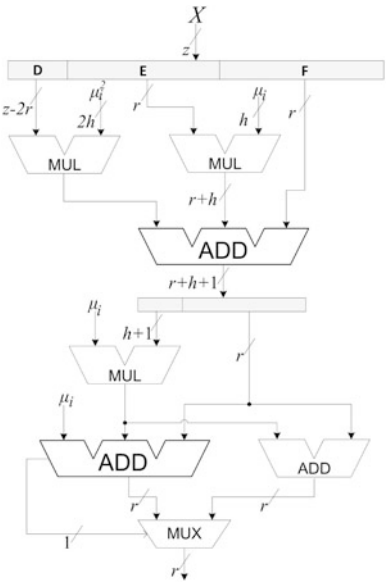

Fig. 12.2 Modulo reduction by moduli of the form $2^r - \mu_i$

great extent [35]. To reconstruct an integer from its residues, two methods may be generally employed, namely the Chinese Remainder Theorem (CRT) and the Mixed-Radix Conversion (MRC) [41].

12.1.3.1 The Chinese Remainder Theorem

CRT is based on the equation

$$X = \left\langle \sum_{i=1}^{L} \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i \right\rangle_M , \tag{12.7}$$

where M_i^{-1} is the inverse of M_i modulo m_i, i.e., a quantity such that $M_i^{-1} \cdot M_i \equiv 1 \mod m_i$. An observation on (12.7) reveals the main characteristics and implications of CRT realization. CRT is a modulo multiply accumulate procedure decomposed in:

- L inner modular products of small r-bit quantities to formulate $\left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i}$
- L non-modular multiplications to formulate $\left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i$
- addition of the previous results modulo M.

It is apparent that CRT suffers not only from the large multiplications in the second step (note that M_is are $r(L-1)$-bit long) but also from the large modulo-M addition in the final step. The addition of the inner products can be achieved in $O(\log L)$ time using standard addition techniques.

Important efforts for reducing the complexity of this scheme have been carried out during the last 20 years [16, 28, 37, 38, 43, 44]. The methods target both the reduction of the complexity of inner product calculation and the substitution of the final modulo-M addition by simpler operations.

The works in [16, 37] are based on the observation that the result in (12.7) is congruent modulo M. In order to obtain the exact value of X, we must compute

$$X = \sum_{i=1}^{L} \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i - \gamma M, \tag{12.8}$$

where γ is an integer correction factor [16, 37]. In other words, the large modulo-M addition is replaced by a subtraction and a multiplication. Efficient calculation of the correction factor γ is critical. Shenoy and Kumaresan [37] have developed an algorithm, which requires a redundant modulus $m_{L+1} \geq L$, so that the RNS base $\mathcal{B}$ is extended to $\mathcal{B} = \{m_1, m_2, \ldots, m_L \mid\mid m_{L+1}\}$. This adds an extra channel of calculations. Let X be an integer with an RNS representation $\vec{X} = \{x_1, x_2, \ldots, x_L \mid\mid x_{L+1}\}$, where $x_{L+1} = \langle X \rangle_{m_{L+1}}$. By reducing both sides of (12.8) mod m_{L+1}, we obtain

$$\begin{aligned} \langle X \rangle_{m_{L+1}} &= \left\langle \left\langle \sum_{i=1}^{L} \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i \right\rangle_{m_{L+1}} - \langle \gamma M \rangle_{m_{L+1}} \right\rangle_{m_{L+1}} \Rightarrow \\ \langle \gamma \rangle_{m_{L+1}} &= \left\langle \left\langle M^{-1} \right\rangle_{m_{L+1}} \left(\left\langle \sum_{i=1}^{L} \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i \right\rangle_{m_{L+1}} - x_{L+1} \right) \right\rangle_{m_{L+1}} . \end{aligned} \tag{12.9}$$

Since $\gamma < L$ and $m_{L+1} \geq L$, it follows that $\gamma = \langle \gamma \rangle_{m_{L+1}}$ [37]. As all terms on the right-hand side of (12.9) are known, the correction factor γ can be substituted in (12.8) to obtain X.

Kawamura et al. [16] employed a different approach for the γ calculation. Starting again from (12.8) and substituting

$$\xi_i = \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i}, \tag{12.10}$$

we obtain

$$X = \sum_{i=1}^{L} \xi_i \cdot M_i - \gamma M. \tag{12.11}$$

Dividing both sides by M, we obtain

$$\sum_{i=1}^{L} \frac{\xi_i}{m_i} = \frac{X}{M} + \gamma. \tag{12.12}$$

Since $0 \leq X/M < 1$, it holds that $\gamma \leq \sum_{i=1}^{L} \frac{\xi_i}{m_i} < \gamma + 1$. Therefore,

$$\gamma = \left\lfloor \sum_{i=1}^{L} \frac{\xi_i}{m_i} \right\rfloor \tag{12.13}$$

with $0 \leq \gamma < L$, since $0 \leq \xi_i/m_i < 1$. Two approximations were employed to avoid hard divisions in (12.13). The denominator m_i is replaced by 2^r, where $2^{r-1} < m_i \leq 2^r$, while the numerator ξ_i is approximated by its most significant q bits, where $q < r$. Thus, instead of γ, an approximated value γ^* can be calculated by

$$\gamma^* = \left\lfloor \sum_{i=1}^{L} \frac{\text{trunc}(\xi_i)}{2^r} + \alpha \right\rfloor, \tag{12.14}$$

where $\text{trunc}(\xi_i) = \xi_i \wedge \overbrace{(1\ldots1)}^{q}\overbrace{(0\ldots0)}^{(r-q)}$ and $\wedge$ denotes an AND operation. An offset value $0 \leq \alpha < 1$ is introduced to compensate the error produced by the approximations. Since division by powers of 2 amounts to simple shifts, (12.14) can be realized by additions alone. The offset value α can be determined, so that the error issued by the approximations is zero [16].

12.1.3.2 Mixed-Radix Conversion

Another popular method for RNS-to-binary conversion is through the Mixed-Radix Conversion (MRC) Algorithm [19]. The MRC of an integer X with an RNS representation $\vec{X} = \{x_1, x_2, \ldots, x_L\}$ is

$$X = W_1 U_1 + W_2 U_2 + \cdots + W_L U_L, \tag{12.15}$$

where $W_i = \prod_{j=1}^{i-1} m_j, \forall i \in [2, L]$ and $W_1 = 1$; The mixed-radix digits $U_1, U_2, \ldots, U_L$ are referred as the Mixed-Radix System (MRS) representation of X and can be computed as

$$\begin{aligned} U_1 &= x_1 \\ U_2 &= \left\langle (x_2 - U_1)\, m_{1,2}^{-1} \right\rangle_{m_2} \end{aligned}$$

$$
\begin{aligned}
U_3 &= \left\langle \left((x_3 - U_1)\, m_{1,3}^{-1} - U_2 \right) m_{2,3}^{-1} \right\rangle_{m_3} \\
&\vdots \\
U_L &= \left\langle \left(\ldots (x_L - U_1)\, m_{1,L}^{-1} - \cdots - U_{L-1} \right) m_{L-1,L}^{-1} \right\rangle_{m_L},
\end{aligned}
\tag{12.16}
$$

where $m_i m_{i,j}^{-1} \equiv 1 \mod m_j$. Equation (12.16) requires $L\frac{L-1}{2}$ modular multiplications. Another version of MRC, which simplifies (12.16) and reduces the total number of modular multiplications to only $L-2$, is based on

$$
\begin{aligned}
U_1 &= x_1 \\
U_2 &= \langle x_2 - x_1 \rangle_{m_2} \\
U_3 &= \langle x_3 - x_1 - W_2 U_2 \rangle_{m_3} \\
&\vdots \\
U_L &= \langle x_L - x_1 - W_2 U_2 - W_3 U_3 - \cdots - W_{L-1} U_{L-1} \rangle_{m_L},
\end{aligned}
\tag{12.17}
$$

provided that the predetermined factors $V_1 \equiv 1$ and $V_i \equiv \left\langle \left(\prod_{j=1}^{i-1} m_j \right)^{-1} \right\rangle_{m_i} = 1,\ \forall i \in [2, L]$ [45].

The main characteristic of MRC is its sequential nature. As any term U_i can't be calculated before U_{i-1} is available, the delay of the scheme is $O(L)$. In practical cryptosystem implementations, both (12.16) and (12.17) have been employed since they avoid the large $\mod M$ reduction of (12.7) of the CRT [3, 11, 12, 16]. In [34], a matrix-based decomposition of operations in (12.17) was presented to further simplify the conversion process.

12.1.3.3 Base Conversion

Things get a bit more complicated when it comes to base conversion (BC). As base conversion we refer to the operation of converting the RNS representation of an integer from one RNS base $\mathcal{B}$ to another RNS base $\mathcal{B}'$. In literature the term "*base extension*" is commonly used, but we avoid it since base extension intuitively refers to adding more moduli on a given RNS base, extending, thus, the range of calculations. A base conversion is something similar, but in cryptography we practically refer to two different bases, and we use it to overcome some mathematical problems that arise from using RNS in modular multiplication, as we'll see later on.

We can formulate the transformation

$$
\vec{X} = \{x_1, x_2, \ldots, x_L\} \xrightarrow{\text{BC}} \vec{X'} = \{x'_1, x'_2, \ldots, x'_L\}
\tag{12.18}
$$

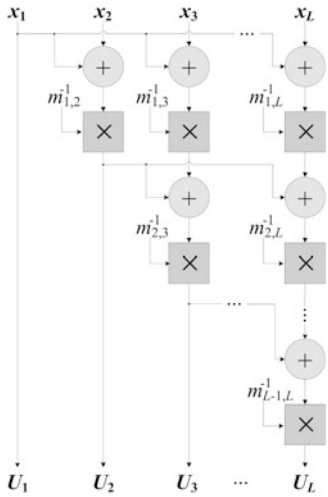

Fig. 12.3 The Mixed Radix Conversion process

such that

$$X \mod M \equiv (X' \mod M') \mod M. \tag{12.19}$$

In other words, the integers X, X' should be congruent modulo M. It is not necessary that the 2 bases have the same number of moduli. However, the moduli should be relatively co-prime, as with any RNS base. In practical implementations, employing the same number of moduli for both bases allows for simpler architectures, as we will analyze in the following paragraphs (Fig. 12.3).

Kawamura et al. Base Conversion

Out of the many options for base conversion [3, 12, 16, 37] let us present the first practical implementation of a fully RNS-based architecture presented in 2000 by Kawamura et al. [16].

They employed the approximation method in Sect. 12.1.3.1 and the algorithm cleverly combines the calculation of the correction factor γ in Eqs. (12.10)–(12.14) with the modulo reduction of the result by the moduli of the new RNS base. The corresponding base conversion algorithm is illustrated below as Algorithm 12.1 (we assume that the two bases have the same number of moduli, L). In step 3, the inner products of the CRT are calculated and then the trick with the approximation of the γ factor is realized. In step 15 the results of the inner loops in steps 6–13 are computed modulo m_i, i.e., the moduli of the new base. *We encourage the reader to match the steps of Algorithm 12.1 with the corresponding Eqs.* (12.10)–(12.14) *and the operations decomposition in Fig. 12.4 to get a better understanding of the operations involved.*

Algorithm 12.1 Base conversion by Kawamura et al. [16]

Input: $\overrightarrow{X'} = \{x'_1, x'_2, \ldots, x'_L\}, \mathcal{B}, \mathcal{B}', \alpha$
Output: $\overrightarrow{X} = \{x_1, x_2, \ldots, x_L\}$
Precompute: $\left\langle M_j'^{-1} \right\rangle_{m'_j}, \left\langle M'_j \right\rangle_{m_i}, \langle -M' \rangle_{m_i} \; (\forall i, j = 1 \ldots L)$

$\sigma_0 = \alpha$
for all $j = 1 \ldots L$ **do**
  $\xi'_j = \left\langle x'_j \cdot M_j'^{-1} \right\rangle_{m'_j}$
  $s_{j,0} = 0$
end for
for all $i = 1 \ldots L$ **do**
  for $j = 1 \ldots L$ **do**
    $\sigma_j = \sigma_{(j-1)} + \text{trunc}(\xi'_j)/2^r$
    $\gamma_j^* = \lfloor \sigma_j \rfloor, \{\gamma_j^* = \{0, 1\}\}$
    $\sigma_j = \sigma_j - \gamma_j^*$
    $s_{i,j} = s_{i,(j-1)} + \xi'_j \cdot \left\langle M'_j \right\rangle_{m_i} + \gamma_j^* \cdot \langle -M' \rangle_{m_i}$
  end for
end for
for all $i = 1 \ldots L$ **do**
  $x_i = \left\langle s_{i,L} \right\rangle_{m_i}$
end for

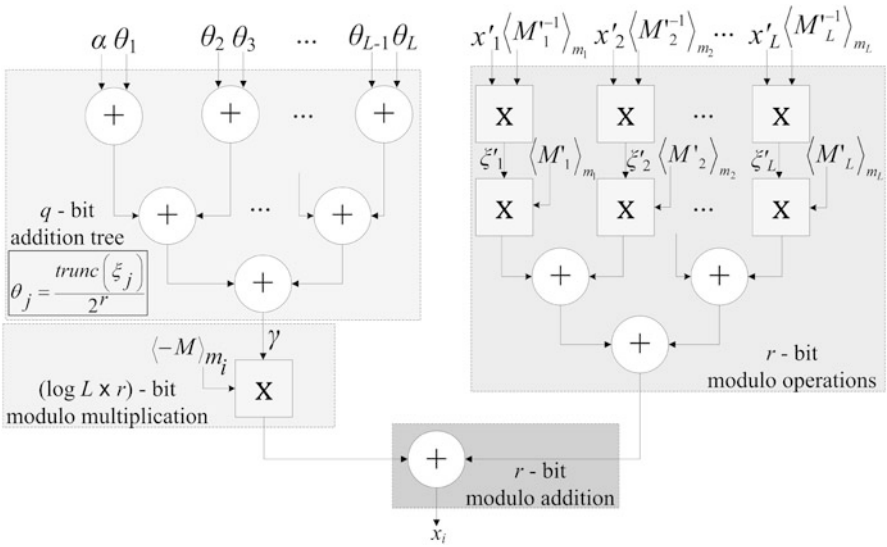

Fig. 12.4 "Kawamura et al." approximated base conversion

12.2 RSA Public-Key Cryptography

The RSA public-key cryptosystem was invented at MIT in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman (RSA stands for the initials of the inventors). The *public key* in this cryptosystem consists of the value N, which is called the modulus, and the value e, which is called the public exponent. The *private key* consists of the modulus N and the value d, which is called the private exponent. This asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the so-called factoring problem. In general, any public-key cryptosystem relies on three vital operations: key establishment, encryption/decryption, and digital signature. Let's see how RSA addresses these operations.

12.2.1 Key Establishment

Key establishment is any process by which a pair of public/secret keys are exchanged between two parties safely, in order to allow the afterward use of a cryptographic algorithm. The steps for RSA key exchange are:

1. Generate randomly a pair of large prime numbers p, q
2. Calculate their product $N = pq$. N is called the public modulus
3. Compute $\varphi(N) = \varphi(p)\varphi(q) = (p-1)(q-1) = N - (p+q-1)$, where φ is Euler's totient function. This value is kept private.
4. Choose an integer e, such that $1 < e < \varphi(N)$ and $\gcd(e, \varphi(N)) = 1$; i.e., e and $\varphi(N)$ are co-prime.
5. Calculate d as $d \equiv e^{-1} \mod \varphi(N)$; i.e., d is the modular multiplicative inverse of $e \mod \varphi(N)$. d is kept secret
6. The pair (N, e) is the public key and the pair (N, d) is the private key.

12.2.2 Encryption/Decryption

The encryption operation of a message m in RSA is basically a modular exponentiation according to

$$c = \text{ENCRYPT}(m) = m^e \mod N \tag{12.20}$$

The output c is referred to as the *ciphertext*. Decryption is the reverse operation, i.e.,

$$m = \text{DECRYPT}(c) = c^d \mod N \tag{12.21}$$

The reader is urged to perform the calculations and prove why Eq. (12.21) *calculates the initial message* (*hint*: check the property in step 5 of key establishment).

12.2.3 Digital Signatures

A digital signature provides the means to prove the authenticity of a digital message or document. A valid digital signature offers sufficient reasons to believe that the message has been generated by a valid sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered by a third-party during transmission (integrity). A digital signature scheme is realized, if we use encryption/decryption in the reverse order, i.e.,

$$s = \mathrm{SIGN}(m) = m^d \mod N \tag{12.22}$$

where s is the signature sent to the other party. Recipient, then, uses the public modulus N and the signature s to compute

$$m = \mathrm{VERIFY}(s) = s^e \mod N. \tag{12.23}$$

Recipient can then compare the message computed from the verification process in (12.23) and the one from the decryption in (12.21). If the two messages match, then the sender and the message are validated. In practice, the message m is not the original one but a function of it, for example, via a one-way hash function [23].

12.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC), presented by Koblitz [18] and Miller [24] independently in 1985, has withstood a large number of attacks and has evolved significantly, so that it is considered nowadays a mature public-key cryptosystem. Extensive research work is currently focusing on the underlying mathematics, security, and its efficient implementations.

By exploiting the Elliptic Curve Discrete Logarithm Problem (ECDLP), ECC offers the highest strength per bit and the smallest key size, when compared to other public-key cryptosystems. ECDLP states that given two points P, Q, on an elliptic curve, such that $Q = [k]P$, it is computationally infeasible to calculate $[k]$ [6].

Although elliptic curves can be defined on a variety of fields, only finite fields are employed in cryptography. Among them, prime Galois fields, GF(p), and binary extension fields, GF(2^n), are considered to be the ones that offer the most efficient and secure implementations [6].

12.3.1 Elliptic Curves over GF(p)

An elliptic curve $\mathcal{E}$ over GF(p) is defined by an equation of the form

$$y^2 = x^3 + ax + b \mod p, \tag{12.24}$$

where $a, b \in \mathrm{GF}(p)$ and $4a^3 + 27b^2 \neq 0 \pmod{p}$, together with a special point $\mathcal{O}$, called the *point at infinity*. The set $\mathcal{E}(\mathrm{GF}(p))$ consists of all points $(x, y), x, y \in \mathrm{GF}(p)$, that satisfy (12.24) together with $\mathcal{O}$. Addition of two points on an elliptic curve can be defined by the group law. Together with the addition operation, the set of points $\mathcal{E}(\mathrm{GF}(p))$ forms a group, with $\mathcal{O}$ serving as its identity element. This is the group that is used in the construction of elliptic curve cryptosystems. The special case of adding a point to itself is called a point doubling.

Examples of point addition and point doubling are depicted in Fig. 12.5. The double of a point P_0 is obtained by taking the tangent line on P_0 until a second intersection point on the curve is found (there is always a second point due to the form of (12.24)). The mirror point of this second intersection with respect to the x-axis is $2P_0$. Similarly, to add two points P_0, P_1, a third intersecting point is found by the line that connects P_0, P_1. The mirror point of the third intersection point is $P_2 = P_0 + P_1$.

Let $P_0 = (x_0, y_0), P_1 = (x_1, y_1) \neq \mathcal{O}$, and $P_0 \neq -P_1$. The coordinates of their sum, $P_2(x_2, y_2)$ are

$$P_2 = P_0 + P_1 = \begin{cases} x_2 = \lambda^2 - x_0 - x_1 \\ y_2 = (x_0 - x_2)\lambda - y_0, \end{cases} \tag{12.25}$$

where $\lambda = \frac{y_1 - y_0}{x_1 - x_0}$. The double of a point is given by

$$P_2 = 2P_0 = \begin{cases} x_2 = \lambda^2 - 2x_0 \\ y_2 = (x_0 - x_2)\lambda - y_0, \end{cases} \tag{12.26}$$

where $\lambda = \frac{3x_0^2 + a}{2y_0}$.

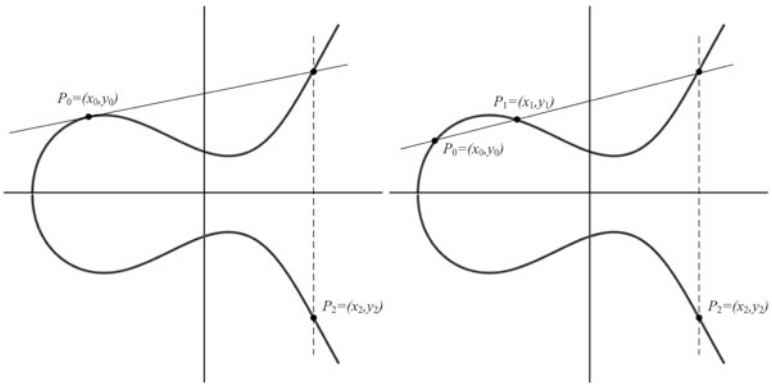

Fig. 12.5 Operations on elliptic curves. (**a**) Point doubling. (**b**) Point addition

From (12.25), (12.26), it is apparent that in order to perform an addition or a doubling of a point in *affine* representation, one needs to compute the inverse of an element in GF(p), which is a time-consuming operation [6]. In order to avoid inversions, the use of *projective coordinates* has been proposed [6]. Given a point $P = (x, y)$ in affine coordinates, the projective coordinates $P = (X, Y, Z)$ are given by

$$X = x;\ Y = y;\ Z = 1. \tag{12.27}$$

There are various projective coordinate representations that lead to more efficient implementations than using the one in (12.27). Jacobian coordinates are an example of such a representation [6]. Using Jacobian coordinates, the affine representation of an EC point is given by

$$x = \frac{X}{Z^2};\ y = \frac{Y}{Z^3}. \tag{12.28}$$

while the point at infinity is given by $\mathcal{O} = (0, 0, 1)$.

Using the representation in (12.28), (12.24) rewrites to

$$\mathcal{E}\,(\mathrm{GF}(p)):\ Y^2 = X^3 + aXZ^4 + bZ^6. \tag{12.29}$$

Let $P_0 = (X_0, Y_0, Z_0)$, $P_1 = (X_1, Y_1, Z_1) \in \mathcal{E}(\mathrm{GF}(p))$. The sum $P_2 = (X_2, Y_2, Z_2) = P_0 + P_1 \in \mathcal{E}(\mathrm{GF}(p))$ can be computed as follows.

If $P_0 = P_1$, then

$$P_2 = 2P_1 = \begin{cases} X_2 = M^2 - 2S \\ Y_2 = M(S - X_2) - T, \\ Z_2 = 2Y_1Z_1 \end{cases} \tag{12.30}$$

where $M = 3X_1^2 + aZ_1^4, S = 4X_1Y_1^2$ and $T = 8Y_1^4$. On the other hand, if $P_0 \neq P_1$, then

$$P_2 = P_0 + P_1 = \begin{cases} X_2 = R^2 - TW^2 \\ 2Y_2 = VR - MW^3, \\ Z_2 = Z_0Z_1W \end{cases} \tag{12.31}$$

where $R = Y_0Z_1^3 - Y_1Z_0^3, T = X_0Z_1^2 + X_1Z_0^2, W = X_0Z_1^2 - X_1Z_0^2,\ M = Y_0Z_1^3 + Y_1Z_0^3$, and $V = TW^2 - 2X_2$.

Elliptic curves over GF(2^n) are defined in a similar manner; it is, however, out of scope of this chapter to further emphasize [6].

12.3.2 Point Multiplication

Having the operations of point doubling and point addition available, the next step is to implement the scalar point multiplication, which is the most important operation in ECC. Among various options [6], the *binary method algorithm* is frequently chosen, because it is easy to implement and minimizes also memory requirements. The binary method algorithm [6] is based on the binary expansion of the scalar $[k]$, as follows.

Algorithm 12.2 Binary method for EC point multiplication

Input: A point P, an n-bit integer $k = \sum_{i=0}^{n-1} k_{(i)} 2^i$

Output: $Q = [k]P$

```
Q ⟵ O
for i = n − 1 to 0 do
    Q ⟵ [2]Q
    if k_(i) = 1 then
        Q ⟵ Q + P
    end if
end for
return Q
```

The binary method requires $n - 1$ point doublings and $W - 1$ point additions, where n is the length and W the Hamming weight of the binary expansion of k. For any positive integer k, the notation $[k]$ is used to denote the *multiplication-by-k* map from the curve to itself. The notation $[k]$ is extended to $k \leq 0$ by defining $[0]P = \mathcal{O}$, and $[-k]P = -([k]P)$. Other methods based on various representations for the scalar $[k]$ include window-based algorithms, signed-digit representations, NAF representations, etc. [6, 23].

What is the key message from the discussion so far? It is clear that consecutive modular multiplications constitute the heart of the cryptosystem in both RSA and ECC. Any design strategy that wishes to adequately address these cryptosystems should target an efficient modular multiplication. In the following section, we present the main ideas behind RNS modular multiplication and its applications to both cryptosystems.

12.4 RNS Application to Public-Key Cryptography

12.4.1 RNS Modular Multiplication

Mathematician P.L. Montgomery presented an ingenious method in 1985 for efficient modular multiplication without trial divisions [26]. This 3-page paper is perhaps the most cited work in the field during the last 30 years and changed radically the available implementation options for efficient cryptosystem design. A modification of Montgomery's modular multiplication (MMM) algorithm was proposed 10 years later by Karl C. Posch and Reinhard Posch in order to accommodate data in RNS format [30]. Later, researchers produced more robust algorithms and implementations, mainly for use in the context of modular exponentiation for RSA [2, 11, 16]. Let us rewrite here the original MMM for convenience.

C is a constant known as the Montgomery radix. Condition $\gcd(C, N) = 1$ ensures the existence of $C^{-1} \mod N$. Condition $N < C$ is sufficient for $W < 2N$ since

$$W = \frac{AB + TN}{C} < \frac{N^2 + CN}{C} = \left(\frac{N}{C} + 1\right) N < 2N. \tag{12.32}$$

Since $WC = AB + TN$, $WC \equiv AB \mod N$ holds. Multiplying with $C^{-1} \mod N$ on both sides of (12.32) we obtain $W \equiv ABC^{-1} \mod N$.

The algorithm requires first to transform the input operands to their corresponding Montgomery representations [26]. Assuming an integer A, its Montgomery representation is defined as $\bar{A} = AC \mod N$. This conversion may be realized by means of an extra Montgomery multiplication by $C^2 \mod N$, i.e., $\bar{A} = A \times (C^2 \mod N) \times C^{-1} \mod N = AC \mod N$. With these inputs the algorithm outputs the Montgomery residue of the result, i.e., $\bar{W} = WC \mod N = ABC \mod N$. An extra Montgomery multiplication with inputs the result $\bar{W} = WC \mod N$ and 1 mod N eliminates the Montgomery radix and produces the exact result $WC \times 1 \times C^{-1} \mod N = W \mod N$.

The challenges to transform this algorithm to RNS format are steps 2 and 5. Step 2 is a modulo C operation. In non-RNS implementations, C is usually chosen to be a power of 2, thus modulo C operations amount to simple shifts. A method

Algorithm 12.3 Montgomery modular multiplication [26]

Input: A, B, N, C, C^{-1} $\{ A, B < N \}$
Output: $W \equiv ABC^{-1} \mod N$, $\{ W < 2N \}$
1: $S \leftarrow A \cdot B$
2: $T \leftarrow S \cdot \left(-N^{-1}\right) \mod C$
3: $U \leftarrow T \cdot N$
4: $V \leftarrow S + U$
5: $W \leftarrow V/C$

Algorithm 12.4 RNS Montgomery modular multiplication

Input: $\vec{A}, \vec{A'}, \vec{B}, \vec{B'}$ { $A, B < 2N$ }
Output: $\vec{W}, \vec{W'}$, { $W < 2N$ and $W \equiv ABM'^{-1} \mod N$ }
Precompute: $\overrightarrow{(-N^{-1})'}, \overrightarrow{(M'^{-1})}, \vec{N}$
1: $(\vec{S}, \vec{S'}) \leftarrow (\vec{A} \cdot \vec{B}, \vec{A'} \cdot \vec{B'})$
2: $\vec{T'} \leftarrow \vec{S'} \cdot \overrightarrow{(-N^{-1})'}$
3: $\vec{T} \leftarrow \vec{T'}$ { base conversion step }
4: $\vec{U} \leftarrow \vec{T} \cdot \vec{N}$
5: $\vec{V} \leftarrow \vec{S} + \vec{U}$
6: $\vec{W} \leftarrow \vec{V} \cdot \overrightarrow{(M'^{-1})}$
7: $\vec{W'} \leftarrow \vec{W}$ { base conversion step}

that provides the modulo operation of step 2 within RNS representation needs to be devised for the RNS version of the algorithm. Similar problems arise for step 5, where division by the Montgomery radix needs to be performed in RNS.

The trick to overcome these issues is to choose a new Montgomery radix. Assume that a base $\mathcal{B}' = \{m'_1, m'_2, \ldots, m'_L\}$ is employed with a corresponding range M'. By assigning the Montgomery radix to be the range M' itself, step 2 is transformed to a step computed modulo M', since RNS is a closed modulo M' system. Unfortunately, computations in the same base cannot be continued, since in step 5 a quantity of the form $M'^{-1} \mod M'$ needs to be computed, which mathematically does not exist. For this reason, another base $\mathcal{B} = \{m_1, m_2, \ldots, m_L\}$ is employed and a base conversion from base $\mathcal{B}'$ to base $\mathcal{B}$ is performed after step 2 of the algorithm.

Steps 3, 4, and 5 are then computed in the new base $\mathcal{B}$, since it is now feasible to compute $M'^{-1} \mod M$ in step 5. The complete RNS Montgomery Modular Multiplication (RNSMMM) algorithm is shown below as Algorithm 12.4. An extra BC is issued at the end of the algorithm so that inputs and outputs are compatible with each other and the algorithm can be used repeatedly in the context of any modular exponentiation algorithm. This applies both to RSA and ECC applications, where consecutive modular multiplications constitute the heart of the cryptosystems. For the same reason the input limits are modified from $A, B < N$ in the original MMM to $A, B < 2N$ in RNSMMM. Note that step 1 is performed in both bases and the complexity of the algorithm depends clearly on the complexity of the BC steps.

The validity of the algorithm is assured, if $\gcd(N, M') = 1$ and $\gcd(M, M') = 1$, so that the quantities $-N^{-1} \mod M'$ (step 2) and $M'^{-1} \mod M$ (step 6) exist, respectively. The conditions for the ranges can also be derived. Considering that

$$W = \frac{AB + TN}{M'} < \frac{(2N)^2 + NM'}{M'} = \left(\frac{4N}{M'} + 1\right) N \leq 2N. \tag{12.33}$$

it follows that $4N \leq M'$. This equation also shows that $2N \leq M$ is sufficient for $W < M$ and $V < M \cdot M'$, since V is the maximum intermediate value and all values

Table 12.2 Number of modular multiplications in RNSMMM [16]

Steps in RNSMMM	[16]
1, 2, 4, 6	$5L$
First BC	$L^2 + 2L$
Second BC	$L^2 + 2L$
Total	$2L^2 + 9L$

are less than $M \cdot M'$. Table 12.2 summarizes the complexity of RNSMMM in terms of small r-bit modular multiplications, assuming that the BC operation is based on Kawamura et al. method in Algorithm 12.1. *The reader is advised to validate the values in the table*.

In the remaining, two strategies will be considered for base conversion. The first one is the BC proposed by Kawamura et al. and will be applied in the context of RSA modulo exponentiation (see Sect. 12.1.3.3). The second strategy is based on the MRC and will be applied in the context of an ECC processor. Other algorithms based on Kawamura's approach are the ones proposed by Bajard and Imbert [2] and Gandino et al. who offered optimizations by reducing the total number of steps required using pre-computations [11]. Prospective readers who require a deeper insight on the latest advances of RNS application in modulo multiplication are encouraged to study these works.

12.4.2 RNS Application in RSA

As described in Sect. 12.2, the core operation in RSA is modular exponentiation. In fact, any modular exponentiation algorithm equipped with the operation of RNSMMM could realize RSA operations via consecutive modulo multiplications. Algorithm 12.5 presents such an option. It is based on a straightforward binary expansion of the exponent e to mechanize the ENCRYPT operation $y = m^e \bmod N$. The only additional steps are the input and output conversions to/from RNS Montgomery domain in steps 1 and 9, respectively. Each RNSMMM accepts as inputs the RNS representations of input operands in both bases.

12.4.2.1 Conversions to/from RNS Representation

Kawamura et al. offered an efficient method for conversions to/from RNS representation [16]. The key idea was to employ high-radix representations so that each high-radix digit can be assigned to an RNS channel. In that way, handling of very large integers can be achieved. A radix-2^r representation of an integer X as an L-tuple $\left(x^{(L-1)}, \ldots, x^{(0)}\right)$ satisfies

Algorithm 12.5 RNS modulo exponentiation

Input: $\vec{m}, \vec{m'}, e$ { e is an n-bit integer, i.e., $e = (e_{(n-1)}e_{(n-2)} \ldots e_{(1)}e_{(0)})_2$}
Output: $\vec{y}, \vec{y'}$, { $y \equiv m^e \mod N$}
Precompute: $\overrightarrow{(C^2 \mod N)}, \overrightarrow{(C^2 \mod N)'}$
1: $(\vec{M}, \vec{M'}) \leftarrow RNSMMM(\vec{m}, \vec{m'}, \overrightarrow{(C^2 \mod N)}, \overrightarrow{(C^2 \mod N)'})$
2: $(\vec{y}, \vec{y'}) \leftarrow (\vec{M}, \vec{M'})$
3: **for** $i = n-2$ to 0 **do**
4: $(\vec{y}, \vec{y'}) \leftarrow RNSMMM(\vec{y}, \vec{y'}, \vec{y}, \vec{y'})$
5: **if** $e_{(i)} = 1$ **then**
6: $(\vec{y}, \vec{y'}) \leftarrow RNSMMM(\vec{y}, \vec{y'}, \vec{M}, \vec{M'})$
7: **end if**
8: **end for**
9: $(\vec{y}, \vec{y'}) \leftarrow RNSMMM(\vec{y}, \vec{y'}, \vec{1}, \vec{1'})$

$$X = \sum_{i=0}^{L-1} x^{(i)} 2^{ri} = \left(2^{r(L-1)}, \ldots, 2^r, 1\right) \begin{bmatrix} x^{(L-1)} \\ \vdots \\ x^{(1)} \\ x^{(0)} \end{bmatrix}, \tag{12.34}$$

where $0 \leq x^{(i)} \leq 2^r - 1$. By applying the modulo m_j operation in (12.34) we can convert the integer X to its associated RNS representation by

$$\langle X \rangle_{m_j} = \left\langle \sum_{i=0}^{L-1} x^{(i)} \left\langle 2^{ri} \right\rangle_{m_j} \right\rangle_{m_j}, \forall j \in [1, L]. \tag{12.35}$$

If constants $\left\langle 2^{ri} \right\rangle_{m_j}$ are pre-computed, this computation is a typical multiply accumulate operation and can be computed in L steps, when executed by L parallel units.

As (12.8) is the basis of the presented RNSMMM algorithm, it is useful to employ it also for the RNS-to-decimal conversion. Let us rewrite (12.8) as

$$\begin{aligned} X &= \sum_{i=1}^{L} \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i} \cdot M_i - \gamma M = \\ &= \left(2^{r(L-1)}, \ldots, 2^r, 1\right) \sum_{i=1}^{L} \left\{ \xi_i \cdot \begin{bmatrix} M_i^{(L-1)} \\ \vdots \\ M_i^{(1)} \\ M_i^{(0)} \end{bmatrix} - \gamma \begin{bmatrix} M^{(L-1)} \\ \vdots \\ M^{(1)} \\ M^{(0)} \end{bmatrix} \right\}, \end{aligned} \tag{12.36}$$

where $\xi_i = \left\langle x_i \cdot M_i^{-1} \right\rangle_{m_i}$. As soon as γ has been evaluated using the methods of Sect. 12.1.3.3, each row of (12.36) can be computed in parallel in each modulus

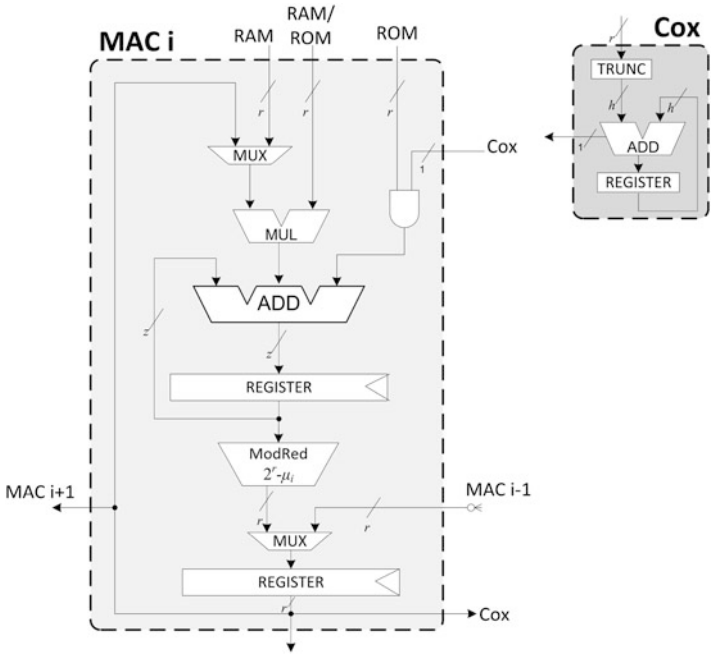

Fig. 12.6 Multiply Accumulate (MAC) RNS cell (adopted from [16])

channel by means of multiply accumulate operations. In this case, carry should be propagated from channel 1 until channel L [16].

All works in [2, 11, 16] utilize cell-based architectures to implement the algorithms in [2, 11, 16], respectively. Each cell corresponds to a single RNS modulus and utilizes a multiply accumulate unit followed by a modular reduction unit, which performs reduction by the corresponding RNS modulus using (12.6). The cell structure is shown in Fig. 12.6 [16]; a common bus that connects the cells is omitted for simplicity reasons. The multiply accumulate unit is depicted at the top of the cell while the modular reduction unit corresponds to the one depicted in Fig. 12.2. A ROM memory to store the moduli and pre-computed values as well as RAM memories to store intermediate results are employed. *The reader is highly recommended to map the MAC architecture with the operations in Algorithm 12.1 and the conversions to/from RNS and verify its operation*. An in-depth analysis on the number of clock cycles required to accomplish all operations along with an architectural comparison can be found in [11, 34]. Quite recently, novel dual-field implementations supporting RNS operations both over integers and polynomials have also been presented [34].

12.4.3 RNS Application in ECC

In the following section, an application of RNSMMM algorithm to ECC is demonstrated [9]. The design employs bases of moduli offering efficient arithmetic and differs from the approaches analyzed before in the sense that dedicated components are employed for IO conversions and BC operations. The design employs Eq. (12.16) to obtain the MRS digits of the result, while (12.15) is computed modulo each modulus of the new base.

Sets of three and four moduli are used in implementing the RNSMMM of Algorithm 12.4. The form of the moduli determines to a large extent the efficiency of the arithmetic operations and the layout of the input/output converters [27]. The RNS bases employed are shown in Table 12.3. In the first base, RNS moduli of the form $2^r - 2^{t_i} - 1$, where $t_i < r/2$, are employed, which offer simple modulo reduction operations [4]. The second base is realized by sets of three or four moduli of the special forms $\{2^r, 2^{r+1} - 1, 2^r - 1\}$ [25] and $\{2^r, 2^r - 1, 2^{r+1} - 1, 2^{r-1} - 1\}$, which also provide efficient arithmetic operations and IO conversions [4]. It is easy to check that the bases satisfy the condition $4N < M < M'$ in Sect. 12.4.1 [4].

Table 12.3 RNS bases for use in ECC

	Field (bit)	Base $\mathcal{B}$	Base $\mathcal{B}'$
3-modulus RNS bases	160	$\{2^{56} - 2^{11} - 1,$ $2^{56} - 2^{16} - 1,$ $2^{56} - 2^{20} - 1$	$\{2^{56},$ $2^{56} - 1,$ $2^{57} - 1\}$
3-modulus RNS bases	192	$\{2^{66} - 2^{17} - 1,$ $2^{66} - 2^{18} - 1,$ $2^{66} - 2^{24} - 1\}$	$\{2^{66},$ $2^{66} - 1,$ $2^{67} - 1\}$
4-modulus RNS bases	192	$\{2^{50} - 2^{20} - 1,$ $2^{50} - 2^{22} - 1,$ $2^{50} - 2^{18} - 1,$ $2^{50} - 2^{10} - 1\}$	$\{2^{50},$ $2^{50} - 1,$ $2^{51} - 1,$ $2^{49} - 1\}$
4-modulus RNS bases	224	$\{2^{58} - 2^{22} - 1,$ $2^{58} - 2^{13} - 1,$ $2^{58} - 2^{10} - 1,$ $2^{58} - 2^{16} - 1\}$	$\{2^{58},$ $2^{58} - 1,$ $2^{59} - 1,$ $2^{57} - 1\}$
4-modulus RNS bases	256	$\{2^{66} - 2^{22} - 1,$ $2^{66} - 2^{24} - 1,$ $2^{66} - 2^{18} - 1,$ $2^{66} - 2^{17} - 1\}$	$\{2^{66},$ $2^{66} - 1,$ $2^{67} - 1,$ $2^{65} - 1\}$

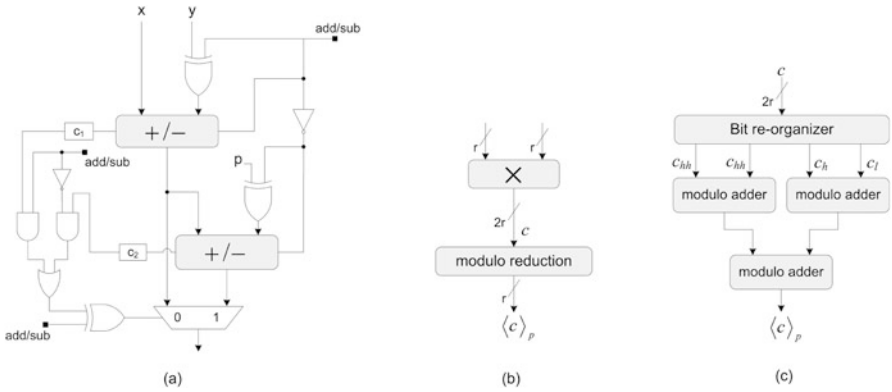

Fig. 12.7 (**a**) Generic modulo p adder/subtractor [35], (**b**) Modulo $2^r - 2^{t_i} - 1$ multiplier, (**c**) Reduction circuit (adopted from [9])

12.4.3.1 Modular Adders and Multipliers

For the first base, where moduli of the form $2^r - 2^{t_i} - 1$ are utilized, the modular adder and multiplier depicted in Fig. 12.7 are employed. Regarding modular multiplication, two r-bit operands are multiplied and a $2r$-bit value is obtained. Modular reduction of a $2r$-bit value c with moduli of the form $2^r - 2^{t_i} - 1$ can be written using its higher r bits, denoted as c_h, and its r lower bits, denoted as c_l, as

$$\langle c \rangle_{2^r - 2^{t_i} - 1} = \langle c_h 2^r + c_l \rangle_{2^r - 2^{t_i} - 1} \,. \tag{12.37}$$

Since $2^r \mod (2^r - 2^{t_i} - 1) = 2^{t_i} + 1$, it holds that

$$\begin{aligned}
\langle c \rangle_{2^r - 2^{t_i} - 1} &= \left\langle \left\langle c_h (2^{t_i} + 1) \right\rangle_{2^r - 2^{t_i} - 1} + c_l \right\rangle_{2^r - 2^{t_i} - 1} \\
&= \left\langle \overbrace{\left\langle c_h 2^{t_i} \right\rangle_{2^r - 2^{t_i} - 1}}^{(r+t)\text{-bits}} + c_h + c_l \right\rangle_{2^r - 2^{t_i} - 1} \\
&= \langle \langle c_{hh} 2^r + c_{hl} \rangle_{2^r - 2^{t_i} - 1} + c_h + c_l \rangle_{2^r - 2^{t_i} - 1} \\
&= \left\langle \left\langle c_{hh} 2^{t_i} + c_{hh} \right\rangle_{2^r - 2^{t_i} - 1} + c_{hl} + c_h + c_l \right\rangle_{2^r - 2^{t_i} - 1} \\
&= \left\langle c_{hh} \underbrace{0 \ldots 0}_{t_i\text{-bits}} + c_{hh} + c_{hl} + c_h + c_l \right\rangle_{2^r - 2^{t_i} - 1} \\
&= \left\langle \underbrace{c_{hh} c_{hh}}_{\text{concatenation}} + c_{hl} + c_h + c_l \right\rangle_{2^r - 2^{t_i} - 1}
\end{aligned} \tag{12.38}$$

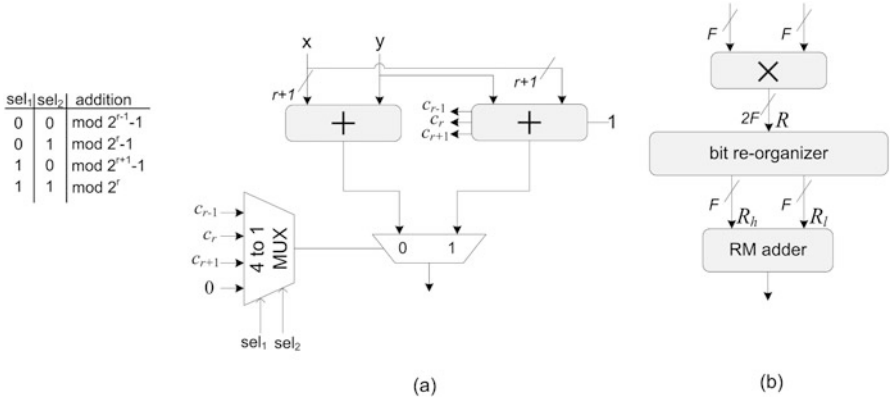

Fig. 12.8 (**a**) Reconfigurable modulo (RM) adder, (**b**) RM Multiplier, ($F = r, r-1, r+1$) (adopted from [9])

For the second base, the reconfigurable modulo (RM) adder shown in Fig. 12.8 is employed. Based on this adder, addition and multiplication modulo 2^r, $2^{r-1}-1$, 2^r-1, and $2^{r+1}-1$ can be accommodated in the same hardware module. Note that the RM adder shown in Fig. 12.8 has $(r-1)$-bit FA delay less than the modulo adder in Fig. 12.7 (worst case). Hence, the second base supports more efficient arithmetic operations. Regarding multiplication, the $2F$-bit result R is split into two F-bit LSD and MSD parts (R_l and R_h, respectively), where ($F = r, r-1, r+1$) and reduction modulo $2^F - 1$ can be achieved by a modular addition of R_l and R_h [4].

12.4.3.2 Conversion from Base $\mathcal{B}'$ to Base $\mathcal{B}$

In step 3 of RNSMMM, a base conversion from base $\mathcal{B}'$ to base $\mathcal{B}$ is required. This conversion consists of a residue-to-MRS conversion in base $\mathcal{B}'$ and then an MRS-to-residue conversion in base $\mathcal{B}$. The core operation in calculation of U_i, $\forall i = 2, 3, 4$, in (12.16) is

$$H = \left\langle \left(x_j - U_i\right) m_{i,j}^{-1} \right\rangle_{m_j}. \tag{12.39}$$

Hardware implementations of (12.39) for area and time efficient designs are shown in Fig. 12.9. Considering four-modulus RNS bases, for each U_i ($i = 2, 3, 4$), the implementation shown in Fig. 12.9 is employed. The bit re-organizer provides the required shifts according to the pre-calculated multiplicative inverses.

Residues in $\mathcal{B}$ must be calculated after the calculation of mixed-radix digits in base $\mathcal{B}'$. For the calculation of MRS-to-RNS conversion from base $\mathcal{B}'$ to base $\mathcal{B}$ for the four-modulus RNS bases, it holds that

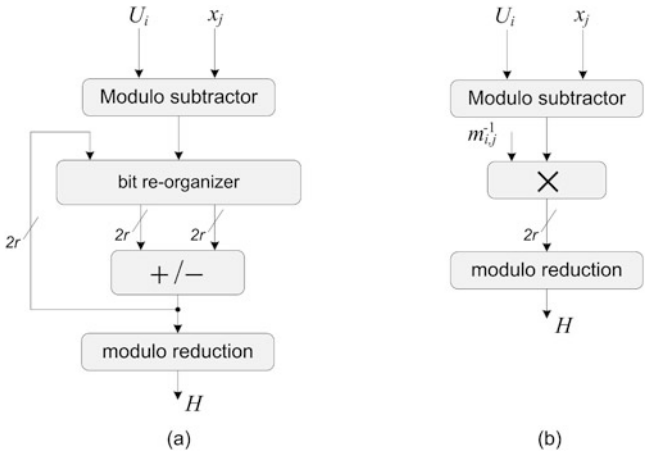

Fig. 12.9 Calculation of core operation H in RNS-to-MRS conversion for the first base (**a**) area efficient design, (**b**) fast design (adopted from [9])

$$x_j = \langle U_1 + m_1 \left(U_2 + m_2(U_3 + m_3 U_4)\right)\rangle_{m_j}, \tag{12.40}$$

where m_j are the moduli of the forms $2^r, 2^r - 12^{r+1} - 1, 2^{r-1} - 1$, and $m_i, i = 1, 2, 3$, are moduli of the form $2^r - 2^{t_i} - 1$. The form of the considered bases with simple multiplicative inverses allows for fast or area efficient adder-based structures, which can be realized by using one RM adder for each modulus [9].

12.4.3.3 Conversion from Base $\mathcal{B}$ to Base $\mathcal{B}'$

In order to mechanize RNS-to-MRS conversion in base $\mathcal{B} = (2^r, 2^r - 1, 2^{r+1} - 1)$, based on (12.16) and considering $m_1 = 2^r, m_2 = 2^r - 1, m_3 = 2^{r+1} - 1$, we get

$$U_1 = x_1 \tag{12.41}$$

$$U_2 = \left\langle (x_2 - U_1)\, m_{1,2}^{-1} \right\rangle_{m_2} \tag{12.42}$$

$$U_3 = \left\langle \left((x_3 - U_1)\, m_{1,3}^{-1} - U_2\right) m_{2,3}^{-1} \right\rangle_{m_3}. \tag{12.43}$$

The required multiplicative inverses in (12.42) and (12.43) are $\left\langle m_1^{-1} \right\rangle_{m_2} = 1$, $\left\langle m_1^{-1} \right\rangle_{m_3} = 2$ and $\left\langle m_2^{-1} \right\rangle_{m_3} = -2$ [25]. Due to the simple form of multiplicative inverses, the aforementioned adder-based structure can be employed, both for the fast and for the area efficient design. Regarding the MRS-to-RNS conversion to base $\mathcal{B}$, it holds that

$$x_j = \langle U_1 + m_1 \left(U_2 + m_2 \left(U_3 + m_3 U_4\right)\right)\rangle_{m_j}. \tag{12.44}$$

It is apparent that all calculations in (12.44) consist of simple shifts and addition operations.

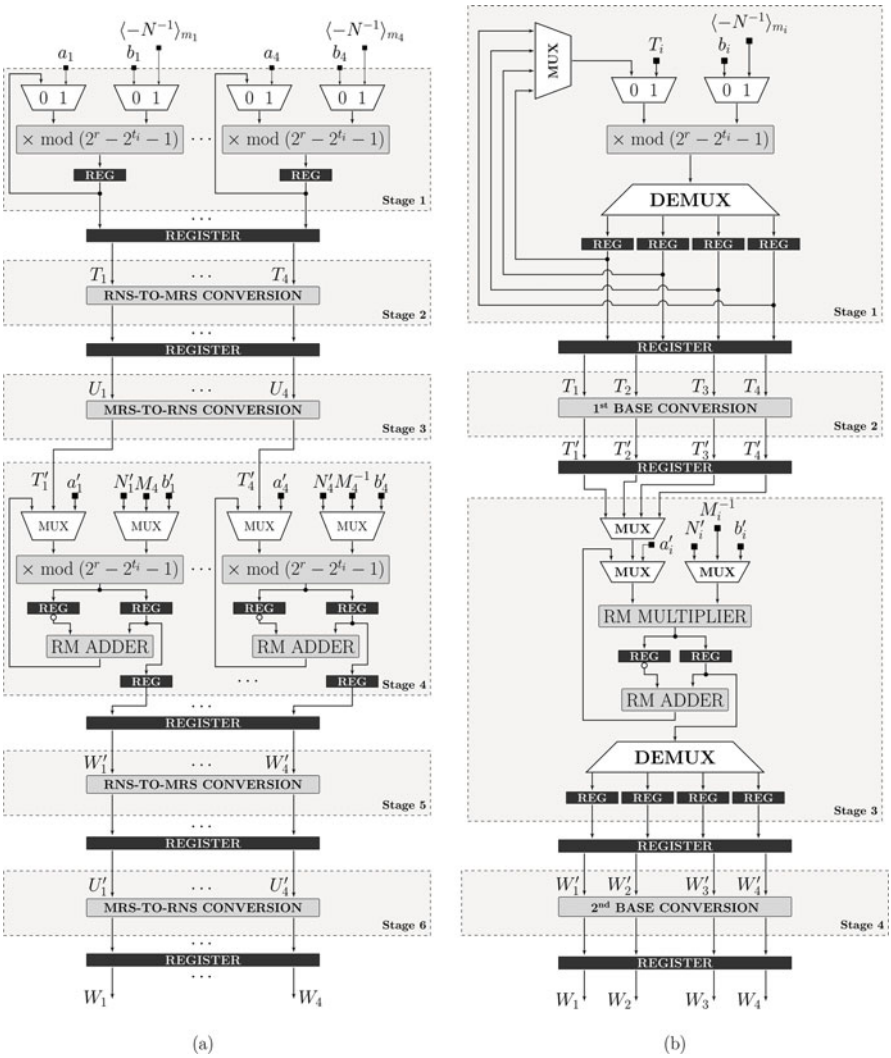

Fig. 12.10 Pipelined RNSMMM architectures for ECC, (**a**) fast design, (**b**) area efficient design (adopted from [9])

12.4.3.4 Hardware Architecture for RNS Montgomery Multiplication

Two versions of pipelined RNS architectures for the RNSMMM targeting area and speed efficiency are shown in Fig. 12.10. In each modulus channel of stages 1 and 4 of the pipelined implementations, the modular multipliers and adders in Figs. 12.7 and 12.8 are employed. For the base conversion operations, the modulo adders and multipliers described in respective sections are utilized. The area efficient design utilizes one modulo $2^r - 2^{t_i} - 1$ multiplier, one RM multiplier, one RM adder, and two

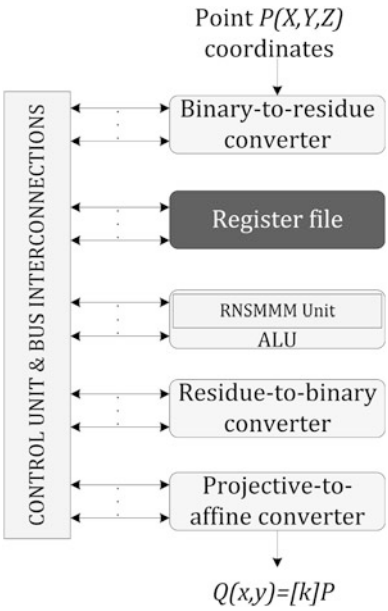

Fig. 12.11 ECC processor architecture

base conversion units with adder-based structure connected in a four-stage pipelined layout (Fig. 12.10b).

A typical ECC processor is illustrated in Fig. 12.11. It consists of IO converters for the conversions to/from RNS representations, a register file, the ALU unit with the presented RNSMMM architectures, and the converter from projective to affine coordinates. Based on the control unit a corresponding algorithm for point multiplication is performed, for example, Algorithm 12.2, based on the binary expansion of scalar $[k]$. Since the presented bases support efficient arithmetic, the IO converters encompass an architecture similar to the ones presented for the modular addition, subtraction, and base conversion.

12.5 Robustness Issues

When it comes to robustness of RNS-based cryptosystems against hardware-fault attacks recent research efforts have revealed some interesting properties of RNS arithmetic [5, 10, 33]. Both ECC and RSA are considered in these works. We discuss in this section RSA-CRT, a popular derivative of RSA. In this scheme, the digital signature operation $s = m^d \mod N$ is split into two operations $s_p = m^{d_p} \mod p$ and $s_q = m^{d_q} \mod q$, where $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$. CRT ensures that the combination of these two values produces the signature s as

$$s = s_q + \left\langle \left(s_p - s_q\right) \cdot \left\langle q^{-1} \right\rangle_p \right\rangle_p \cdot q \tag{12.45}$$

denoted from now on as $s = \mathrm{CRT}(s_p, s_q)$ [17]. In this way, an approximate 4-time speedup of operations is achieved, explaining why RSA-CRT has gained momentum in recent years [20, 21].

Despite this significant performance improvement, RSA-CRT has proven to be extremely vulnerable against hardware-fault attacks [1, 8, 13]. Let's assume an erroneous output generated randomly during the execution of a cryptographic operation. Without loss of generality, let the fault be in the modulus p channel, denoted as $\tilde{s}_p$. This will produce a faulty signature $\tilde{s} = \mathrm{CRT}\left(\tilde{s}_p, s_q\right)$. An adversary can then factorize the public modulus N by computing its prime factor q as $q = \gcd\{(\tilde{s}^e - m) \mod N, N\}$ and consequently obtain $p = N/q$.

In [36], Shamir modified the basic RSA-CRT algorithm in (12.45), by introducing a random prime r so that $s_{pr} = m^{d \bmod (p-1)(r-1)} \bmod pr$ and $s_{qr} = m^{d \bmod (q-1)(r-1)} \bmod qr$. The method checks whether $s_{pr} \equiv s_{qr} \mod r$ holds before combining them with CRT. If $s_{pr} \equiv s_{qr} \mod r$, the computation is error-free, but the step of CRT combination is left unprotected. Moreover, Shamir's method requires the knowledge of the straightforward RSA private key d in an RSA-CRT context, which is impractical, since the key is given in CRT format [42].

In fact, Shamir's countermeasures were broken by exploiting this weakness [1]. The authors proposed an improved implementation that included the protection of the CRT recombination step. But random number generation is a problem in this scheme, since generating random numbers for each signature operation results in large time overhead.

In [42], the authors proposed a method based on modulus expansion. It computes $m^d \mod N$ in $\mathbb{Z}_{Nr^2}$, where r is a small random integer co-prime with N. The message m is transformed to $\widehat{m}$, so that $\widehat{m} = m \mod N$ and $\widehat{m} = 1 + r \mod r^2$. Then, s and $\widehat{s}$ are computed as $s = \left\langle m^d \right\rangle_N$ and $\widehat{s} = \left\langle \widehat{m}^d \right\rangle_{Nr^2}$. If $\widehat{s} \equiv s \mod N$, then the protocol is error-free. However, the method did not improve much the performance overhead [22].

The Maximum Likelihood Estimation (MLE) algorithm was also exploited as a countermeasure scheme [1]. Unlike the square-and-multiply algorithm which performs on average 1.5 modular multiplications per bit of the exponent, the MLE algorithm performs two modular multiplications for each bit of exponent, and thereby increases execution time.

An ingenious fault attack based on the *safe-error* concept has been developed in [46]. It was observed that during a modular exponentiation using typical square-and-multiply algorithms, if the exponent bit is 0, then the result of a modular multiplication is not used. By inducing an error during multiplication and by testing whether the result is correct or not, the attacker can deduce the bit of the secret exponent. However, a countermeasure was provided using MLE [15].

Another class of countermeasures is based on "fault-infective" techniques [7, 47]. They are based on the idea of modifying the RSA-CRT computations in such a way

that a faulty signature in one channel will infect the final signature after the CRT recombination. Unfortunately, like in [36], not only the knowledge of d is required, but also the techniques rely on some very strong assumptions. For example, some parameters t_1 and t_2 introduced in [7] require that $\gcd(t_1, t_2) = \gcd(d, \varphi(t_1)) = \gcd(d, \varphi(t_2)) = 1$, where φ is the Euler's totient function. t_1, t_2 should normally be generated once, along with the RSA key, and the same values should be used throughout the key's lifetime. However, these values cannot be stored in such a personalized context, meaning that the generation of t_1, t_2 for each signature is not a task of negligible computational complexity.

The majority of the aforementioned countermeasures is based on modifications of the RSA-CRT protocol, which amount to extra operations and increased algorithmic complexity for the RSA-CRT execution. These solutions rely on the 2-modulus splitting of RSA calculations using a primitive RNS consisting of just the moduli p and q. Following a different approach, the multi-modulus RNS Montgomery multipliers presented in the previous sections are examined from a hardware-fault tolerance point of view.

12.5.1 Hardware-Fault Tolerance in MRC-Based RNS Montgomery Multipliers

It is apparent that steps 1, 2, 4, 5, and 6 of the RNSMMM Algorithm 12.4 are performed in parallel in each modulus channel. If the algorithm was completely parallel, an error in modulus channel i would not influence the remaining channels and, thus, the GCD attack would be feasible. Fortunately, it has been shown that the base conversion provides the desired mechanism for fault tolerance [33].

Assume a permanent error $\tilde{t'}_i$ in modulus channel $1 \leq i \leq L$. Note that since step 2 of Algorithm 12.4 uses the result of step 1, the faulty result will always amount to $\tilde{t'}_i$. By observation, employing $\tilde{t'}_i$ in the MRC base conversion of step 3 yields

$$\tilde{U}_i \leftarrow \left\langle \left(\tilde{t'}_i - t'_j\right) m_{j,i}^{-1} \right\rangle_{m_i}, i \in [2, L], \forall j \in [1, i-1]. \tag{12.46}$$

(12.46) corresponds to Eq. (12.16) and, due to the recursive nature of MRC, an error that occurred in position i will always cascade to the next channels and produce a faulty $\tilde{U}_L$, even if the error occurs at the very last step of calculations in channel L. This value is used in Eq. (12.15) to complete the base conversion process and as a result, a faulty $\vec{\tilde{T}}$ is generated and injected in step 4 of Algorithm 12.4. Note also that, since the quantity $\tilde{U}_L$ is involved in the calculation of all moduli of the new RNS base, all channels are infected. Using a similar analysis, it is easy to show that even if the error occurs after the first base conversion, the second base conversion at step 7 of Algorithm 12.4 will infect all channels in the same manner, rendering thus the GCD attack infeasible [33].

12.5.2 Hardware-Fault Tolerance in CRT-Based RNS Montgomery Multipliers

Let us analyze in a similar manner the BC Algorithm 12.1 based on CRT. Clearly, steps 1–5 and 14–16 involve parallel operations in all channels, so fault tolerance should be examined for the steps 6–13. In the case of a permanent error, a faulty $\tilde{\gamma}_j^*, j \in [1, L]$, is generated in steps 8–9, which consequently produces

$$\tilde{s}_{i,j} = s_{i,(j-1)} + \xi'_j \cdot \langle M'_j \rangle_{m_i} + \tilde{\gamma}_j^* \cdot \langle (-M') \rangle_{m_i}, \forall i, j \in [1, L]. \tag{12.47}$$

This means that all channels are affected by the error, thus the parallel operations of steps 14–16 are also affected.

However, if an adversary is able to insert an error during the steps 14–16, only one (or several) channel can be affected, which makes the GCD attack easily mountable. To overcome this issue, an extra checking procedure could be inserted in steps 14–16 of Algorithm 12.1 based on the following pseudo code:

```
for all i = 1, ..., L do
    if s_{i,L} == s_{i,L} of step 11 then
        x_i = ⟨s_{i,L}⟩_{m_i}
    else
        error detected
    end if
end for
```

The solution checks whether the quantities $s_{i,L}$ are identical to the values obtained in the previous step 11. If not, a malicious error has been detected. The solution requires the storage of the L values of step 11 and a comparison with the $s_{i,L}$s employed in step 15. Note that this solution does not issue significant overhead, since the checking procedure can be executed only once at the end of an RSA exponentiation.

12.6 Conclusion and Future Outlook

Crypto-hardware design forms a special category, as the prospect designer should not only assess performance but also resistance against attacks. Under this perspective, crypto-hardware design poses extra difficulties and challenges considering especially the fact that, as years pass by, security standards need to be constantly strengthened. This chapter attempted to approach the problem of cryptographic hardware design with the use of unconventional arithmetic in a holistic manner, covering aspects of algorithmic analysis, crypto-hardware design, and security validation of the presented algorithms.

Looking forward, the cryptanalytic properties of RNS-based architectures can be further extended, to include attacks other than hardware-fault related. The role of base conversion should be meticulously analyzed to reveal new possibilities for cryptanalytic resistance. An interesting derivative is the security potential offered in RNS-based cryptosystems by seamlessly changing the modulus set. Investigating the applicability of RNS to other PKC systems, for example, the emerging lattice-based cryptography [14, 31], could also generate new and interesting cryptanalytic properties, architectures, and algorithms.

In general, existing solutions employing conventional binary arithmetic for Montgomery multiplication have been extensively analyzed and the optimizations proposed are so incremental, that the research space on the field narrows steadily. We hope that this chapter provided solid indications and insights that non-conventional arithmetic may offer new means for tackling design problems of crypto-hardware and further extend the research space in this active field.

Acknowledgements Our warmest thanks to Mrs. Elli Kyrmanidou (PhD candidate in LMU Munich) for editing the chapter.

References

1. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.P. Seifert, Fault attacks on RSA with CRT: concrete results and practical counter-measures, in Proceedings of International Workshop Cryptographic Hardware and Embedded Systems (CHES'02) (2002), pp. 260–275
2. J.C. Bajard, L. Imbert, A full RNS implementation of RSA. IEEE Trans. Comput. **53**, 769–774 (2004)
3. J.C. Bajard, L.S. Didier, P. Kornerup, Modular multiplication and base extensions in residue number systems, in Proceedings of the 15th Symposium on Computer Arithmetic, ARITH'01 (2001), pp. 59–65
4. J. Bajard, M. Kaihara, T. Plantard, Selected RNS bases for modular multiplication, in 19th IEEE International Symposium on Computer Arithmetic (2009), pp. 25–32
5. J. Bajard, J. Eynard, F. Gandino, Fault detection in RNS Montgomery modular multiplication, in 2013 21st IEEE Symposium on Computer Arithmetic (ARITH) (2013), pp. 119–126
6. I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography (Cambridge University Press, Cambridge, 2002)
7. J. Blömer, M. Otto, J.P. Seifert, A new CRT-RSA algorithm secure against bellcore attacks, in In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS'03 (2003), pp. 311–320
8. D. Boneh, R. DeMillo, R. Lipton, On the importance of eliminating errors in cryptographic computations. J. Cryptol. **14**, 101–119 (2001)
9. M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, K. Navi, Efficient RNS implementation of elliptic curve point multiplication over $GF(p)$. IEEE Trans. Very Large Scale Integr. VLSI Syst. **8**(21), 1545–1549 (2013)
10. A.P. Fournaris, N. Klaoudatos, N. Sklavos, C. Koulamas, Fault and power analysis attack resistant RNS based Edwards curve point multiplication, in *Proceedings of the Second Workshop on Cryptography and Security in Computing Systems, CS2'15* (ACM, New York, 2015), pp. 43:43–43:46 http://doi.acm.org/10.1145/2694805.2694814
11. F. Gandino, F. Lamberti, P. Montuschi, J. Bajard, A General Approach for Improving RNS Montgomery exponentiation using pre-processing, in *2011 20th IEEE Symposium on Computer Arithmetic (ARITH)* (2011), pp. 195–204

12. F. Gandino, F. Lamberti, G. Paravati, J.C. Bajard, P. Montuschi, An algorithmic and architectural study on Montgomery exponentiation in RNS. IEEE Trans. Comput. **61**(8), 1071–1083 (2012)
13. C. Giraud, An RSA implementation resistant to fault attacks and to simple power analysis. IEEE Trans. Comput. **55**(9), 1116–1120 (2006)
14. O. Goldreich, S. Goldwasser, S. Halevi, Public-key cryptosystems from lattice reduction problems, in *Advances in Cryptology CRYPTO'97*, ed. by B.J. Kaliski. Lecture Notes in Computer Science, vol. 1294 (Springer Berlin, 1997), pp. 112–131. doi:10.1007/BFb0052231. http://dx.doi.org/10.1007/BFb0052231
15. M. Joye, S.M. Yen, The Montgomery powering ladder, in: *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES'02) LNCS* (2002), pp. 291–302
16. S. Kawamura, M. Koike, F. Sano, A. Shimbo, Cox-Rower architecture for fast parallel Montgomery multiplication, in *EUROCRYPT'00: Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques* (Springer, Berlin, 2000), pp. 523–538
17. D.E. Knuth, *The Art of Computer Programming, Seminumerical Algorithms, vol. 2 (3rd ed.)* (Addison-Wesley Longman, Boston, 1997)
18. N. Koblitz, Elliptic curve cryptosystems. Math. Comput. **48**, 203–209 (1987)
19. I. Koren, *Computer Arithmetic Algorithms* (A K Peters, Natick, 2002)
20. R. Lab, High-Speed RSA Implementation (2011). ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf
21. R. Lab, RSA Hardware Implementation (2011). ftp://ftp.rsasecurity.com/pub/pdfs/tr801.pdf
22. K. Ma, H. Liang, K. Wu, Homomorphic property-based concurrent error detection of RSA: a countermeasure to fault attack. IEEE Trans. Comput. **61**(7), 1040–1049 (2012)
23. A.J. Menezes, S.A. Vanstone, P.C.V. Oorschot, Handbook of Applied Cryptography, 1st edn. (CRC Press, Boca Raton, 1996)
24. V. Miller, Use of elliptic curves in cryptography, in *Advances in Cryptology (CRYPTO'85)*. LNCS, vol. 218 (1986), pp. 47–426
25. P. Mohan, RNS-to-binary converter for a new three-moduli set $\{2^{n+1}-1, 2^n, 2^n-1\}$. IEEE Trans. Circuits Syst. Express Briefs **54**(9), 775–779 (2007)
26. P.L. Montgomery, Modular multiplication without trial division. Math. Comput. **16**, 519–521 (1985)
27. K. Navi, A. Molahosseini, M. Esmaeildoust, How to teach residue number system to computer scientists and engineers. IEEE Trans. Educ. **54**(1), 156–163 (2011)
28. H. Nozaki, M. Motoyama, A. Shimbo, S. Kawamura, Implementation of RSA algorithm based on RNS Montgomery multiplication, in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*. LNCS, vol. 2162 (2001), pp. 364–376
29. K. Posch, R. Posch, Base extension using a convolution sum in residue number systems. Computing **50**, 93–104 (1993)
30. K. Posch, R. Posch, Modulo reduction in residue number systems. Trans. Parallel Distrib. Syst. **6**(5), 449–454 (1995)
31. O. Regev, Lattice-based cryptography, in *Advances in Cryptology CRYPTO'06*. Lecture Notes in Computer Science (Springer, Berlin, 2006), pp. 131–141
32. R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**, 120–126 (1978)
33. D. Schinianakis, T. Stouraitis, Hardware-fault attack handling in RNS-based Montgomery multipliers, in 2013 IEEE International Symposium on Circuits and Systems (ISCAS) (2013), pp. 3042–3045
34. D. Schinianakis, T. Stouraitis, Multifunction residue architectures for cryptography. IEEE Trans. Circuits Syst. Regul. Pap. **61(4)**, 1156–1169 (2014)
35. D. Schinianakis, A. Fournaris, H. Michail, A. Kakarountas, T. Stouraitis, An RNS implementation of an F_p elliptic curve point multiplier. IEEE Trans. Circuits Syst. I **56**(6), 1202–1213 (2009)
36. A. Shamir, Improved method and apparatus for protecting public key schemes from timing and fault attacks. US Patent 5,991,415 (1999)

37. M. Shenoy, R. Kumaresan, A fast and accurate RNS scaling technique for high speed signal processing. IEEE Trans. Acoust. Speech Signal Process. **37**(6), 929–937 (1989)
38. A. Skavantzos, Y. Wang, New efficient RNS-to-weighted decoders for conjugate-pair-moduli residue number systems, in *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers, 1999* (1999), vol. 2, pp. 1345–1350
39. A. Skavantzos, M. Abdallah, T. Stouraitis, D. Schinianakis, Design of a balanced 8-modulus RNS, in *16th IEEE International Conference on Electronics, Circuits, and Systems, 2009. ICECS 2009* (2009), pp. 61–64
40. N. Szabo, R. Tanaka, *Residue Arithmetic and its Applications to Computer Technology* (McGraw-Hill, New York, 1967)
41. F.J. Taylor, Residue arithmetic: a tutorial with examples. IEEE Comput. **17**, 50–62 (1988)
42. D. Vigilant, RSA with CRT: a new cost-effective solution to thwart fault attacks, in *Proceedings of International Workshop Cryptographic Hardware and Embedded Systems (CHES 08)* (2008), pp. 130–145
43. W. Wang, M. Swamy, O. Ahmad, Y. Wang, New Chinese remainder theorems applications to special moduli sets, in *CCECE99*, vol. 2 (1999), pp. 1345–1350
44. Y. Wang, Residue-to-binary converters based on new Chinese remainder theorems. IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. **47**(3), 197–205 (2000)
45. H.M. Yassine, W. Moore, Improved mixed-radix conversion for residue number system architectures. IEE Proc. G Circuits Devices Syst. **138**(1), 120–124 (1991)
46. S. Yen, M. Joye, Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Comput. **49**(9), 967–970 (2000)
47. S. Yen, S. Kim, S. Lim, S. Moon, RSA Speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. IEEE Trans. Comput. **52**(4), 461–472 (2003)

Chapter 13
RNS Approach in Lattice-Based Cryptography

Jean-Claude Bajard and Julien Eynard

Conventions

a, b, x	Integers
$\boldsymbol{A}, \boldsymbol{R}$	Matrices
$\boldsymbol{c}, \boldsymbol{p}$	Vectors
$\lvert a \rvert_p$	Modular reduction (a mod p)
$\mathcal{M}$	An RNS base, or the product of elements of $\mathcal{M}$
$\mathcal{M}^{\#}$	Cardinality of RNS base $\mathcal{M}$
$a_{\mathcal{M}}$	Representation of integer a in RNS base $\mathcal{M}$
$\boldsymbol{c}_{\mathcal{M}}$	Representation of vector $\boldsymbol{c}$ in RNS base $\mathcal{M}$

13.1 Introduction

Lattice-based cryptography (LBC) is a main branch of modern cryptography. Growing up and evolving mainly since 1996 [1], it is known to be a viable post-quantum alternative to all cryptosystems based on the factorization and discrete logarithm problems [30], such as the widely used Diffie-Hellman [8] key-exchange protocol and RSA [26]. Beyond a post-quantum security, many other important

J.-C. Bajard (✉)
Sorbonne Universités, UPMC, CNRS, LIP6, Paris, France
e-mail: jean-claude.bajard@lip6.fr

J. Eynard
Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
e-mail: jeynard@uwaterloo.ca

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_13

properties make LBC interesting. Nowadays, primitives based on q-ary ideal lattices arouse interest since they are related to special properties, such as security proofs based on worst-case of computationally hard problems in lattices like Learning-with-Error [21, 24, 25], or homomorphic encryption [6, 7, 10], which is constantly evolving since its appearance in 2009.

Along these general considerations about current LBC, some of the first cryptosystems were based on general integer lattices. Shortly after that Ajtai showed tight connection between average/worst-case of Shortest Vector Problem (SVP) in a lattice together with NP-hardness of worst-case, the GGH asymmetric cryptosystem inspired by these ideas appeared [11], together with NTRU [12], whose principle is very similar to GGH but in q-ary ideal lattices. Even if GGH does not support any security proof, it does not require the use of specific structural conditions, like q-ary or ideal lattices, and it also benefits from asymptotic efficiency. Originally proposed with size of parameters making it viable in practice, encryption and signature GGH schemes, however, contained major flaws [19, 20], forcing the lattice dimensions to be greater than suggested, essentially in order to put attacks based on lattice reduction out of action. Thus, many consecutive works were concerned by as well thwarting these attacks [17, 32] as bringing practical improvements [18, 22, 27] to make such scheme viable under safe parameters.

The asymptotic computational efficiency of LBC is due to the fact that some core primitives rely on matrix–vector products. However, large cryptographic parameters can make even such simple computations burdensome. Residue Number Systems are very useful for accelerating computation patterns like sum-of-products (and then matrix algebras). Thus, the use of RNS within LBC seems a natural approach. A core operation in lattices is the "Rounding-off". It was suggested by Babai [2] for efficiently inverting a class of trapdoor functions when the secret key is known. Beyond potential applications in cryptography (e.g. within GGH-like decryption), this procedure can also be useful in cryptanalysis. To sum up, it is a basic tool in the domain of lattices, which can be used for many purposes. Hence, the question of its optimization is relevant and it is the object of this chapter.

13.2 Introduction to Lattices

13.2.1 Some Hard Problems in Lattices

A lattice is a mathematical object which can be viewed as a regular grid in the real ℓ-dimensional space $\mathbb{R}^\ell$.

Definition 1. A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^\ell$.

A (full-rank) lattice is defined by a basis $\mathcal{R} = (\boldsymbol{r_1}, \ldots, \boldsymbol{r_i}, \ldots, \boldsymbol{r_\ell})$ of ℓ independent vectors $\boldsymbol{r_i} \in \mathbb{R}^\ell$. An element $\boldsymbol{x}$ of $\mathcal{L}$ is represented by $\boldsymbol{x} = \sum_{i=1}^{\ell} x_i \boldsymbol{b_i}$ with $x_i \in \mathbb{Z}$. In other words, $\mathcal{L} = \boldsymbol{r_1}\mathbb{Z} \oplus \ldots \oplus \boldsymbol{r_\ell}\mathbb{Z}$.

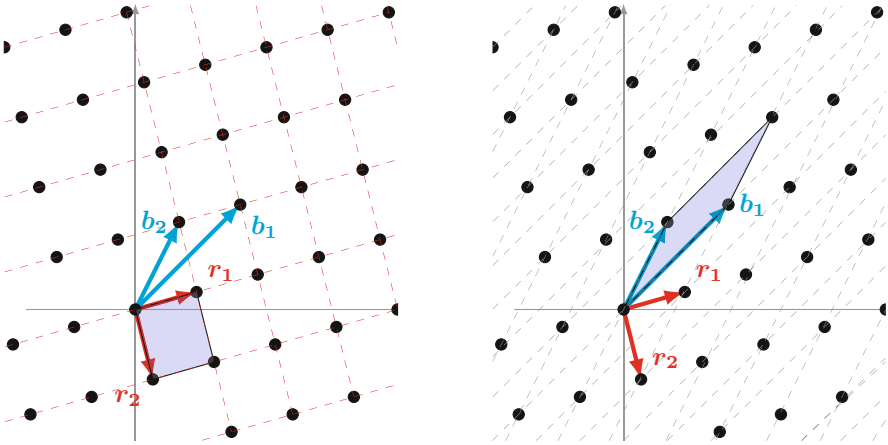

Fig. 13.1 Two different basis $\mathcal{R} = (\boldsymbol{r_1}, \boldsymbol{r_2})$, $\mathcal{B} = (\boldsymbol{b_1}, \boldsymbol{b_2})$ of a same lattice (*black grid*) with their fundamental domains (in *blue*)

For a matter of computation efficiency, LBC used to involve integer lattices. It means that the vectors $\boldsymbol{r_i}$ will lie in $\mathbb{Z}^\ell$. A basis $\mathcal{R}$ of a lattice $\mathcal{L}$ can be represented as a matrix $\boldsymbol{R}$. Each column of $\boldsymbol{R}$ is an element of the basis $\mathcal{R}$. Then, $\mathcal{L}$ is simply the set of all vectors $\boldsymbol{Rv}$, with $\boldsymbol{v}$ describing $\mathbb{Z}^\ell$.

A lattice owns infinitely many basis as soon as $\ell \geqslant 2$. From a specific basis $\mathcal{R}$ of a same lattice, any other basis can be obtained by multiplying it by a unimodular matrix $\boldsymbol{U} \in GL_\ell(\mathbb{Z})$. A lattice $\mathcal{L}$ is characterized by a fundamental domain $\mathcal{D}_\mathcal{R}$ associated with a basis $\mathcal{R}$. It is the parallelotope $\left\{\sum_{i=1}^{\ell} t_i \boldsymbol{r_i}, t_i \in [0, 1)\right\}$. Another basis provides another fundamental domain (cf. Fig. 13.1). The fundamental volume $\mathcal{V}$ of the lattice $\mathcal{L}$ is independent of the basis, and is an invariant of the lattice. It represents the volume of each of the fundamental domains. Given any basis $\mathcal{R}$, $\mathcal{V}$ is equal to $|\det \boldsymbol{R}|$.

Some of the main problems related to lattices are the closest vector problem (CVP) and the shortest vector problem (SVP). The CVP is, given a lattice and a target vector in the space, to find the point of the lattice which is the closest to the target. The SVP aims at finding the shortest non-zero vector of a lattice.

The hardness of these problems is related to the difficulty of computing a reduced basis. A reduced basis corresponds to a basis "close" to an orthogonal one. As a consequence, the coordinates of its elements are small. In Fig. 13.1, the basis $\boldsymbol{R}$ is more reduced than $\boldsymbol{B}$.

Basically, if a "nearly orthogonal" basis is known, it becomes easy to solve these problems. To solve a CVP given a "bad" basis (poorly reduced), then even by using the better lattice reduction algorithms like LLL and BKZ [14, 28] one will only be able to compute, in polynomial time, a lattice point close to the target up to an exponential (in ℓ) factor.

As shown in Fig. 13.1, different bases give different fundamental domains, and the difficulty of SVP and CVP will depend on the shape of the domain. The hardness of CVP inspired the creation of the GGH cryptosystem. Given a lattice $\mathcal{L}$ defined by a public "bad" basis $\boldsymbol{B}$, a plaintext $\boldsymbol{p}$ (which is a small integer vector) is encrypted by adding a lattice vector to it. That way, the decryption consists in finding the closest lattice vector to the ciphertext (the target). The trapdoor is the knowledge of a well-reduced basis $\boldsymbol{R}$, which allows to solve the CVP easily.

In practice, once a basis is determined, solving the CVP can be attempted by using Babai's rounding-off or nearest plane algorithms [2]. Next discussions will focus on the adaptation of the rounding-off to RNS arithmetic. Despite a straightforward principle, constant progress of lattice reduction implies the need of large dimensions ($> 1,000$, [15]) for security purposes, making the computation of a rounding-off burdensome.

13.2.2 Babai's Rounding-Off Algorithm

The main idea of the procedure is to convert the coordinates of a target point from the canonical orthogonal basis to a given lattice basis. The integer part of the coordinates gives the coordinates of a vector of the lattice which is close to the considered point. The target point belongs to the fundamental domain centred in the element of the lattice found. Under certain conditions, the vector found can be the closest one but it is not always the case, as it is described in Fig. 13.2. The secret key in GGH is, for instance, a good basis for which the rounding-off succeeds in returning the closest vector to a ciphertext.

The rounding-off algorithm uses the property that a vector $\boldsymbol{x}$ of a lattice $\mathcal{L}$ can be defined as an integer linear combination of the elements of the basis $\mathcal{R} = (\boldsymbol{r_1}, \ldots, \boldsymbol{r_i}, \ldots, \boldsymbol{r_\ell})$, i.e., $\boldsymbol{x} = \sum_{i=1}^{\ell} \widehat{x_i}\boldsymbol{r_i}$ with $\widehat{x_i} \in \mathbb{Z}$. Hence, the coordinates of such vector in the base $\mathcal{R}$ are integers. The method proposed by Babai is to convert the coordinates from the canonical orthonormal basis $\mathcal{E} = (\boldsymbol{e_1}, \ldots, \boldsymbol{e_i}, \ldots, \boldsymbol{e_\ell})$ to $\mathcal{R}$. We notice that, given $\boldsymbol{R}$, which finally turns out to be the matrice composed of the coordinates of the elements of $\mathcal{R}$ in $\mathcal{E}$, if $\boldsymbol{t} = \sum_{i=1}^{\ell} t_i\boldsymbol{e_i}$, then the product $\boldsymbol{R}^{-1}\boldsymbol{t}$ gives the coordinates of $\boldsymbol{t}$ in $\mathcal{R}$, and the rounding-off $\lfloor \boldsymbol{R}^{-1}\boldsymbol{t} \rceil$ is a close vector of the lattice $\mathcal{L}$, but may be not the closest one within the canonical basis $\mathcal{E}$. Given $\boldsymbol{R}$, the following theorem will enable to describe a set of target vectors for which the rounding-off returns the closest vector.

Theorem 1. *Let's denote* $\rho_{\boldsymbol{R}} = \|\boldsymbol{R}^{-1}\|_\infty = \max\{\sum_{j=1}^{\ell} |(\boldsymbol{R}^{-1})_{i,j}| : i \in [1, \ell]\}$ *the maximum among the* L_1 *norms of the rows of* $\boldsymbol{R}^{-1}$. *If* $0 < \rho_{\boldsymbol{R}} < \frac{1}{2\sigma}$, *with* σ *an integer, and if* $\boldsymbol{p}$ *is a (real) vector such that* $\|\boldsymbol{p}\|_\infty < \sigma$, *then* $\lfloor \boldsymbol{R}^{-1}\boldsymbol{p} \rceil = 0$.

According to the previous theorem, if a point $\boldsymbol{c}$ is the sum of a lattice vector and a small vector $\boldsymbol{p}$ with $\|\boldsymbol{p}\|_\infty < \sigma$, then $\lfloor \boldsymbol{R}^{-1}\boldsymbol{c} \rceil$ returns the closest vector of the lattice, with coordinates in basis $\mathcal{R}$. Thus, in this precise case, $\boldsymbol{R} \lfloor \boldsymbol{R}^{-1}\boldsymbol{c} \rceil$ is the closest of the

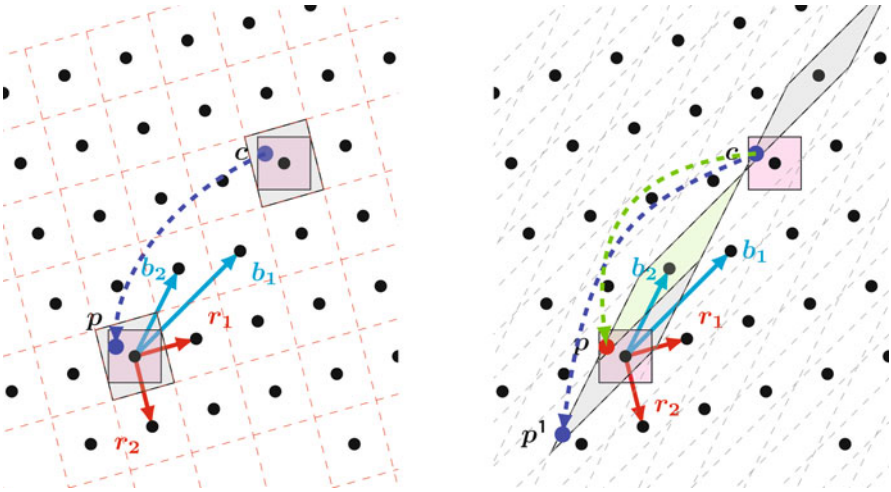

Fig. 13.2 Result of the rounding-off algorithm when performed with two different basis of a same lattice (*black grid*)

lattice, from $\boldsymbol{c}$, in the canonical basis $\mathcal{E}$. In other words, the rounding-off succeeds in solving the CVP for any target vector contained in a ball (for the infinity norm, and in canonical basis $\mathcal{E}$) centred in a lattice vector and with radius σ.

In Fig. 13.2, gray areas correspond to the round-off $\lfloor \boldsymbol{R}^{-1}\boldsymbol{t} \rceil$ and red squares contain points at a distance lower than σ to the lattice point. It can be noticed that the radius σ can be greater when using the basis $(\boldsymbol{r_1}, \boldsymbol{r_2})$ than when working with the basis $(\boldsymbol{b_1}, \boldsymbol{b_2})$. The closest vector to the point $\boldsymbol{p}$ can be recovered in $(\boldsymbol{r_1}, \boldsymbol{r_1})$ by using the rounding-off, whereas it is not possible with the basis $(\boldsymbol{b_1}, \boldsymbol{b_2})$. Figure 13.2 also depicts the subtraction of the rounding-off to the target, providing a point close to the origin: $\boldsymbol{p'} = \boldsymbol{c} - \lfloor \boldsymbol{R}^{-1}\boldsymbol{t} \rceil$. One can see that using the basis $(\boldsymbol{b_1}, \boldsymbol{b_2})$ frequently gives a point which is not close to the origin.

13.2.3 Application to Trapdoor Functions

The previous properties were used to create a trapdoor function, where the secret key is a reduced basis $\mathcal{R}$ and the public key is a bad basis $\mathcal{B}$. In other words, $\boldsymbol{B} = \boldsymbol{RU}$ with $\boldsymbol{U} \in GL_\ell(\mathbb{Z})$ unimodular, and it is such that $\frac{1}{2} < \rho_{\boldsymbol{B}}$ even if it is reduced with classical lattice reduction algorithms. Thus, it is computationally infeasible to recover $\mathcal{R}$ from $\mathcal{B}$. The framework of GGH-like cryptosystems [11] is detailed hereafter:

- `GenKeys`: A lattice $\mathcal{L}$ of dimension ℓ, a positive integer parameter σ, a secret good basis $\mathcal{R}$ verifying $\rho_{\boldsymbol{R}} < \frac{1}{2\sigma}$, a public bad basis $\mathcal{B}$.

- `Encryption`: Given a plaintext $\boldsymbol{p}$ in $[-\sigma, \sigma]^\ell$, pick up a random integer vector $\boldsymbol{k} \in \mathbb{Z}^\ell$, and return the ciphertext $\boldsymbol{c} \leftarrow \boldsymbol{p} + \boldsymbol{Bk}$ ($\boldsymbol{Bk}$ is a random vector of the lattice $\mathcal{L}$).
- `Decryption`: Given $\boldsymbol{c}$, return $\boldsymbol{p} \leftarrow \boldsymbol{c} - \boldsymbol{R}\left\lfloor \boldsymbol{R}^{-1}\boldsymbol{c} \right\rceil$.

The correctness of the scheme is ensured by Theorem 1. Indeed, because $\boldsymbol{R}^{-1}\boldsymbol{Bk} = \boldsymbol{Uk}$ is integer, one has

$$\left\lfloor \boldsymbol{R}^{-1}\boldsymbol{c} \right\rceil = \left\lfloor \boldsymbol{R}^{-1}\boldsymbol{p} \right\rceil + \boldsymbol{Uk}.$$

Given that $\|\boldsymbol{R}^{-1}\boldsymbol{p}\|_\infty \leqslant \rho_{\boldsymbol{R}} \times \sigma < \frac{1}{2}$ by hypothesis on $\boldsymbol{R}$, then $\left\lfloor \boldsymbol{R}^{-1}\boldsymbol{p} \right\rceil = \boldsymbol{0}$. Thus, one obtains $\boldsymbol{c} - \boldsymbol{RUk} = \boldsymbol{p}$.

When an attacker tries to use another basis $\boldsymbol{G}$ (e.g. the output of a lattice reduction algorithm applied to $\boldsymbol{B}$), this one would not be reduced enough to guarantee that $\left\lfloor \boldsymbol{G}^{-1}\boldsymbol{p} \right\rceil = \boldsymbol{0}$. Thus, `Decryption` would fail in returning the plaintext $\boldsymbol{p}$.

We notice that almost the whole scheme boils down to straightforward operations on integer vectors and matrices. The bottleneck is the rounding-off which involves a rational matrix ($\boldsymbol{R}^{-1}$) as well as a rounding-off operation. The goal of next section is to find a (efficient) way to "untie" this knot, making it compatible with RNS arithmetic.

13.3 Transcription in RNS

13.3.1 Some RNS Notations

As RNS are defined by a base of coprime numbers called "moduli", the letter "m" will be used to denote them. Thus, $\mathcal{M}$ represents the RNS base $\{m_1, \ldots, m_n\}$, with $\gcd(m_i, m_j) = 1$ for any $(i, j) \in \{1, \ldots, n\}^2$ and $i \neq j$. To lighten the notations, the product of the moduli of a base $\mathcal{M}$ is denoted by the same letter: $\mathcal{M} = \prod_{i=1}^n m_i$. Moreover, the product of $n-1$ moduli $\frac{\mathcal{M}}{m_i}$ is denoted by $\mathcal{M}_i$. The RNS representation of an integer $a \in [0, \mathcal{M})$ in base $\mathcal{M}$ is denoted by $a_{\mathcal{M}} = (a_1, \ldots, a_n)$ with $a_i = |a|_{m_i} = a \bmod m_i$. Another RNS base, denoted by $\mathcal{M}' = \{m'_1, \ldots, m'_{n'}\}$, will be necessary to implement an RNS modular reduction. It will be chosen coprime to $\mathcal{M}$, i.e., $\gcd(\mathcal{M}, \mathcal{M}') = 1$.

When an integer $a < \mathcal{M}$ is known in a base $\mathcal{M}$, its representation can be extended to another base $\mathcal{M}'$, coprime to $\mathcal{M}$. In particular, if $a < \mathcal{M}'$, then $a_{\mathcal{M}'}$ will completely define a. We denote by $a_{\mathcal{M}'} \leftarrow Bex_{meth}(a_{\mathcal{M}}, \mathcal{M}, \mathcal{M}')$ the extension procedure of the representation of a from $\mathcal{M}$ to $\mathcal{M}'$ made by using a method named *meth*.

There are two families of base extension: one is based on Lagrange's interpolation, as in the constructive proof of the Chinese Remainder Theorem (13.1); another

one is similar to Newton's interpolation (13.2) approach, by using another kind of representation called Mixed Radix System (MRS) [9].

$$x = \sum_{i=1}^{n} \left| x_i \, |\mathcal{M}_i|_{m_i}^{-1} \right|_{m_i} \mathcal{M}_i \bmod \mathcal{M} = \sum_{i=1}^{n} \left| x_i \, |\mathcal{M}_i|_{m_i}^{-1} \right|_{m_i} \mathcal{M}_i - \alpha \mathcal{M}. \tag{13.1}$$

$$\begin{cases} x = \breve{x}_1 + \breve{x}_2.m_1 + \breve{x}_3.m_1 m_2 + \ldots + \breve{x}_n.\prod_{t=1}^{n-1} m_t, \\ \breve{x}_i = (\ldots((x_i - \breve{x}_1)m_1^{-1} - \breve{x}_2)m_2^{-1} - \ldots - \breve{x}_{i-1})m_{i-1}^{-1} \bmod m_i. \end{cases} \tag{13.2}$$

The MRS-based conversion (13.2) involves a transitional positional numeral system. It makes this kind of approach especially burdensome, because it breaks the parallelization properties of RNS. Indeed, for computing the ith coefficient $\breve{x}_i$, one has to know the previous coefficients $\breve{x}_1, \ldots, \breve{x}_{i-1}$ together with the residue x_i. Thus, the computation of the MRS coefficients $(\breve{x}_i)_{1 \leqslant i \leqslant n}$ from the RNS residues $(x_i)_{1 \leqslant i \leqslant n}$ is intrinsically partially sequential, and then computationally burdensome. We will denote by $\breve{x}_{\mathcal{M}} = (\breve{x}_1, \breve{x}_2, \ldots, \breve{x}_n)$ the MRS representation of x associated with the RNS base $\mathcal{M}$.

The previous considerations can be extended to any vector $\boldsymbol{v}$ (resp. matrix $\boldsymbol{A}$). Such object can be represented by its RNS residues, which are denoted by $\boldsymbol{v}_{\mathcal{M}}$ (resp. $\boldsymbol{A}_{\mathcal{M}}$), each of them being a vector (resp. matrix) too.

13.3.2 Adapting Babai's Rounding-Off Algorithm to RNS

In most of the different approaches of lattice-based cryptography like GGH [11] or NTRUsign [12, 13], the search for a closest vector, by using a rounding-off, for instance, is a core operation.

As previously noticed, the rounding-off algorithm involves the use of a rational matrix (the inverse of an integer matrix) and a rounding-off operation. Both are the only parts which are not directly compatible with an integer arithmetic. As Residue Number Systems are dedicated to integer operations, it is necessary to rewrite the rounding-off procedure as a full integer operation. This can be made in the following way:

$$\forall (a, b) \in \mathbb{Z}^2, \left\lceil \frac{a}{b} \right\rfloor = \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor = \frac{(2a + b) - ((2a + b) \bmod b)}{2b}. \tag{13.3}$$

In the present context, $\left\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \right\rfloor$ has to be evaluated, where $\boldsymbol{R}^{-1}$ is a rational matrix and $\boldsymbol{c}$ a vector. $\boldsymbol{R}^{-1}$ is the invert of the integer matrix $\boldsymbol{R}$. Thus $\widehat{\boldsymbol{R}} = \boldsymbol{R}^{-1} \times d$ is an integer matrix, where d denotes $\det \boldsymbol{R}$. Consequently, by denoting $\boldsymbol{d} = (d, \ldots, d) \in \mathbb{Z}^{\ell}$, and by using (13.3), one can write:

$$\left\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \right\rfloor = \left\lceil \frac{\widehat{\boldsymbol{R}}\boldsymbol{c}}{d} \right\rfloor = \frac{(2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}) - ((2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}) \bmod (2d))}{2d}. \tag{13.4}$$

The evaluation of $(2\widehat{\boldsymbol{Rc}} + \boldsymbol{d})$ is easily doable in RNS. As we subtract $((2\widehat{\boldsymbol{Rc}} + \boldsymbol{d}) \bmod (2d))$ to this value, the division by $2d$ is exact and can be done by multiplying by the inverse of $2d$ in the underlying RNS base (which has to be coprime with $2d$). Thus, the evaluation of $((2\widehat{\boldsymbol{Rc}} + \boldsymbol{d}) \bmod (2d))$ remains the main obstacle to an RNS rounding-off, especially because the modular reduction has to be completed. To overcome this issue, it is possible to use the adaptation to RNS of Montgomery's modular reduction. This is the classical way to reduce a value in RNS.

Algorithm 1 $\texttt{RnsModRed}(x_{\mathcal{M}\cup\mathcal{M}'}, p, \mathcal{M}, \mathcal{M}')$

Require: $\mathcal{M}$, $\mathcal{M}'$ with $\gcd(\mathcal{M}, \mathcal{M}'p) = 1$, residues $x_{\mathcal{M}\cup\mathcal{M}'}$ of $x < (1-\Delta)\mathcal{M}.p$ (for some $\Delta \in [0,1)$), $(2-\Delta+\lambda_{\texttt{meth1}})p < (1-\alpha_{\texttt{meth2}})\mathcal{M}'$.
Ensure: residues $s_{\mathcal{M}\cup\mathcal{M}'}$ of $s < p$, $s \equiv x.\mathcal{M}^{-1} \bmod p$.

1: $q_{\mathcal{M}} \leftarrow \left|-x.p^{-1}\right|_{\mathcal{M}}$ ▷ in parallel in base $\mathcal{M}$
2: $\widehat{q}_{\mathcal{M}'} \leftarrow \mathrm{Bex}_{\texttt{meth1}}(q_{\mathcal{M}}, \mathcal{M}, \mathcal{M}')$ ▷ $\widehat{q} < (1+\lambda_{meth1})\mathcal{M}$
3: $t_{\mathcal{M}'} \leftarrow \left|x + \widehat{q}.p\right|_{\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M}'$
4: $s_{\mathcal{M}'} \leftarrow \left|t.\mathcal{M}^{-1}\right|_{\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M}'$; $s < (2-\Delta+\lambda_{\texttt{meth1}})p$
5: $s_{\mathcal{M}} \leftarrow \mathrm{Bex}_{\texttt{meth2}}(s_{\mathcal{M}'}, \mathcal{M}', \mathcal{M}, \alpha_{\texttt{meth2}})$
6: If $s_{\mathcal{M}\cup\mathcal{M}'} > p$ then $s_{\mathcal{M}\cup\mathcal{M}'} \leftarrow s_{\mathcal{M}\cup\mathcal{M}'} - p_{\mathcal{M}\cup\mathcal{M}'}$ ▷ comparison: requires to switch from RNS to a positional system (e.g. MRS)
7: **return** $s_{\mathcal{M}\cup\mathcal{M}'}$

About RNS Montgomery Reduction

Algorithm 1 implements the RNS Montgomery modular reduction. The idea is, for reducing x modulo p, to compute $\frac{x+qp}{\mathcal{M}}$ where $q \leftarrow |-xp^{-1}|_{\mathcal{M}}$ is computed in the main base $\mathcal{M}$. An auxiliary base $\mathcal{M}'$, and two base extensions $\mathrm{Bex}_{\texttt{meth1}}$, $\mathrm{Bex}_{\texttt{meth2}}$, allow to handle the division by $\mathcal{M}$ which cannot be performed in the base $\mathcal{M}$. For our purpose, the main issue of such modular reduction is that it generally returns an almost (but possible not completely) reduced value (i.e. x plus a small multiple of p).

The functions $\mathrm{Bex}_{\texttt{meth1}}(q_{\mathcal{M}}, \mathcal{M}, \mathcal{M}')$ and $\mathrm{Bex}_{\texttt{meth2}}(s_{\mathcal{M}'}, \mathcal{M}', \mathcal{M}, \alpha_{meth2})$ in Algorithm 1 correspond to different conversion techniques between two RNS bases. Parameters $\lambda_{\texttt{meth1}}$ and $\alpha_{\texttt{meth2}}$ are related to the facts that Bex_{meth1} may provide an uncompletely reduced value modulo $\mathcal{M}$ whereas $\mathrm{Bex}_{\texttt{meth2}}$ has to be an exact conversion (which can be guaranteed if $s < (1-\alpha_{\texttt{meth2}})\mathcal{M}'$). More precisely, residues $s_{\mathcal{M}}$ at line 4 are those of an integer $s = \frac{x+\widehat{q}.p}{\mathcal{M}} < (2-\Delta+\lambda_{\texttt{meth1}})p$. To limit the number of comparisons of s to p, the method $\texttt{meth1}$ can be carefully chosen so that $\lambda_{\texttt{meth1}} \leqslant \Delta$. Then, should the output be completely reduced, only one comparison would be required. But it has to be noticed that such comparison remains a bottleneck when a complete modular reduction modulo p is required, since it cannot be performed in RNS directly.

It is not mandatory that Bex_{meth1} provides a complete result in the sense that a multiple of $\mathcal{M}$, up to $\lambda_{meth1}\mathcal{M}$, may appear in the output of the conversion. This is because, since $\widehat{q}_{\mathcal{M}'}$ is multiplied by p, the output remains correct modulo p. Because the second conversion must be exact (i.e. no multiple of $\mathcal{M}'$ should appear in the residues of the output in base $\mathcal{M}$), the base $\mathcal{M}'$ has to be large enough to completely represent the result. In further parts, different approaches and their consequences will be discussed [4, 5].

From now on, all the required operators for evaluating the expression (13.4) are known. Algorithm 1 is applied on each component of $(2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d})$ for evaluating $((2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}) \bmod (2d))$. Thus, Algorithm 2 implements the proposed RNS Babai's rounding-off.

Algorithm 2 $\texttt{RnsBabaiRO}(\boldsymbol{c}_{\mathcal{M}\cup\mathcal{M}'}, \mathcal{R}, \mathcal{M}, \mathcal{M}')$

Require: $d = \det \boldsymbol{R}$, $\boldsymbol{d} = (d, \ldots, d)$, $\widehat{\boldsymbol{R}} = \boldsymbol{R}^{-1} \times d$, $\widetilde{\boldsymbol{R}} = 2 \times \boldsymbol{R}^{-1} \times d \times \mathcal{M} \bmod (2d)$, and $\widetilde{\boldsymbol{d}} = \boldsymbol{d} \times \mathcal{M} \bmod (2d)$, $\widetilde{\boldsymbol{d}} = (\widetilde{d}, \ldots, \widetilde{d})$.

Ensure: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'} = \left\lceil \frac{\widehat{\boldsymbol{R}}\boldsymbol{c}}{d} \right\rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor$ in $\mathcal{M} \cup \mathcal{M}'$.

1: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'} \leftarrow (2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d})$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}'$
2: $\widetilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'} \leftarrow (\widetilde{\boldsymbol{R}}\boldsymbol{c} + \widetilde{\boldsymbol{d}})$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}'$
3: $\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'} \leftarrow \texttt{RnsModRed}(\widetilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'}, 2d, \mathcal{M}, \mathcal{M}')$ ▷ $\boldsymbol{g} = \lfloor \boldsymbol{f} \rfloor_{2d}$
4: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'} \leftarrow (\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'} - \boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'}).(2d)^{-1}_{\mathcal{M}\cup\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}'$
5: **return** $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'}$

The complexity of Algorithm 2 mainly depends on the operation `RnsModRed`. Next, a first approach is suggested, in which $\text{Bex}_{\texttt{meth2}}$ in `RnsModRed` relies on the use of an intermediate MRS. The main advantages of this type of conversion is that it is always exact and it naturally enables to make comparison (thanks to the positional nature of MRS). However, such method is costly. Thus, a second full RNS approach will be studied. It is based on CRT-based conversions, which in the present context may introduce errors. But an efficient correction technique, available when some conditions on the lattice are satisfied, will be presented.

13.4 A Mixed RNS-MRS Approach of Babai's Rounding-Off Algorithm

In this section, a variant of Algorithm 1 is considered, in which the second base extension involves MRS. It allows to perform a complete modular reduction as well as the final comparison which is mandatory in the context of the rounding-off Algorithm 2. About the first base extension, two variants will be detailed thereafter: a (burdensome) MRS-based one and a (fast) CRT-based other one. If the latter is more efficient, it generates some overflows which have to be managed. For this purpose, a new variant of RNS modular reduction is proposed in order to keep the benefit of such fast first base extension.

13.4.1 MRS Approach for Bex_{meth1}

The approach depicted in this part was first described in [4]. The performances of implementations on CPU or GPU have been investigated in [16].

In order to output the same value in both $\mathcal{M}$ and $\mathcal{M}'$, Bex_{meth2} must be exact. Moreover, because final comparisons are necessary, choosing Bex_{mrs} for this second base extension appears quite natural. In particular, this choice implies that $\alpha_{\texttt{meth2}} = \alpha_{mrs} = 0$, i.e., any value modulo $\mathcal{M}'$ is always exactly extended.

About the first extension, it has been previously noticed that it can generate an extra term, multiple of $\mathcal{M}$ and no greater than $\lambda_{meth1}\mathcal{M}$ (without changing the correctness of s modulo p in $\mathcal{M}'$). In this case, the output of the extension Bex_{meth1} verifies $\widehat{q} < (1+\lambda_{meth1})\mathcal{M}$. By using Bex_{mrs}, the output is guaranteed to be equal to q and then lower than $\mathcal{M}$ (i.e. in this case $\lambda_{meth1} = \lambda_{mrs} = 0$). The main consequence is that the result s in base $\mathcal{M}'$ now satisfies $s < 2p$. Thus, only one comparison has to be performed within the second base extension.

Algorithm 3 $\texttt{MrsRnsModRed}(x_{\mathcal{M}\cup\mathcal{M}'}, \left|(\mathcal{M})^{-1}\right|_{\mathcal{M}'}, p, \mathcal{M}, \mathcal{M}')$

Require: $\mathcal{M}$, $\mathcal{M}'$ with $\gcd(\mathcal{M}, \mathcal{M}'.p) = 1$, residues $x_{\mathcal{M}\cup\mathcal{M}'}$ of $x < (1-\Delta)\mathcal{M}.p$ (for some $\Delta \in [0,1)$), $(2-\Delta)p < \mathcal{M}'$.
Ensure: residues $s_{\mathcal{M}\cup\mathcal{M}'}$ of $s < p$, $s \equiv x.\mathcal{M}^{-1} \bmod p$.
1: $q_{\mathcal{M}} \leftarrow \left|-x.p^{-1}\right|_{\mathcal{M}}$ ▷ in parallel in base $\mathcal{M}$
2: $\widehat{q}_{\mathcal{M}'} \leftarrow \mathrm{Bex}_{mrs}(q_{\mathcal{M}}, \mathcal{M}, \mathcal{M}')$ ▷ $\widehat{q} < \mathcal{M}$ and $\widehat{q} = q$
3: $t_{\mathcal{M}'} \leftarrow \left|x + \widehat{q}.p\right|_{\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M}'$
4: $s_{\mathcal{M}'} \leftarrow \left|t.\mathcal{M}^{-1}\right|_{\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M}'$; $s < (2-\Delta)p$
5: $s_{\mathcal{M}}, \check{s}_{\mathcal{M}'} \leftarrow \mathrm{Bex}_{mrs}(s_{\mathcal{M}'}, \mathcal{M}', \mathcal{M}, \alpha_{mrs} = 0)$
6: If $\check{s}_{\mathcal{M}'} > \check{p}$ then $s_{\mathcal{M}\cup\mathcal{M}'} \leftarrow s_{\mathcal{M}\cup\mathcal{M}'} - p_{\mathcal{M}\cup\mathcal{M}'}$ ▷ comparison in MRS
7: **return** $s_{\mathcal{M}\cup\mathcal{M}'}$

The drawback of the modular reduction depicted in Algorithm 3 is due to the MRS extensions. Even if the need of a final comparison remains mandatory (then obliging to adopt such kind of extension for $\mathrm{Bex}_{\texttt{meth2}}$), it is possible to adopt another strategy for the first extension. By using a CRT-based extension (13.1), $\widehat{q}_{\mathcal{M}'}$ represents an integer which is congruent to $q_{\mathcal{M}} \bmod \mathcal{M}$, but not necessarily equal to it. The best efficiency occurs when the sum in (13.1) is not reduced modulo $\mathcal{M}$ at all. In this case, λ_{meth1} can be any integer in $[0, n-1]$. Hence, s would verify $s < (1-\Delta+n)p$ in this case, and a simple comparison to p would not be sufficient for a complete reduction. In next part, a way to deal with the issues introduced by this kind of approach is discussed.

13.4.2 New Approach for Bex_{meth1}

Equation (13.1) enables us to recover an integer from its residues in base $\mathcal{M}$. If one wants to use it for constructing a base extension procedure, an important issue appears. Indeed, obtaining a complete reduction modulo $\mathcal{M}$ is not straightforward when staying in RNS only. When the reduction modulo $\mathcal{M}$ may be not performed at all, one obtains the following quantity:

$$\sum_{i=1}^{n} \left| x_i \left| \mathcal{M}_i \right|_{m_i}^{-1} \right|_{m_i} \mathcal{M}_i = x + \alpha\mathcal{M}, \alpha \in [0, n-1]. \tag{13.5}$$

The evaluation of the coefficient α has always been a challenge. Shenoy and Kumaresan [29] proposed an approach involving an extra redundant modulus to recover α. The drawback, for our purpose, is that the knowledge of the residue x modulo this extra modulus is necessary. Yet, the first extension Bex_{meth1} extends q which is computed modulo $\mathcal{M}$, implying that only the residues of q in $\mathcal{M}$ are known. Thus, this kind of approach is impossible for Bex_{meth1}.

To efficiently recover α, Posch and Posch [23] suggested to evaluate the approximation $\lfloor \sum_{i=1}^{n} \left| x_i \left| \mathcal{M}_i \right|_{m_i}^{-1} \right|_{m_i} \frac{1}{m_i} \rfloor$, which allows to obtain either α or $\alpha - 1$. This may not be an issue in general, but for our purpose it is necessary to have $\widehat{q} < (1 + \lambda_{meth1})\mathcal{M}$ with $\lambda_{meth1} < \Delta < 1$. Indeed, if one can have $\widehat{q}_{\mathcal{M}'} = q_{\mathcal{M}} + \mathcal{M}$ for any possible value $q_{\mathcal{M}}$ in $[0, \mathcal{M})$, then $s_{\mathcal{M}'}$ could be greater than $2p$ in certain cases.

To overcome this issue, another approach, which was introduced in [4], is possible, where $\widehat{q}_{\mathcal{M}'}$ is reduced inside the base $\mathcal{M}'$ by using a small Montgomery reduction based on an extra modulus $\tilde{m}$. Algorithm 4 depicts this new modular reduction.

As in Shenoy and Kumaresan's extension, it is not possible to get efficiently the exact residue of q modulo $\tilde{m}$. So, during the first extension, $\tilde{m}$ is considered as a part of the output base, besides $\mathcal{M}'$. It means that $\widehat{q}_{\tilde{m}}$ is obtained with the same error α as well modulo $\tilde{m}$ as modulo $\mathcal{M}'$.

For the sake of efficiency, Bex_{meth1} may only compute $\widehat{q}$ as being the sum $\sum_{i=1}^{n} |q_i \left| \mathcal{M}_i \right|_{m_i}^{-1} \Big|_{m_i} \mathcal{M}_i$ in the base $\mathcal{M}' \cup \{\tilde{m}\}$. Thus, the output verifies $\widehat{q}_{\mathcal{M}' \cup \tilde{m}} < (1 + \alpha)\mathcal{M} \leqslant n\mathcal{M}$.

The next step is the computation of $t = \frac{x + \widehat{q}.p}{\mathcal{M}}$ both in $\mathcal{M}'$ and modulo $\tilde{m}$. In particular, one easily shows that $t < (2 + \alpha)p \leqslant (n + 1)p$. This bound implies that, in order to avoid any loss of information, it is necessary to have $\mathcal{M}' \times \tilde{m} > (2 + \alpha)p$.

Now, as soon as $\tilde{m}$ is chosen such that $2 + \alpha \leqslant n + 1 < \tilde{m}$, a second Montgomery reduction using $\tilde{m}$ will ensure $s_{\mathcal{M}'} < 2p$. More precisely, the quantity $\tilde{q} = t_{\tilde{m}} |-p|_{\tilde{m}}^{-1} \bmod \tilde{m}$ is computed in the base $\{\tilde{m}\}$. Then, the reduction is achieved in base $\mathcal{M}'$ by computing $s_{\mathcal{M}'} \leftarrow \left| (t + \tilde{q}p).(\tilde{m})^{-1} \right|_{\mathcal{M}'}$. When $\tilde{m}$ is carefully chosen, i.e., such that $\tilde{m} < \min\{m \in \mathcal{M}'\}$, then one achieves a full extension of $\tilde{q}$ toward $\mathcal{M}'$ through a simple "copy-paste" procedure.

A sufficient size for $\tilde{m}$ depends on the fact that $s_{\mathcal{M}'} < 2p$ is expected to be verified. For that purpose, it is easy to establish that

$$\frac{t + \tilde{q}p}{\tilde{m}} < \frac{(n+1)p + \tilde{m}p}{\tilde{m}} = \left(1 + \frac{n+1}{\tilde{m}}\right)p.$$

Hence, it is sufficient to have $\tilde{m} \geqslant n+1$. One will notice that this bound is available even by taking $\Delta = 0$, i.e., if $x < \mathcal{M}p$. Finally, Algorithm 4 implements this new RNS Montgomery reduction.

Algorithm 4 `NewRnsModRed`$(x_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}, p, \mathcal{M}, \mathcal{M}', \tilde{m})$

Require: $x < \mathcal{M}.p$ and $p, \mathcal{M}, \mathcal{M}', \tilde{m}$ pairwise coprimes;, $n+1 \leqslant \tilde{m}$ and $(1 + \frac{n+1}{\tilde{m}})p < \mathcal{M}'$.
Ensure: residues $s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$ of $s = x(\tilde{m}\mathcal{M})^{-1} \bmod p$.

1: $q_{\mathcal{M}} \leftarrow \left|-x.p^{-1}\right|_{\mathcal{M}}$ ▷ in parallel in base $\mathcal{M}$
2: $\widehat{q} \leftarrow \sum_{m\in\mathcal{M}} \left|q_i\,|\mathcal{M}_i|_{m_i}^{-1}\right|_{m_i} \times \mathcal{M}_i$ in $\mathcal{M}' \cup \{\tilde{m}\}$ ▷ $\widehat{q} < n\mathcal{M}$
3: $(t_{\mathcal{M}'}, t_{\tilde{m}}) \leftarrow \left(\left|(x + \widehat{q}.p)(\mathcal{M})^{-1}\right|_{\mathcal{M}'}, \left|(x + \widehat{q}.p)(\mathcal{M})^{-1}\right|_{\tilde{m}}\right)$ ▷ in parallel
4: $\tilde{q} \leftarrow t_{\tilde{m}}\,|-p|_{\tilde{m}}^{-1} \bmod \tilde{m}$
5: Extension of $\tilde{q}$ toward $\mathcal{M}'$ ▷ simple duplication
6: $s_{\mathcal{M}'} \leftarrow \left|(t + \tilde{q}p).(\tilde{m})^{-1}\right|_{\mathcal{M}'}$ ▷ in parallel in base $\mathcal{M}'$
7: $s_{\mathcal{M}}, s_{\tilde{m}}, \breve{s}_{\mathcal{M}'} \leftarrow \mathrm{Bex}_{mrs}(s_{\mathcal{M}'}, \mathcal{M}', \mathcal{M}, \alpha_{mrs} = 0)$
8: If $\breve{s}_{\mathcal{M}'} > \breve{p}_{\mathcal{M}'}$ then $s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} - p_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$
9: **return** $s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$

To use `NewRnsModRed` within the Babai's rounding-off, Algorithm 2 has to be slightly modified so as to integrate the new modulus $\tilde{m}$. In particular, the Montgomery representation with respect to $\tilde{m}\mathcal{M}$ is handled within the precomputations. Algorithm 5 depicts the whole procedure.

Algorithm 5 `NewRnsBabaiRO`$(\boldsymbol{c}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}, \mathcal{R}, \mathcal{M}, \mathcal{M}', \tilde{m})$

Require: $d = \det \boldsymbol{R}$, $\boldsymbol{d} = (d, \ldots, d)$, $\widehat{\boldsymbol{R}} = \boldsymbol{R}^{-1} \times d$, $\tilde{\boldsymbol{R}} = 2 \times \boldsymbol{R}^{-1} \times d \times \mathcal{M} \times \tilde{m} \bmod (2d)$, and $\tilde{d} = d \times \mathcal{M} \times \tilde{m} \bmod (2d)$, $\tilde{\boldsymbol{d}} = (\tilde{d}, \ldots, \tilde{d})$.
Ensure: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} = \left\lceil \frac{\widehat{\boldsymbol{R}}\boldsymbol{c}}{d} \right\rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor$ in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$.

1: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow (2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d})$ ▷ in parallel in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$
2: $\tilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow (\tilde{\boldsymbol{R}}\boldsymbol{c} + \tilde{\boldsymbol{d}})$ ▷ in parallel in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$
3: $\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow$ `NewRnsModRed`$(\tilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}, 2d, \mathcal{M}, \mathcal{M}', \tilde{m})$
4: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow (\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} - \boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}).(2d)^{-1}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$ ▷ in parallel in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$
5: **return** $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$

13.5 Adapting Babai's Rounding-Off Algorithm to a Pure RNS Arithmetic

13.5.1 Main Idea

The efficiency of the RNS-MRS variant of Babai's rounding-off proposed so far is dramatically limited by the MRS-based extensions involved in the procedure `RnsModRed` (cf. Algorithm 1). Such kind of extension was used because of the need for a final comparison after the second base extension Bex_{meth2} in order to obtain an output which is fully reduced $2d$. Without such comparison, it is direct to see from (13.4) that the vector $\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'}$ at line 3 of Algorithm 2 may contain an error of the form $2d\boldsymbol{e}$ due to a possible incomplete reduction modulo $2d$. In particular, this is the case with Algorithm 6 presented in Sect. 13.5.2.

To summarize, an incomplete modular reduction at line 3 of Algorithm 2 would provide

$$\boldsymbol{g} = |\boldsymbol{f}|_{2d} + 2d\boldsymbol{e}. \tag{13.6}$$

In this case, the evaluation of line 4 will provide the following result:

$$\frac{\boldsymbol{f} - (\boldsymbol{g} + 2d\boldsymbol{e})}{2d} = \frac{\boldsymbol{f} - |\boldsymbol{f}|_{2d}}{2d} - \boldsymbol{e} = \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor - \boldsymbol{e}. \tag{13.7}$$

Previously, it has been possible to use an efficient CRT-based extension for Bex_{meth1}. By introducing an extra modulus $\tilde{m}$, it is possible to have only one comparison to make. This is the purpose of Algorithm 4. However, it is difficult to do better. By its own design, the Montgomery reduction outputs a value which may be greater than the modulus. So, instead of looking for a way to modify this reduction scheme in order to guarantee a full reduction, it could be more fruitful to look for another method for correcting such possible incomplete reduction. Consequently, from now on, the new strategy is the following one: first, the MRS-based extension is simply replaced by a fast, efficient, but approximate, CRT-based extension; second, an efficient way to correct the error $\boldsymbol{e}$ has to be find. The idea of the correction method is highlighted in the following remark.

Remark 1. The principal idea is to deal with multiples of an integer γ greater than $2\|\boldsymbol{e}\|_\infty$. If it is possible to turn $(\boldsymbol{f} - |\boldsymbol{f}|_{2d})(2d)^{-1}$ into a multiple of γ, then we would obtain $(\boldsymbol{f} - \boldsymbol{g})(2d)^{-1} \bmod c\, \gamma = (\boldsymbol{f} - |\boldsymbol{f}|_{2d})(2d)^{-1} - \boldsymbol{e} \bmod c\, \gamma = -\boldsymbol{e}$ (where modc represents a modular reduction with residues in $(-\frac{\gamma}{2}, \frac{\gamma}{2}]$). As a consequence, $\boldsymbol{e}$ could be recovered and then corrected.

To achieve the previous idea, $\gamma\boldsymbol{f}$ is considered in Algorithm 2 instead of $\boldsymbol{f}$. In this case, the algorithm aims at computing the quantity $\left\lceil \gamma \frac{\widehat{\boldsymbol{Rc}}}{d} \right\rfloor = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor$ in $\mathcal{M} \cup \mathcal{M}'$. By definition, $\boldsymbol{c} = \boldsymbol{p} + \boldsymbol{Bk}$ (cf. Sect. 13.2.3) and $\boldsymbol{B} = \boldsymbol{RU}$, for $\boldsymbol{U}$ in $GL_\ell(\mathbb{Z})$. Thus, one gets

$$\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor = \lceil \gamma \boldsymbol{R}^{-1}(\boldsymbol{p} + \boldsymbol{Bk}) \rfloor = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} + \gamma \boldsymbol{R}^{-1}\boldsymbol{Bk} \rfloor = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor + \gamma \boldsymbol{Uk}.$$

Besides, the following equalities hold too:

$$\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{p} + \boldsymbol{R}^{-1}\boldsymbol{B}\boldsymbol{k} \rfloor = \boldsymbol{U}\boldsymbol{k}.$$

Thus, it comes that

$$\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor + \gamma \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor .$$

Now, the result of line 4 in Algorithm 2 is considered with error $\boldsymbol{e}$, and with input $\gamma \boldsymbol{f}$ instead of $\boldsymbol{f}$ (cf. (13.7). In this case, the quantity to be corrected is the following one:

$$\boldsymbol{e}' = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor - \boldsymbol{e}. \tag{13.8}$$

To sum up, the output of Algorithm 2 becomes

$$\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor = \gamma \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor + \boldsymbol{e}'.$$

Remark 2. If $\gamma > 2\|\boldsymbol{e}'\|_\infty$, then $\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor \bmod c\, \gamma = \boldsymbol{e}'$.

The problem is to evaluate the size of γ, through the bound in Remark 2. Ideally, it should be small for the purpose of efficiency. Indeed, if γ can be set as a single modulus base, the computation of the centred remainder (requiring a comparison with $\gamma/2$) is straightforward. Actually, because of the term $\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor$ in the definition of $\boldsymbol{e}'$ (13.8), the value of γ will depend on the lattice $\mathcal{L}$ and more precisely on the geometrical shape of the secret basis $\mathcal{R}$ (cf. Sect. 13.5.3).

Because the term $\boldsymbol{e}$ in (13.8) is not critical with regard to the consideration on the size of γ (this one will mostly depend on the norm of $\lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor$), the use of a fast RNS modular reduction is preferred, as described next.

13.5.2 Fast RNS Modular Reduction

This approach was first published at ARITH conference in 2001 [3]. The first CRT-based extension is made roughly, i.e., without any correction of the term $\alpha\mathcal{M}$ in (13.1). In this case, it provides $\widehat{q} = q + \lambda\mathcal{M}'$ in $\mathcal{M}' \cup \tilde{m}$ with λ an integer in $[0, n-1]$ ($n = \mathcal{M}^{\#}$). Then, the value s computed in $\mathcal{M}' \cup \tilde{m}$ satisfies $s < (2+\lambda)p \leqslant (1+n)p$. In particular, this inequality constrains the size of $\mathcal{M}'$: $(1+n)p \leqslant \mathcal{M}'$. The value s is then correctly extended to $\mathcal{M}$ by using Shenoy and Kumaresan's technique. Thus, the output of the reduction is the same in $\mathcal{M}$ and in $\mathcal{M}' \cup \tilde{m}$.

Shenoy and Kumaresan's idea to get an efficient exact extension of $s_{\mathcal{M}'}$ is to add an extra modulus $\tilde{m}$ besides $\mathcal{M}'$. Assuming that $s_{\tilde{m}}$ has also been computed, this residue enables an easy computation of the coefficient α in the following formula:

$$s = \sum_{i=1}^{n'} \left| s'_i (\mathcal{M}'_i)^{-1} \right|_{m'_i} \mathcal{M}'_i - \alpha\mathcal{M}' \in [0, \mathcal{M}').$$

α represents the reduction of the sum modulo $\mathcal{M}'$. Thus, it is an integer in $[0, n'-1]$. By inverting the previous equation in the ring $\mathbb{Z}_{\tilde{m}}$, $|\alpha|_{\tilde{m}}$ can be recovered. Consequently, if $\tilde{m} \geqslant n'$, then $|\alpha|_{\tilde{m}} = \alpha$.

$$\left| \left(\sum_{i=1}^{n'} \left| s'_i (\mathcal{M}'_i)^{-1} \right|_{m'_i} \mathcal{M}'_i - s_{\tilde{m}} \right) (\mathcal{M}')^{-1} \right|_{\tilde{m}} = |\alpha|_{\tilde{m}} = \alpha.$$

The fast RNS modular reduction is implemented in Algorithm 6. Comparatively to a reduction involving sequential MRS-based extensions, the CRT-based extensions within this fast reduction are simple matrix–vector multiplications. Thus, it is highly parallelizable.

Algorithm 6 `FastRnsModRed`$(x_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}, p, \mathcal{M}, \mathcal{M}', \tilde{m})$

Require: $x < \mathcal{M}.p$ and $p, \mathcal{M}, \mathcal{M}', \tilde{m}$ pairwise coprimes, $(1+n)p \leqslant \mathcal{M}'$, $n' \leqslant \tilde{m}$.
Ensure: residues $s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$ of $s \equiv x\mathcal{M}^{-1} \bmod p$ with $s < (1+n)p$.

1: $q_{\mathcal{M}} \leftarrow \left| -x.p^{-1} \right|_{\mathcal{M}}$ ▷ in parallel in base $\mathcal{M}$

2: $\begin{cases} \widehat{q}_{\mathcal{M}'} \leftarrow \sum_{m_i \in \mathcal{M}} \left| q_i \, |\mathcal{M}_i|_{m_i}^{-1} \right|_{m_i} \mathcal{M}_i \text{ in } \mathcal{M}' \\ \widehat{q}_{\tilde{m}} \leftarrow \sum_{m_i \in \mathcal{M}} \left| q_i \, |\mathcal{M}_i|_{m_i}^{-1} \right|_{m_i} \mathcal{M}_i \bmod \tilde{m} \end{cases}$ ▷ fast base conversion $\widehat{q} < n\mathcal{M}$

3: $\begin{cases} s_{\mathcal{M}'} \leftarrow (x + \widehat{q}.p).\, |\mathcal{M}|_{\mathcal{M}'}^{-1} \text{ in } \mathcal{M}' \\ s_{\tilde{m}} \leftarrow (x + \widehat{q}.p).\, |\mathcal{M}|_{\tilde{m}}^{-1} \bmod \tilde{m} \end{cases}$ ▷ $s < (1+n)p$

4: $\begin{cases} s_{\mathcal{M}} \leftarrow \sum_{m'_i \in \mathcal{M}'} \left| s_{m'_i} \, |\mathcal{M}'_i|_{m'_i}^{-1} \right|_{m'_i} \mathcal{M}'_i \text{ in } \mathcal{M} \\ \widehat{s}_{\tilde{m}} \leftarrow \sum_{m'_i \in \mathcal{M}'} \left| s_{m'_i} \, |\mathcal{M}'_i|_{m'_i}^{-1} \right|_{m'_i} \mathcal{M}'_i \bmod \tilde{m} \end{cases}$ ▷ fast base conversion

5: $\alpha \leftarrow \left(\widehat{s}_{\tilde{m}} - s_{\tilde{m}}\right) \left| \mathcal{M}'^{-1} \right|_{\tilde{m}} \bmod \tilde{m}$ ▷ Shenoy-Kumaresan [29]
6: $s_{\mathcal{M}} \leftarrow s_{\mathcal{M}} - \alpha\mathcal{M}'$ in $\mathcal{M}$
7: **return** $s_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$

13.5.3 Conditions for Correction

For making the previous correction technique possible, γ has to match with Remark 2 and the "Babai conditions" in Theorem 1.

$$\text{Remark 2} \rightarrow \begin{cases} \boldsymbol{e}' = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor - \boldsymbol{e} \\ \gamma > 2\|\boldsymbol{e}'\|_\infty \end{cases} \tag{13.9}$$

$$\text{Babai conditions} \rightarrow \begin{cases} \rho_{\boldsymbol{R}} = \|\boldsymbol{R}^{-1}\|_\infty \\ \|\boldsymbol{p}\|_\infty < \sigma \\ 0 < \rho_{\boldsymbol{R}}\sigma < \frac{1}{2} - \varepsilon, \text{ for some } 0 < \varepsilon < \frac{1}{2} \end{cases} \tag{13.10}$$

From (13.9) and (13.10), it can be deduced that

$$\left\| \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor \right\|_\infty \leqslant \left\| \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \right\|_\infty + \frac{1}{2} < \gamma\rho_{\boldsymbol{R}}\sigma + \frac{1}{2} < \gamma\left(\frac{1}{2} - \varepsilon\right) + \frac{1}{2}.$$

In the context of Algorithm 6, the error $\boldsymbol{e}$ is due to the fast first base extension (at line 2). Thus, $\|\boldsymbol{e}\|_\infty \leqslant n$. So, $\|\boldsymbol{e}'\|_\infty < \gamma(\frac{1}{2} - \varepsilon) + \frac{1}{2} + n$. Therefore, a sufficient condition for γ is given by

$$\gamma \geqslant \gamma(1 - 2\varepsilon) + 1 + 2n > 2\|\boldsymbol{e}'\|_\infty \Rightarrow \gamma \geqslant \frac{1 + 2n}{2\varepsilon}.$$

Theorem 2. *Let $\mathcal{R}$ a reduced basis of a lattice $\mathcal{L}$ verifying $0 < \rho_{\boldsymbol{R}}\sigma < \frac{1}{2} - \varepsilon$ with $\rho_{\boldsymbol{R}} = \|\boldsymbol{R}^{-1}\|_\infty$. Let $\boldsymbol{p}$ be a plaintext, i.e., $\|\boldsymbol{p}\|_\infty \leqslant \sigma$, and $\boldsymbol{c}$ an encryption of $\boldsymbol{p}$. Let γ be an integer and $\boldsymbol{e}' = \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{p} \rfloor - \boldsymbol{e}$, with $\|\boldsymbol{e}\|_\infty \leqslant n$. Then*

$$\gamma \geqslant \frac{1 + 2n}{2\varepsilon} \Rightarrow \lceil \gamma \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor \operatorname{modc} \gamma = \boldsymbol{e}'. \tag{13.11}$$

Based on Theorem 2, Algorithm 7 implements Babai's rounding-off with the γ-correction technique. Its correctness is stated by the following theorem.

Theorem 3. *Let's consider Algorithm 7 applied to an input $\boldsymbol{c}$ (in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$) encrypting $\boldsymbol{p}$ with $\|\boldsymbol{p}\|_\infty \leqslant \sigma$, and with a reduced basis $\boldsymbol{R}$ verifying $0 < \rho_{\boldsymbol{R}}\sigma < \frac{1}{2} - \varepsilon$. Let $\mathcal{M} > \gamma(\|\boldsymbol{c}\|_1 + 1)$. If $\gamma \geqslant \frac{1+2n}{2\varepsilon}$, then Algorithm 7 outputs $\boldsymbol{p}$ in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$.*

Proof. Given the precomputations $\widehat{\boldsymbol{R}}$, $\tilde{\boldsymbol{R}}$, and $\tilde{\boldsymbol{d}}$, the term $\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma}$ at line 3 is the output of Montgomery's reduction of $(2\gamma\widehat{\boldsymbol{R}} + \boldsymbol{d})$ modulo $(2d)$:

$$\begin{aligned} \boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} &= \frac{\gamma\tilde{\boldsymbol{R}}\boldsymbol{c} + \tilde{\boldsymbol{d}} + 2d(| - (\gamma\tilde{\boldsymbol{R}}\boldsymbol{c} + \tilde{\boldsymbol{d}})(2d)^{-1}|_{\mathcal{M}} + \tilde{\boldsymbol{e}}\mathcal{M})}{\mathcal{M}} \\ &= |2\gamma\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}|_{2d} + 2d\boldsymbol{e}. \end{aligned}$$

The term $\tilde{\boldsymbol{e}}$ comes from a fast conversion from $\mathcal{M}$. So, $\|\tilde{\boldsymbol{e}}\|_\infty \leqslant n - 1$. Moreover, it is straightforward to establish that $\|\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma}\|_\infty < 2d(\|\tilde{\boldsymbol{e}}\|_\infty + 1) + 2d\gamma\frac{\|\boldsymbol{c}\|_1+1}{\mathcal{M}}$. Consequently, due to the hypothesis $\mathcal{M} > \gamma(\|\boldsymbol{c}\|_1 + 1)$, it can be deduced that $\|\boldsymbol{e}\|_\infty \leqslant n$. And the proof is ended by applying Theorem 2.

Remark 3. Ideally, the correction technique is applied when the parameter ε associated with $\boldsymbol{R}$ is such that $\frac{1+2n}{2\varepsilon}$ is no larger than a modulus. Consequently, γ can be reduced to a single modulus RNS base as soon as $\sigma\rho_{\boldsymbol{R}} < \frac{1}{2} - \frac{1+2n}{2^{r+1}}$ (where r is the maximal binary size of any modulus), and the computation of a centred remainder modulo γ is straightforward.

Algorithm 7 $\texttt{GamRnsBabaiRO}(\boldsymbol{c}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}, \mathcal{R}, \mathcal{M}, \mathcal{M}', \tilde{m}, \gamma)$

Require: $d = \det \boldsymbol{R}$, $\boldsymbol{d} = (d, \ldots, d)$, $\widehat{\boldsymbol{R}} = d \times \boldsymbol{R}^{-1}$, $\tilde{\boldsymbol{R}} = 2 \times d \times \mathcal{M} \times \boldsymbol{R}^{-1} \bmod (2d)$, and $\tilde{d} = d \times \mathcal{M} \bmod (2d)$, $\tilde{\boldsymbol{d}} = (\tilde{d}, \ldots, \tilde{d})$; $\mathcal{M} > \gamma(\|\boldsymbol{c}\|_1 + 1)$, $\mathcal{M}' > (1+n)2d$, $\tilde{m} \geqslant n'$.

Ensure: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} = \left\lceil \frac{\widehat{\boldsymbol{R}}\boldsymbol{c}}{d} \right\rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor$ in $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$.

1: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} \leftarrow (2\gamma\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d})$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m} \cup \gamma$
2: $\tilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} \leftarrow (\gamma\tilde{\boldsymbol{R}}\boldsymbol{c} + \tilde{\boldsymbol{d}})$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m} \cup \gamma$
3: $\boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} \leftarrow \texttt{FastRnsModRed}(\tilde{\boldsymbol{f}}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma}, 2d, \mathcal{M}, \mathcal{M}' \cup \tilde{m} \cup \gamma)$
4: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} \leftarrow (\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma} - \boldsymbol{g}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma})(2d)^{-1}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}\cup\gamma}$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m} \cup \gamma$
5: $\boldsymbol{e}' \leftarrow \boldsymbol{f}_\gamma \operatorname{modc} \gamma$
6: $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} \leftarrow \left(\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}} - \boldsymbol{e}'\right)\gamma^{-1}$ ▷ in parallel in base $\mathcal{M} \cup \mathcal{M}' \cup \tilde{m}$
7: **return** $\boldsymbol{f}_{\mathcal{M}\cup\mathcal{M}'\cup\tilde{m}}$

13.5.4 Back to Lattice Cryptography

In cryptographic context, the rounding-off is useful to recover a vector $\boldsymbol{p}$ which is known to have components lower than a parameter σ. The decryption is then similar to solving a bounded distance decoding problem.

In Sect. 13.2.3, it is clear that RNS arithmetic is well adapted to the encryption part, where only additions and multiplications of matrices and vectors are involved. Since Babai's rounding-off can now be done efficiently, RNS is finally also convenient for implementing the decryption process. This is especially true because, in order to recover $\boldsymbol{p}$ such that $\|\boldsymbol{p}\|_\infty < \sigma$, it is sufficient to obtain the output of the rounding-off in a single modulus base m_σ which verifies $m_\sigma > 2\sigma$. Once $\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor \bmod m_\sigma$ is known, it follows that

$$p = (\boldsymbol{c} - \boldsymbol{R}\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor) \operatorname{modc} m_\sigma. \tag{13.12}$$

The computational effort of decryption is then decreased. Some costly (quadratic) base conversion between $\mathcal{M}$ and $\mathcal{M}'$ is replaced by (linear) conversions between $\mathcal{M}$ and m_σ.

13.6 Complexity Analysis and Comparison with Multi-Precision Computation

13.6.1 Complexity Analysis

13.6.1.1 RNS-Based Approaches

Complexity analysis of an RNS algorithm is usually made by considering the number of "elementary" multiplications (denoted EMM for Elementary Modular Multiplication), i.e., modular multiplications of residues. In practice, the size of moduli is determined by some hardware considerations, such as the size of a machine word. It is denoted by r from now on (i.e. any modulus m verifies $m < 2^r$). That way, any standard RNS multiplication in an n-moduli base is achieved with nEMM.

Algorithm 1 presents a generic version of an RNS modular reduction using Montgomery's approach. As previously stated, the cost of a multiplication, when performed in a base $\mathcal{M}$ or $\mathcal{M}'$, is equal to the cardinal of the base, in terms of EMM. Furthermore, if the number of RNS units is sufficient, such operation can be done in parallel on each element m_i of the base. In this case, the parallel cost is simply reduced to one EMM. More generally, a parallel step of computation over n units will refer to an "atomic" computation like $a \leftarrow |a + b \times c|_m$ or $a \leftarrow |(a + b) \times c|_m$ when executed in parallel on n residues. In practice, elementary modular reduction (by a modulus m) is not systematically done after every product. It can be performed only after a set of sums and products. Such approach is called lazy reduction. In order to simplify further discussions, this type of consideration is avoided. Finally, it has to be noticed that many of EM can be avoided by using adequate precomputations. Thus, further cost analysis will integrate this remark.

About the Mixed RNS/MRS Approach

We first focus on the procedures `NewRnsModRed` and `NewRnsBabaiRO` given by Algorithms 4 and 5. These variants were implemented on GPU/CPU [16].

First of all, the size of the bases $\mathcal{M}$ and $\mathcal{M}'$ have to be determined. The main purpose of $\mathcal{M}'$ is to be large enough to contain $s_{\mathcal{M}'}$ (line 4 of Algorithm 4), which is actually equal to either $(2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}) \bmod (2d)$ or $(2\widehat{\boldsymbol{R}}\boldsymbol{c} + \boldsymbol{d}) \bmod (2d) + 2d\boldsymbol{e}$ with $\|\boldsymbol{e}\|_\infty \leqslant 1$. Thus, having $\mathcal{M}' \geqslant 4d$ is necessary and sufficient. To determine the size of the main base $\mathcal{M}$, we remark that in the present context, for any given vector $\boldsymbol{c} = \boldsymbol{p} + \boldsymbol{R}\boldsymbol{k}$ in the space, only its distance to the lattice matters for the present context. Thus, by computing $\boldsymbol{c} \bmod d$, one obtains another vector $\boldsymbol{c}'$ which lies in $\boldsymbol{c} + \mathcal{L}$, and such that $\boldsymbol{c}' = \boldsymbol{p} + \boldsymbol{R}\boldsymbol{k}'$. It follows that $\boldsymbol{c}' - \boldsymbol{R}\lceil \boldsymbol{R}^{-1}\boldsymbol{c}' \rfloor = \boldsymbol{c} - \boldsymbol{R}\lceil \boldsymbol{R}^{-1}\boldsymbol{c} \rfloor$, i.e., both vectors $\boldsymbol{c}$ and $\boldsymbol{c}'$ are in $\boldsymbol{p} + \mathcal{L}$. Hence, due to the condition in Algorithm 5 about $\mathcal{M}$, and because $\|\boldsymbol{c}\tilde{\boldsymbol{R}} + \tilde{\boldsymbol{d}}\|_\infty < 2d(\|\boldsymbol{c}\|_1 + 1)$, it is sufficient to have $\mathcal{M} \geqslant \ell d$, where ℓ is the dimension of space. Finally, it can be assumed that $n = \lceil \log_{2^r}(\ell d + 1) \rceil$ and $n' = \lceil \log_{2^r}(4d) \rceil$.

Table 13.1 Cost of a mixed RNS/MRS scheme ($n = \lceil \log_{2^r}(\ell d + 1) \rceil$, $n' = \lceil \log_{2^r}(4d) \rceil$, ℓ: dimension of the lattice, d: determinant of the lattice)

Algorithm	EMM	Parallel steps (on $n+n'+1$ units)
NewRnsModRed	$(2nn' + \frac{n'(n'-1)}{2} + n + 4n' + 1)\ell$	$(n + n' + 5)\ell$
NewRnsBabaiRO	$2(n + n' + 1)\ell^2 + (2nn' + \frac{n'(n'-1)}{2} + 2n + 5n' + 2)\ell$	$2\ell^2 + (n + n' + 6)\ell$

Table 13.2 Cost of a full RNS scheme ($n = \lceil \log_{2^r}(\gamma(\ell d + 1)) \rceil$, $n' = \lceil \log_{2^r}(2d(n + 1)) \rceil$, ℓ: dimension of the lattice, d: determinant of the lattice)

Algorithm	EMM	Parallel steps (on $n+n'+2$ units)
FastRnsModRed	$(2nn' + 3n + 3n' + 3)\ell$	$(n + n' + 4)\ell$
GamRnsBabaiRO	$2(n + n' + 1)\ell^2 + (2nn' + 5n + 5n' + 6)\ell$	$2\ell^2 + (n + n' + 7)\ell$

The computation of MRS coefficient in base $\mathcal{M}'$ requires $\frac{n'(n'-1)}{2}$EMM. And the reconstruction of the value in $\mathcal{M}$ through a Horner scheme can be achieved with $n(n' - 1)$EMM. The full MRS conversion can be done in at least $n' - 1$ steps when $n + n'$ units can be run simultaneously. Consequently, the costs are summarized in Table 13.1.

About the Full RNS Approach

The pure RNS approach deals with the procedures FastRnsModRed and GamRnsBabaiRO depicted in Algorithms 6 and 7, which have been implemented on FPGA [5]. The complexity analysis is summarized in Table 13.2. In this case, the size of $\mathcal{M}$ is determined by the condition $\mathcal{M} > \gamma(\|\boldsymbol{c}\|_1 + 1)$, so $n = \lceil \log_{2^r}(\gamma(\ell d + 1)) \rceil$ is sufficient. And the size of $\mathcal{M}'$ has now to verify $\mathcal{M}' > 2d(n + 1)$, i.e., $n' = \lceil \log_{2^r}(2d(n + 1)) \rceil$.

Comparing the Two Approaches

When dealing with a basis $\boldsymbol{R}$ for which γ can fit on a single modulus (optimal case), the base $\mathcal{M}$ could require one more modulus for the full RNS approach ($n = \lceil \log_{2^r}(\ell d + 1) \rceil$ for RNS/MRS and $n = \lceil \log_{2^r}(\gamma(\ell d + 1)) \rceil$) for RNS). Despite that, the advantage of the full RNS computation in terms of number of elementary operations is clear. Getting rid of MRS-based conversions enables to cancel roughly $n'^{2/2\mathtt{EMM}}$. The minimal number of parallel steps remains similar though. However, this remark does not reflect a fundamental difference. Because the MRS conversion is (partially) sequential, the number of steps can dramatically increase as soon as the number of parallel units is reduced. Thus, even if, optimally, each method could be run as fast as the other one, the full RNS approach is a lot more flexible in terms of trade-off area/time.

Table 13.3 Cost of an RNS-based GGH-like decryption scheme $(\boldsymbol{c} - \boldsymbol{R}\lceil \boldsymbol{R}^{-1}\boldsymbol{c}\rfloor) \bmod m_\sigma$ (with $n = \lceil \log_{2^r}(\ell d + 1)\rceil$, $n' = \lceil \log_{2^r}(4d)\rceil$, $\gamma < 2^r$)

Algorithm	# units	EMM	Parallel steps
Mixed RNS/MRS	$n + n' + 1$	$(2n + 2n' + 3)\ell^2 + (nn' + \frac{n'(n'-1)}{2} + 3n + 3n' + 3)\ell$	$3\ell^2 + (n + n' + 5)\ell$
Full RNS	$n + 2$	$(2n + 3)\ell^2 + (2n + 5)\ell$	$3\ell^2 + (n + 5)\ell$

Time Complexity of a Full GGH-Like Decryption Computation

As a practical example, when the full RNS version for lattice cryptographic protocols as GGH, it is sufficient to compute $(\boldsymbol{c} - \boldsymbol{R}\lceil \boldsymbol{R}^{-1}\boldsymbol{c}\rfloor) \bmod m_\sigma$.

In this context, the advantage of a full RNS approach becomes more significant, because the base $\mathcal{M}'$ can be reduced to one element m_σ. On the contrary, the mixed RNS/MRS scheme still requires a larger base $\mathcal{M}'$, together with the modulus $\tilde{m}$ for the small Montgomery reduction inside $\mathcal{M}'$, in order to enable a comparison. However, only the computation of MRS coefficients is needed. And m_σ can be part of $\mathcal{M}'$.

Finally, these simplifications make the complexity of a full RNS scheme becoming linear in n, whereas the mixed RNS/MRS scheme keeps quadratic terms (n'^2, nn'), as depicted in Table 13.3.

13.6.1.2 Asymptotic Comparison with a Multi-Precision Approach

For a fair comparison, the computation of (13.12) by using Algorithm 7 (and whose cost is summarized in Table 13.3) is compared to the same computation performed in a classical positional system with radix 2^r. For instance, it means that any integer $x < \mathcal{M}$ is represented either by n residues in RNS or by n words in multi-precision representation. Furthermore, a multiplication of two n-words integers is assumed to cost $\mathcal{O}(n^{1+\varepsilon})$ elementary multiplications EM ($\varepsilon = 1$ for schoolbook algorithm, $\varepsilon \sim 0.585$ for Karatsuba's algorithm, etc.). Lastly, the basis $\boldsymbol{R}$ and the integer γ involved in Algorithm 7 (and verifying the condition of Theorem 3) are assumed to be such that $\gamma < 2^r$, i.e., γ is a single modulus.

For the purpose of efficiency, the matrix $\boldsymbol{R}^{-1}$ is assumed to be precomputed with a precision of t words. Thus, one computes $\boldsymbol{R}_E = \boldsymbol{R}^{-1} + \boldsymbol{E}$, where $\|\boldsymbol{E}\|_\infty < 2^{-rt}$. The parameter t has to be large enough so that, for any $\boldsymbol{c} = \boldsymbol{p} + \boldsymbol{kUR}$ (mod$d$, and with $\boldsymbol{U} \in GL_\ell(\mathbb{Z})$), the following equalities stands:

$$\lceil \boldsymbol{R}_E\boldsymbol{c}\rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{c}\rfloor \Leftrightarrow \lceil \boldsymbol{R}^{-1}\boldsymbol{p} + \boldsymbol{Ec}\rfloor = \lceil \boldsymbol{R}^{-1}\boldsymbol{p}\rfloor.$$

Since it is assumed that $\|\boldsymbol{R}^{-1}\boldsymbol{p}\|_\infty < \frac{1}{2} - \varepsilon$ with $\varepsilon = \frac{1+2n}{2\gamma}$, one has in particular $\|\boldsymbol{R}^{-1}\boldsymbol{p}\|_\infty < \frac{1}{2} - n2^{-r}$. Therefore, it is sufficient that $\|\boldsymbol{Ec}\|_\infty \leqslant n2^{-r}$, or again that $\|\boldsymbol{E}\|_\infty \leqslant n(2^r\ell d)^{-1}$ by using $\|\boldsymbol{Ec}\|_\infty \leqslant \|\boldsymbol{E}\|_\infty\|\boldsymbol{c}\|_1 < \ell d\|\boldsymbol{E}\|_\infty$.

Table 13.4 Asymptotic complexities of full RNS and multi-precision GGH-like decryption

Method	Elementary operations	Space complexity
Full RNS	$\mathcal{O}(\ell^2 n)$	$\mathcal{O}(\ell^2 n)$
RNS/MRS	$\mathcal{O}(\ell n(\ell + n))$	$\mathcal{O}(\ell^2 n)$
Multi-precision	$\mathcal{O}(\ell^2 n^{1+\varepsilon})$	$\mathcal{O}(\ell^2 n)$

To sum up, each coefficient of $\boldsymbol{R}_E$ is computed with $\lceil \log_{2^r}(2^r \ell d/n) \rceil$ words of precision. If n is the parameter associated with the full RNS method, then $\lceil \log_{2^r}(2^r \ell d/n) \rceil \sim n$ as soon as $\gamma < 2^r$ (we remind that $n = \lceil \log_{2^r}(\gamma(\ell d + 1)) \rceil$). Moreover, in this case, the RNS/MRS approach requires $n - 1$ moduli in the base $\mathcal{M}$ (i.e. one less than the full RNS scheme). Consequently, the space complexity of the three approaches is in $\mathcal{O}(\ell^2 n)$.

The asymptotic number of elementary operations in RNS scheme has been shown to be in $\mathcal{O}(\ell^2 n)$. The one of RNS/MRS approach is in $\mathcal{O}(\ell^2 n + \ell(n'^2 + nn'))$. By noticing that $n' \in \mathcal{O}(n)$, the computational complexity of mixed approach is in $\mathcal{O}(\ell n(\ell + n))$. About the multi-precision computation, the cost is dominated by the product $\boldsymbol{R}_E \boldsymbol{c}$, which can be achieved with at most $\mathcal{O}(\ell^2 n^{1+\varepsilon})\mathtt{EM}$. Hence, a full RNS approach enables as well a practical enhancement by allowing an easy parallelization as a better asymptotic time complexity (for an equivalent space complexity) (Table 13.4). This is especially important for large cryptographic parameters.

13.6.2 Example of Implementation on CPU/GPU of RNS/MRS Approach

As a matter of example, some practical results provided in [16] are indicated hereafter. The RNS/MRS approach was implemented on multi-core CPU and GPU and was compared with a classical multi-precision computation. Even though an optimized multi-precision multiplication algorithm can make the classical approach asymptotically better, the results of implementation show that the parallelism brought by RNS allows noticeable speed-ups, despite burdensome RNS to MRS conversions.

The tests were run on GGH-like decryptions. The RNS/MRS implementation was based on Algorithm 5, with a modulus size of 32 bits. The multi-precision algorithm was implemented by using the NTL library [31], and the algorithm was only based on integer and floating point multi-precision arithmetic (no modular arithmetic at all). In the case of a CPU+GPU platform, the GPU was used to run the rounding-off, while the remaining of the computation was ended on the multi-core CPU. The speed-ups are reported in Fig. 13.3.

Surprisingly, the results are better when the computation is entirely run on a multi-core CPU than on GPU. The bandwidth is a critical issue. The arithmetic intensity is too low to make the use of GPU advantageous. Furthermore, the use of

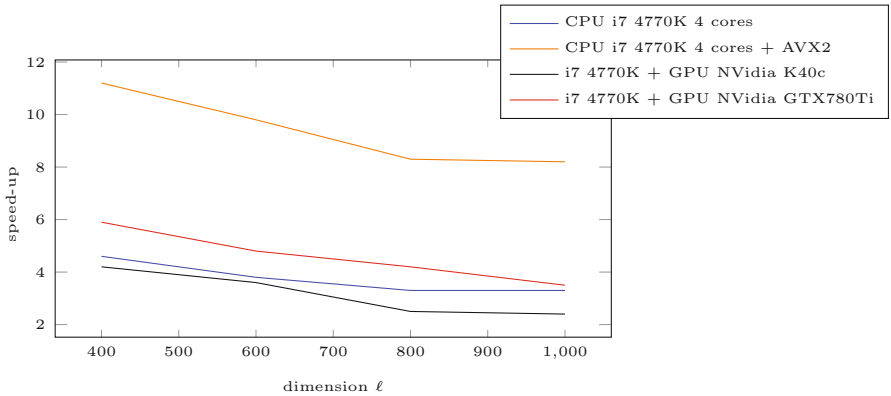

Fig. 13.3 Speed-ups of RNS/MRS-based GGH-like decryption comparatively to a classical multi-precision implementation, on different platforms

Single Instruction Multiple Data (SIMD) (through AVX2 instruction sets) enables to exploit the natural concurrency features of RNS, which greatly enhances the performance.

13.7 Conclusion

In this chapter, the advantages of RNS arithmetic have been investigated in the domain of lattice-based cryptography. Many primitives in this domain rely on matrix–vector products, for which RNS is particularly well adapted.

Babai's rounding-off is a core operation, which can be useful as well for designing trapdoor functions as for cryptanalytic purposes. Since it involves rational arithmetic and rounding-off function, it does not match with RNS arithmetic as is. It has been shown how to overcome these issues in order to turn it into a more suitable algorithm. Different solutions have been analysed. Beyond a first natural approach mixing RNS and MRS arithmetics, a specific solution to these problems has enabled to develop a full RNS algorithm for implementing the rounding-off exactly. Beyond a better asymptotic complexity, this allows to keep benefits of parallelization features of RNS.

References

1. M. Ajtai, Generating hard instances of lattice problems (Extended abstract), in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing (STOC)* (ACM, New York, 1996), pp. 99–108

2. L. Babai, On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**(1), 1–13 (1986)
3. J.-C. Bajard, L.-S. Didier, P. Kornerup, Modular multiplication and base extensions in residue number systems, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (2001), pp. 59–65
4. J.-C. Bajard, J. Eynard, N. Merkiche, T. Plantard, Babai round-off CVP method in RNS: application to lattice based cryptographic protocols, in *2014 14th International Symposium on Integrated Circuits (ISIC)* (2014), pp. 440–443
5. J.-C. Bajard, J. Eynard, N. Merkiche, T. Plantard, RNS arithmetic approach in lattice-based cryptography: accelerating the "Rounding-off" core procedure, in *2015 IEEE 22nd Symposium on Computer Arithmetic (ARITH)* (2015), pp. 113–120
6. J.W. Bos, K. Lauter, J. Loftus, M. Naehrig, Improved security for a ring-based fully homomorphic encryption scheme, in *Cryptography and Coding*, vol. 8308, ed. by M. Stam. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 2013), pp. 45–64
7. Z. Brakerski, V. Vaikuntanathan, Efficient fully homomorphic encryption from (Standard) LWE, in *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE Computer Society, Washington, 2011), pp. 97–106
8. W. Diffie, M. Hellman, New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (2006)
9. H.L. Garner, The residue number system, in *Papers Presented at the March 3–5, 1959, Western Joint Computer Conference*. IRE-AIEE-ACM (Western) (ACM, New York, 1959), pp. 146–153
10. C. Gentry, Fully homomorphic encryption using ideal lattices, in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC)* (ACM, New York, 2009), pp. 169–178
11. O. Goldreich, S. Goldwasser, S. Halevi, Public-key cryptosystems from lattice reduction problems, in *Advances in Cryptology, CRYPTO*, vol. 1294, ed. by B.S. Kaliski Jr. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 1997), pp. 112–131
12. J. Hoffstein, J. Pipher, J.H. Silverman, NTRU: a ring-based public key cryptosystem, in *Lecture Notes in Computer Science* (Springer, Berlin, 1998), pp. 267–288
13. J. Hoffstein, N. Howgrave-Graham, J. Pipher, W. Whyte, *Practical Lattice-Based Cryptography: NTRUEncrypt and NTRUSign* (Springer, Berlin/Heidelberg, 2010), pp. 349–390
14. A.K. Lenstra, H.W. Lenstra Jr., L. Lovász, Factoring polynomials with rational coefficients. Math. Ann. **261**(4), 515–534 (1982)
15. C. Ludwig, The security and efficiency of Micciancio's cryptosystem. Cryptology ePrint Archive, Report 2004/209 (2004). http://eprint.iacr.org/2004/209
16. P. Martins, L. Sousa, J. Eynard, J.-C. Bajard, Programmable RNS lattice-based parallel cryptographic decryption, in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2015), pp. 149–153
17. C.A. Melchor, X. Boyen, J.-C. Deneuville, P. Gaborit, Sealing the leak on classical NTRU signatures, in *Post-Quantum Cryptography*, vol. 8772, ed. by M. Mosca. Lecture Notes in Computer Science (Springer International Publishing, New York, 2014), pp. 1–21
18. D. Micciancio, Improving lattice based cryptosystems using the hermite normal form, in *Cryptography and Lattices*, vol. 2146, ed. by J.H. Silverman. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 2001), pp. 126–145
19. P.Q. Nguyen, Cryptanalysis of the Goldreich-Goldwasser-Halevi cryptosystem from crypto 97, in *In Proceedings of Crypto*. Lecture Notes in Computer Science, vol. 1666 (Springer, Berlin, 1999), pp. 288–304
20. P.Q. Nguyen, O. Regev, Learning a parallelepiped: cryptanalysis of GGH and NTRU signatures, in *Advances in Cryptology - EUROCRYPT 2006*, vol. 4004, ed. by S. Vaudenay. Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, 2006), pp. 271–288
21. C. Peikert, Public-key Cryptosystems from the worst-case shortest vector problem: extended abstract, in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC)* (ACM, New York, 2009), pp. 333–342

22. T. Plantard, M. Rose, W. Susilo, Improvement of lattice-based cryptography using CRT, in *Quantum Communication and Quantum Networking*, vol. 36, ed. by A. Sergienko, S. Pascazio, P. Villoresi. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (Springer, Berlin/Heidelberg, 2010), pp. 275–282
23. K.C. Posch, R. Posch, Modulo reduction in residue number systems. IEEE Trans. Parallel Distrib. Syst. **6**(5), 449–454 (1995)
24. O. Regev, New lattice based cryptographic constructions, in *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing (STOC)* (ACM, New York, 2003), pp. 407–416
25. O. Regev, On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009)
26. R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
27. M. Rose, T. Plantard, W. Susilo, Improving BDD cryptosystems in general lattices, in *Information Security Practice and Experience*, vol. 6672, ed. by F. Bao, J. Weng. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 2011), pp. 152–167
28. C.P. Schnorr, A hierarchy of polynomial time lattice basis reduction algorithms. Theor. Comput. Sci. **53**(2–3), 201–224 (1987)
29. P.P. Shenoy, R. Kumaresan, Fast base extension using a redundant modulus in RNS. IEEE Trans. Comput. **38**(2), 292–297 (1989)
30. P.W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in *1994 Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134
31. V. Shoup, NTL 5.4.1: A Library for doing Number Theory. www.shoup.net/ntl (2007)
32. M. Yoshino, N. Kunihiro, Improving GGH cryptosystem for large error vector, in *2012 International Symposium on Information Theory and its Applications (ISITA)* (2012), pp. 416–420

Chapter 14
RNS Applications in Computer Networks

Azadeh Alsadat Emrani Zarandi

14.1 Introduction

Residue number system (RNS) is a non-weighted number system, which represents any integer number with some residues, based on the selected moduli set. Therefore, computations can be done on these residues in parallel to decrease the delay, and also due to the reduction of bit-length, low power processors can be designed. Moreover, another characteristic of this number system is its ability to improve the reliability by using redundant moduli. These are the most famous advantages of RNS, which numerous works have been done on them, and they led to valuable achievements [1]. However, recently modern applications of RNS have been introduced which profit from RNS in new aspects with distinct goals and also new challenges [2]. RNS can be used in network applications with the aim of solving some issues such as determining output port in simpler way [3] or increasing a wireless sensor network (WSN) lifetime [4]. Moreover, it is possible to design a switch which can process multicast or unicast traffic by using RNS concepts [3, 5]. Redundant residue number system is even used in cloud storage with the aim of providing a novel method to enhance reliability [6]. Another application of RNS in WSN is its usage in increasing the reliability of data transmission by improving packet reception rate and end-to-end delay [7]. Furthermore, RNS can be applied to WSN in order to perform data aggregation [8] which results in a good trade-off between energy consumption and reliability.

This chapter briefly reviews some of the important and effective applications of RNS in computer networks to show how RNS can improve performance in different ways than conventional methods. In other words, our aim is creating a

A.A.E. Zarandi (✉)
Department of Computer Engineering, Shahid Bahonar University of Kerman, Kerman 7616914111, Iran
e-mail: a.emrani@uk.ac.ir

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6_14

better understanding about RNS and its brilliant features in networking applications. This chapter is organized as follows. Section 14.2 explains how RNS can be useful in forwarding of unicast and multicast packets in Software-Defined Network (SDN). Section 14.3 focuses on Mobile Ad Hoc Network (MANETs) and solves the same problem in it. In Sect. 14.4, the application of RNS in replacing table lookup with a novel method is investigated. Finally, prolonging wireless sensor network lifetime with RNS is stated in Sect. 14.5, and Sect. 14.6 concludes the chapter.

14.2 RNS for Unicast/Multicast Routing and Forwarding in SDN

Software-defined networking, by separating the data and control plane, introduces significant potentials in network management, and leads to enhanced resource utilization [9]. SDN offers several benefits such as automated load balancing, on-demand provisioning, and a better scalability. However, there are still some unsolved concerns which should be considered in this emerging network paradigm according to its features. One of these concerns which can greatly benefits modern networks is multicasting. Multicasting, by transmitting a packet to multiple recipients, has a significant effect on networks bandwidth and its throughput. Not only a replicated packet should be delivered to all destinations of the multicast group, but also hosts out of the multicast group should not receive this data packet. Distinct multicast routing protocols such as DVMRP [10] and PIM [11] were introduced, which need to keep routing states in Multicast Routing Table (MRT). Such traditional multicast algorithms suffer from huge amount of required memory space and high process time to perform the routing. With the aim of improving flexibility, scalability, and the efficiency of multicast forwarding, and based on Chinese remainder theorem (CRT) properties, residues are used in Scalar-pair Vectors Routing and Forwarding (SVRF) algorithm [3]. Simulation results illustrate that the mentioned method can lead to less memory usage, processing time, and hardware cost compared with the general Bloom filter [12], which is widely used in network functions especially unicast/multicast forwarding. Besides, scalability and accuracy are other findings of this method.

14.2.1 SVRF Approach

This section describes a recently introduced multicast forwarding/routing method (SVRF) [3] based on utilizing RNS properties to improve multicast packet forwarding in SDNs. This algorithm treats multicast and unicast packets in the same way, and enhances the forwarding process. The first step of SVRF is gathering required routing information for the unicast path or multicast tree by SDN controller.

A unique key should be assigned to each group. These keys are pairwise relative prime, and the function in (14.1) generates these keys:

$$K = \{P(a_i) \,|\forall i \leq n \quad \text{and } a_i \in A\} \tag{14.1}$$

In this equation $P(a_i)$ is a perfect function to generate prime numbers from a defined range either sequentially or randomly [3] for each routing identifier, a_i. Moreover, $k_i \in K \subset P$, and P is the set of prime numbers. It should be mentioned that k_i must be larger than 2^ρ where ρ is the number of ports in the switch. The next scalar, $\boldsymbol{M}_{\text{cp}}$, is the product of all n elements in K as indicated in (14.2):

$$M_{\text{cp}} = \prod_{i=1}^{n} k_i \text{ where } k_i \in K \tag{14.2}$$

The other required set is $M = \{m_1, m_2, \ldots, m_{\text{n}}\}$ so that $m_i \pmod{b_i} = 0$ for all m_i where b_i is an output port bitmap(s) and its elements are calculated based on M_{cp} and set K as:

$$M = \left\{ \frac{M_{\text{CP}}}{k_1} \middle| \forall i \leq n, \text{ and } k_i \in K \right\} \tag{14.3}$$

Moreover, another required set is defined as $C = \{c_1, c_2, \ldots, c_n\}$, where

$$C = \left\{ m_i \times \left(m_i^{-1} \pmod{k_i} \right) \middle| \forall i \leq n, \text{ and } m_i \in M \right\} \tag{14.4}$$

In this equation, m_i^{-1} is the multiplicative inverse [13] which is calculated by $m_i^{-1}\, m_i \pmod{k_i} = 1$. Now, M_{crt} can be computed by Eq. (14.5):

$$M_{\text{crt}} = \left(\sum_{i=1}^{n} (b_i \times c_i) \right) \left(\text{mod } M_{\text{cp}} \right) \tag{14.5}$$

By having all required values, residues can help us in two directions, first a zero or nonzero residue can help to recognize that the packet is belonging to which multicast group. Moreover, the remainder of the other division is Output Port Bitmap (OPB) or Output Port Index (OPI), which can help to send packets to suitable output ports. In other words, scalars (M_{cp}, M_{crt}) are the dividends and node-specific key k_i *is* the divisor:

$$M_{\text{cp}} \equiv \begin{cases} 0 \pmod{k_1} \\ 0 \pmod{k_2} \\ \quad \cdots \\ 0 \pmod{k_n} \end{cases} \tag{14.6}$$

$$M_{\text{crt}} \equiv \begin{cases} b_1 \pmod{k_1} \\ b_2 \pmod{k_2} \\ \quad \cdots \\ b_n \pmod{k_n} \end{cases} \tag{14.7}$$

To sum up, the scalar-pair is computed and added to switch flow table. The following equation can obtain the residue b_i at i-th switch:

$$b_i = \begin{cases} M_{\text{crt}} \,(\bmod k_i)\,, & \text{if } M_{\text{cp}} \pmod{k_i} = 0 \\ 0 & , \text{ if } M_{\text{cp}} \pmod{k_i} \neq 0 \end{cases} \tag{14.8}$$

The relation (14.8) states that at the first, the residue of the division of M_{cp} by k_i should be checked. If this residue is zero, the other reminder achieved from dividing M_{crt} by k_i is the OPB. On the other hand, the packet should be dropped if M_{cp} (mod k_i) is nonzero and b_i is zero. As it can be seen, by using this method a unitary scalar-pair with sufficient keys is enough to forward multicast and unicast packets without any conflict. In order to have a better understanding of SVRF functionality in an SDN switch, its block diagram is shown in Fig. 14.1.

The SDN controller calculates the scalar-pair (M_{cp}, M_{crt}) and it is saved in the switch's memory in advance. When a packet arrives, firstly, the destination address a_i is extracted and the required data are achieved from it by address analyzer, such as determining being unicast or multicast. Then, the related key is obtained from key generation function (14.1). Finally, the output port index is calculated by (M_{CRT} mod k_i) if (M_{CP} mod k_i) is zero. If the packet is a unicast packet, this residue is the output port index, but in the case of multicast, normalizing should be done to have output port bitmap [3].

In conclusion, by using M_{cp} and M_{crt}, this method results in decreasing the amount of memory, processing time, and cost when it is compared with Bloom filter [12], since forwarding ports can be easily computed via simple modular operations. With introducing algorithms such as SVRF, it is expected to achieve better scalability for future SDN networks specially with increasing speed.

14.3 RNS-Based Unicast/Multicast Routing in MANETs

Mobile ad hoc Networks consist of mobile devices with wireless capability to communicate. Their dynamic and autonomous topology has attracted lots of researches due to their significant role in the next generation networking, and also its important unsolved challenges. One of these challenges is designing an efficient routing protocol, since the available unicast and multicast routing protocols suffer from high overhead and forwarding delay. Moreover, the current protocols require different routing methods for unicasting and multicasting [5]. Recently, to enhance routing in MANETs, CRT has been used to improve multicast functionality.

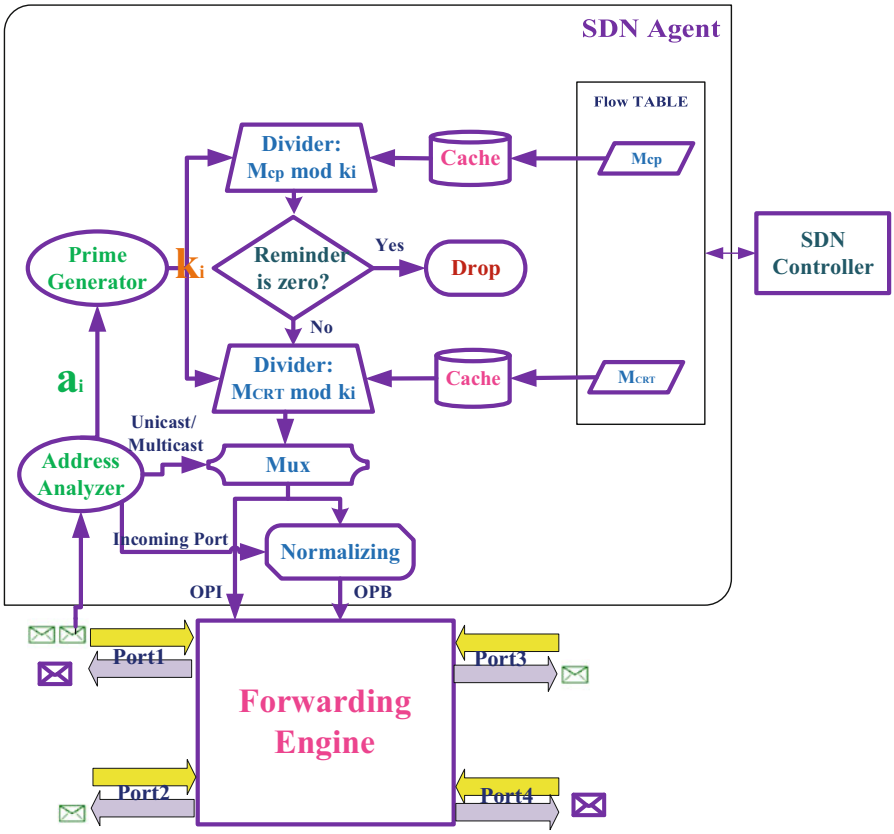

Fig. 14.1 The block diagram of SVRF functionality in an SDN [3]

Arithmetic Labeled EXplicit (ALEX) multicast [5] is suggested to achieve a stateless multicasting using moduli as node-specific keys. Its details are explained in the following subsection.

14.3.1 ALEX Approach

In order to implement ALEX in a MANET with n forwarding nodes in a multicast tree (S,G), two arrays with n elements are required including specific keys for each node which are co-prime with each other, $K(S,G) = \{k_1, k_2, \ldots, k_n\}$, and also a forwarding bitmap, named as $B(S,G) = \{b_1, b_2, \ldots b_n\}$, where b_i are integers such that $b_i < k$. In this method, source nodes need to find the shortest unicast path and also multicast tree, by cooperating with different receivers. For these paths, the KEYs and *Forwarding Group Bitmap (FGB)* are used to calculate path identifier

in order to attach it to packets. If there is one path between source and destination nodes, the key set, $K(S,G) = \{k_1, k_2, \ldots k_i\}$ which consists of positive integers as keys, assigned to the nodes in the path, and for n receivers paths, the union of all related key sets should be considered. The next step is calculating *Multicast Channel-specific IDentifier (*MCID*)* using $K(S,G)$ and $B(S,G)$ arrays to attach it to packets. Each node of multicast tree uses MCID to find its forwarding neighbor(s). Since, MCID is calculated based on K and B, each intermediate node can achieve FGB_{bi} using its specific key k_i. The related operations are described in the following with more details.

As a first step, the product of all numbers in the array $K(S, G)$ is computed as follows:

$$\kappa = \prod_{i=1}^{v} k_i \text{ for } k_i \in K(S, G) \text{ and } v = |K(S, G)| \tag{14.9}$$

After calculating the κ, another array is needed which is named as M and it contains $m_1, m_2, \ldots, m_v$, and each of them can be achieved using the following relation:

$$m_i = \frac{k}{k_i} \text{ for } i \leq v \tag{14.10}$$

By knowing the multiplicative inverse (m_i^{-1}) [13], another array is formed as $C = \{c_1, c_2, \ldots, c_v\}$:

$$c_i = m_i \left(m_i^{-1} \bmod k_i\right) \text{ for } i \leq v \tag{14.11}$$

The multiplicative inverse can be easily computed, e.g., by using a few lines of recursive C code [14]. Finally, the MCID can be calculated as follows:

$$\mathrm{MCID} = \left(\sum_{i=1}^{v} k_i c_i\right) \bmod \kappa \tag{14.12}$$

Therefore, CRT is used to create the scalar MCID from K and B. Thereafter, elements of B can be recalculated in each intermediate node by having MCID and keys. In order to have a better understanding of this method, an example is shown in Fig. 14.2. As it can be seen in this figure, a packet is going to be sent to three nodes including A, F, and J. Each node has a special key, as an example the moduli associated with A is K_A. The MCID is calculated based on the path and node's keys, and it is attached to each packet. In each intermediate node, the residue of dividing MCID to its moduli, which is node-specific key, represents the next step. Whenever the residue becomes zero, the packet is not forwarded anymore.

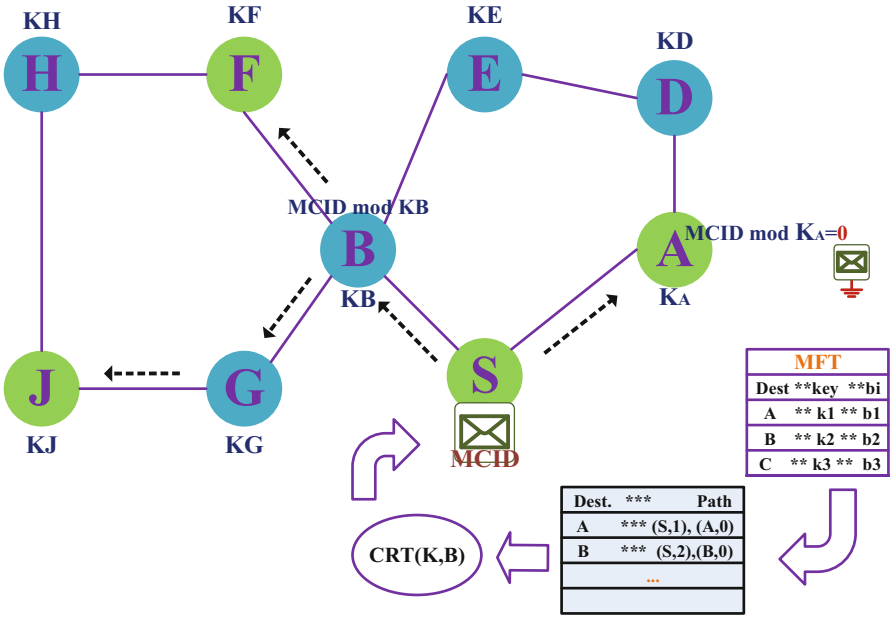

Fig. 14.2 The ALEX forwarding in MANET [5]

14.3.2 ALEX Achievements

As it was mentioned, ALEX uses some moduli as node special keys and by creating MCID can lead to self-routing of multicast packets. In other words, simple modulo operation in intermediate nodes determines the next steps. The main advantage of the ALEX is increasing the scalability which means there can be more multicast groups in MANETs. Moreover, by using MCID and encoding the list of nodes in the packet by CRT instead of having a multicast address, the routing states are not needed anymore in intermediate nodes. Furthermore, ALEX can achieve faster forwarding by decreasing the protocol overhead and processing time, and also a better security can be achieved by using this method of coding. Simulation results show that using ALEX results in a good packet delivery rate in the large-scale MANETs with lot of crowded groups and also high traffic. Another significant feature of the ALEX is its ability to forward both multicast and unicast traffic. Finally, for future MANETs with the need of Quality of Service (QOS), this method can achieve an efficient bandwidth utilization together with QOS guarantees.

14.4 RNS in SDN Core Fabrics

A significant challenge in communication networks is designing novel methods focusing on efficient and flexible header processing to achieve sufficient speed at high data rate. Rapid growth of networks makes lookup a bottleneck in switches particularly by increasing the number of nodes. An approach recently introduced based on using residue number system, which can replace the table lookup by some RNS operations. The KeyFlow [15] method can reduce round trip time (RTT) more than 50 % as it is indicated by simulation results.

In order to use this method, an RNS moduli should be assigned to each switch as its ID. These numbers should be pairwise-relatively prime and larger than the number of switches' physical interfaces. Furthermore, if the ID of the switch is N, a number from the range $[0, N-1]$ is assigned to each interface of the switch. Whenever an appropriate route is found by a routing algorithm, the KeyFlow in the controller should know the set of switch IDs along this path, and also the outgoing port number. Now, RNS can be used to generate unique path ID to be assigned to each packet that wants to traverse this path. The path ID can be achieved using any reconstruction algorithm such as CRT by considering switches' ID as moduli and port numbers as residues.

To forward a packet in the KeyFlow-based network, the path ID is attached in the packet header. In each hop, the reminder of dividing path ID by switch ID is calculated, and this indicates the output port number. The simple view of this algorithm is shown in Fig. 14.3. It should be mentioned that the number assigned to each switch is its ID.

To sum up, a KeyFlow approach can be used to design flexible network-fabric-based model, with the aim of replacing table lookup with some modular operations. Simulation results indicate that using RNS results in improving RTT about 50 %, especially for populated networks with high node density. Moreover, keeping active flow state in the network is reduced over 30 %.

14.5 RNS for Improving Energy Saving and Reliability in WSN

Wireless sensor networks consist of several nodes with limited processing ability. Based on the network application, these nodes should send the necessary information to the sink periodically via multi-hop transmission. Besides, these nodes usually operate based on battery and due to their limited lifetime, energy is a critical parameter in their performance. Not only the period of transmission is effective on their energy consumption, but also the number of transmitted bits is so significant. Furthermore, in WSNs, it is preferred to have a load balancing, rather than missing some nodes sooner than others, since it may damage the network connectivity or it may divide the network into some disjoint sub-networks. To overcome these

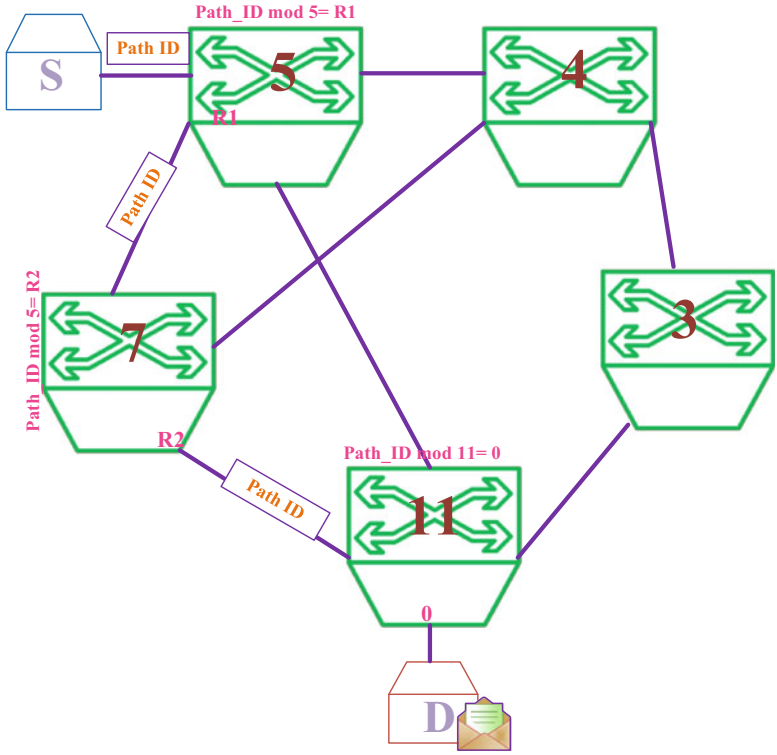

Fig. 14.3 KeyFlow: the RNS-based switching approach [15]

challenges, it is suggested to use small residues with the aim of reducing the number of bits to achieve energy efficiency and reliability [4]. Therefore, as another modern application of RNS, its usability in WSNs to prolong its lifetime is discussed in this section.

As it was mentioned, the bit-length of the forwarded message effects the required transmission energy, and a method to reduce the number of bits is using residues in RNS, which can help with two aims [4]. First, using residues reduce the bit-length and therefore the consumed energy of transmissions. Second, the average load can be distributed between distinct nodes instead of consuming the energy of few nodes, hence, the network lifetime can be increased.

Considering a WSN where its nodes are arranged in clusters during the initialization phase. After this phase, nodes know its next and predecessor nodes while transmitting information. When a packet is forwarded to the sink, it can be split once whenever there are at least N next-hop nodes. It means that, when there are at least N next hops, the complete packet is sent to all of them. After receiving the packet, each node computes the residue of dividing it by the selected modulo, and reminders will be sent to continue their path. On the other hand, after receiving all reminders in the sink, the original message can be reconstructed using CRT.

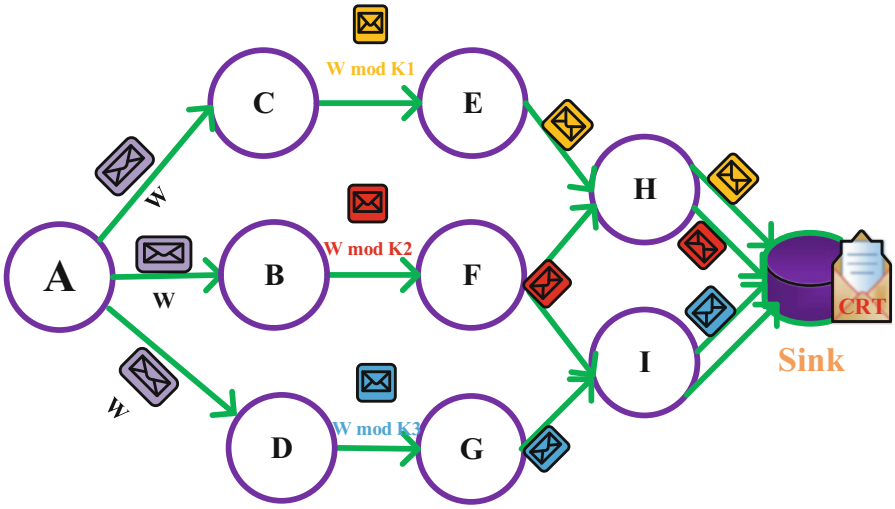

Fig. 14.4 Packet splitting in WSN using RNS [4]

The other advantage of this method is its ability to increase the reliability. In other words, the redundant residue number system can be used instead of regular RNS which means that the system can tolerate f failures. Although the overhead is increased in this situation, still the original data can be retrieved by missing some residues. Since the prime numbers used as moduli are kept in lookup tables and for each N and f, the same prime set is used in the whole network, moduli are not needed to be sent to the sink with packets. It is only enough to send the number of chunks (N) and also the component index (i), which are placed in the header.

Figure 14.4 shows a simple example of forwarding a packet in WSN, using the mentioned method. This w-bit packet is converted to residues while forwarding from node A to the sink at the second hop. Given N and f, a lookup-table and a list of moduli stored in each sensor's memory can help to know the minimum prime set with f possible faults. As it is obvious, increasing the number of possible faults results in more energy consumption. An interesting point about this method is that the sink is performing reverse conversion to retrieve the original data. That means sensor nodes with their limited energy source only perform simple modulo operation to compute residues. On the other hand, reverse conversion [16] which needs more energy is done in the sink which has no concern about energy. Therefore, the major finding of using RNS in this way is consuming less energy and prolonging the network lifetime.

14.6 Conclusion

Using RNS as a number system which can perform parallel computations is the most well-known feature of RNS. Although lots of works have been done based on this feature and brilliant results have been achieved, RNS has the potential to improve other applications. This chapter discusses some modern applications of residue number system, in the field of computer networks. It was shown that not only RNS can be used to design low power computational systems, but it can also be helpful to solve distinct challenges in different types of networks. It was indicated how RNS can be useful in energy saving and load balancing in WSN. Moreover, RNS can play a significant role in future networks by its valuable ability to simplify packet forwarding. Additionally, there are other fields that RNS can be helpful such as data aggregation [8]. Therefore, by creating a better understanding about RNS and its features, distinct improvements can be achieved.

References

1. C.H. Chang, A.S. Molahosseini, A.A.E. Zarandi, T.F. Tay, Residue number systems: a new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. IEEE Circuits Syst. Mag. **15**(4), 26–44 (2015)
2. A.S. Molahosseini, S. Sorouri, A.A.E. Zarandi, Research challenges in next-generation residue number system architectures, in *Proceedings of International Conference on Computer Science & Education (ICCSE)*, 2012
3. W.K. Jia, L.C. Wang, A unified unicast and multicast routing and forwarding algorithm for software-defined datacenter networks. IEEE J. Selected Areas Commun. **31**(12), 2646–2657 (2013)
4. G. Campobello, A. Leonardi, S. Palazzo, Improving energy saving and reliability in wireless sensor networks using a simple crt-based packet-forwarding solution. IEEE ACM Trans. Netw. **20**(1), 191–205 (2012)
5. W.K. Jia, C.Y. Chen, Y.C. Chen, ALEX: an arithmetic-based unified unicast and multicast routing for MANETs, in *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, 2014
6. A. Celesti, M. Fazio, M. Villari, A. Puliafito, Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. J. Netw. Comput. Appl. **59**, 208–218 (2016)
7. R. Ye, A. Boukerch, H. Wang, X. Zho, RESIDENT: a reliable residue number system-based data transmission mechanism for wireless sensor networks. Springer J. Wireless Netw. 1–14, to appear, 2016
8. G. Campobello, S. Serrano, L. Galluccio, S. Palazzo, Applying the Chinese remainder theorem to data aggregation in wireless sensor networks. IEEE Commun. Lett. **17**(5), 1000–1003 (2013)
9. H. Kim, N. Feamster, Improving network management with software defined networking. IEEE Commun. Mag. **51**(2), 114–119 (2013)
10. D. Waitzman, C. Partridge, S. Deering, Distance vector multicast routing protocol, RFC 1075, Internet Engineering Task Force, 1988
11. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, L. Wei, An architecture for wide-area multicast routing. SIGCOMM Comput. Commun. Rev. **24**(4), 126–135 (1994)

12. B.H. Bloom, Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
13. K. Navi, A.S. Molahosseini, M. Esmaeildoust, How to teach residue number system to computer scientists and engineers. IEEE Trans. Educ. **54**(1), 156–163 (2011)
14. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to algorithms* (MIT Press, Cambridge, MA, 2001)
15. M. Martinello, M.R.N. Ribeiro, R. Emerick, Z.D. de Oliveira, R.A. Vitoi, KeyFlow: a prototype for evolving SDN toward core network fabrics. IEEE Netw. **28**(2), 12–19 (2014)
16. A.A.E. Zarandi, A.S. Molahosseini, L. Sousa, M. Hosseinzadeh, An efficient component for designing signed reverse converters for a class of RNS moduli sets of composite form $\{2^k, 2^p - 1\}$. IEEE Trans. Very Large Scale Integr. Syst., to appear, 2016

Index

A
Adaptive filters, 202–204
Ad hoc on-demand multipath distance vector (AOMDV), 86
Analog-to-digital converter (ADC), 12
Analog to residue converters (ARC), 12
Application-specific integrated circuit (ASIC), 13, 277
Arithmetic Labeled EXplicit (ALEX)
  achievements, 375
  approach, 373–375
Arithmetic operations, RNS
  arithmetic channels, 25–26
  binary-to-RNS conversion, 25
  RNS-to-binary conversion
    CRT algorithm, 31–33
    MRC algorithm, 32–36
ASIC. *See* Application-specific integrated circuit (ASIC)

B
Babai's rounding-off algorithm
  lattices, 348–349
  mixed RNS-MRS approach, 353–356
  RNS arithmetic
    correction technique, 359–361
    decryption process, 361
    fast RNS modular reduction, 358–359
    Montgomery reduction, 357
    MRS-based extensions, 357
  transcription, RNS
    integer operations, 351
    Montgomery's modular reduction, 352–353
Base conversion (BC), 313, 320–322
BEPPG. *See* Booth encoder and partial product generation (BEPPG)
Binary Coded Decimal (BCD), 92–94, 102
Binary Integer Decimal (BID), 99–101, 103
Binary method algorithm, 327
Binary number systems, programmable FIR filter design
  ASIC synthesis results
    critical path delays, 305, 307
    EDBNS method, 302
    gate count *vs.* number of taps, 306
    POBS block, 304
    throughput rate, 305
    TMCM implementation, 303
  FPGA synthesis, 307–308
Bit error rate (BER), 85
Bit-error-rate (BER), 271
Bit per symbol (BPS), 85
Booth-encoded multipliers
  analyses and discussions
    high-radix booth encoding *vs.* simple booth encoding, 141–142
    NBBE and RBBE, 139–141
    redundant binary coding efficiency, 142–144
  booth algorithms, redundant binary multiplier, 116–120
  converters for RBA interface, 133–134
  digital multiplication, 114
  existing RB multipliers, 122–124
  N × N-bit RB multipliers, BEPPG, 129–131
  one-digit BEPPG module, 126–128
  one-digit RB adder cells, 131–133
  performance evaluation

© Springer International Publishing AG 2017

A.S. Molahosseini et al. (eds.), *Embedded Systems Design with Special Arithmetic and Number Systems*, DOI 10.1007/978-3-319-49742-6

Booth-encoded multipliers (*cont.*)
numerical simulation results, 136–138
RB booth multipliers configurations, 134–136
RBA and carry-free adding rule, 115–116
redundant binary coding interface component, 120–122
taxonomy, booth encoders and partial product generators
NBBE, 125–126
RBBE scheme, 126
VLSI performance factors, 114
Booth encoder and partial product generation (BEPPG), 123, 124, 126–128

C

Canonical double based number system (CDBNS), 286–288
Canonical signed digit (CSD), 282, 285
Carry propagate adders (CPAs), 117
Carry save adder (CSA), 42, 122, 269, 302
Cascade Digit Generation, 150–151
CDBNS. *See* Canonical double based number system (CDBNS)
Chinese remainder theorem (CRT), 6, 20, 31–33, 50, 185, 205, 350, 370
hardware-fault tolerance, 341
non-modular operations
magnitude comparison, 53
moduli set, constraints on, 50
MOMA, 50, 53
performance analysis, 62–63
residue-to-binary conversion, 52–53
residue representation, 68
RNS-to-binary conversion, 317–319
Closest vector problem (CVP), 347, 348
"Coding overhead," 188–189
Common Subexpression Elimination (CSE), 281
Common Subexpression Generator (CSG), 288
Complex logarithmic number system (CLNS), 248
Computer networks
energy saving and reliability, WSN, 376–378
MANETs, unicast/multicast routing in (*see* Mobile Ad Hoc Network (MANETs))
redundant moduli, 369
SDN
core fabrics, RNS in, 376
unicast/multicast routing and forwarding, 370–372
Continuous valued number system (CVNS)
ADD_b and ADD_g, 164–167
analog-digits, 150
addition, 152–153
Cascade Digit Generation, 150–151
Modular Digit Generation, 152
multiplication algorithm
example, 168–168
low-resolution environment, 169–171
synapse multiplier, VLSI implementation of
current-mode circuits, 172–173
CVNS registers, 176
layout of proposed CVNS multiplier, 173–174
post-layout simulation results, 173–176
sigmoid activation function, 172
TSMC CMOS 0.18 μm technology, 171
truncated addition
ADD_{ts} and ADD_{tu}, 164–167
binary to CVNS conversion, 156
lower index digits, 155, 156
sliding groups, 157–160
truncation signal, 156
uniform groups, 161–164
two binary operands, addition of
binary to CVNS conversion, 153–154
CVNS to binary conversion, 155
reverse evolution, 155
Correction algorithms
multiple residue digit error detection and
arithmetic operation errors, 84
base extension approach, 77
comparision, 84
CRT-based approaches, 83
erroneous residue digits, 81
flowchart, 79
hardware implementation, 78
legitimate and illegitimate moduli ranges, 80
m_i-projection concept, 79, 80
procedure, 83
received residue representation, 81
redundant moduli, 77
scenarios, 82
syndrome approach, 84
single residue digit error detection
arithmetic operations, 76
coding theory approach, 74
consistency checking method, 71
erroneous residue digit, 71
error-free residue representation, 72
flowchart, 73, 76, 77
m_i-projection approach, 74

m_i-projection method, 73
mixed-radix digits, 74
modular subtraction, 75
residue-to-binary computation, 72
syndrome approach, 75
Covalent Redundant Binary Booth Encoding (CRBBE), 126, 128, 139, 140, 144
Cox–Rower architecture, 39–40, 42–44
CRT. *See* Chinese remainder theorem (CRT)
CVNS. *see* Continuous valued number system (CVNS)

D

Decimal floating point number system
addition
adders, 99
BCD, 93–94
BID, 99–101
big shifters, 99
DPD, 99–101
exponent and sign, 99
exponent difference, 97
leading zeros, 96–97
multiplexers, 99
rounding directions, 96
significand result, steps for, 98
combination and trailing significand field, 94
commercia1applications, 106–107
decimal coding schemes, 92–93
division
definition, 104
designs, 105
subtractive and multiplicative methods, 105
fused multiply add, 103–104
leading zeros, 95
multiplication
BCD, 102
BID multipliers, 103
DPD designs, 103
significands, 101–102
need for, 89–92
square root and elementary functions, 105–106
verification methods, 106
Densely Packed Decimal (DPD), 99–101, 103
Design space exploration (DSE)
characterization results, 193–195
maximum speed corner, 191
minimum area corner, 191
TCS MADD, 191–192
D flip-flops (DFFs), 257, 263
Digital signal processing (DSP), 12, 19
ASIC platforms, 210–211
fault-tolerant RNS, FIR filter
CRT block, 205
erroneous module, 205
hardware implementation, 208–209
implementation, 206
microelectronic technologies, 205
minority voter, 206
output converters, 207–208
FPGA platforms, 211–212
moduli selection
"coding overhead," 188–189
criteria for, 187
MRC, 187
power dissipation, 188
RNS base, 188
RNS background and notation
fault-tolerant techniques, 186
input conversion, 185
modular multiplication, 183–184
output conversion, 185
QRNS, 183
RNS filters
adaptive filters, 202–204
DSE, 191–195
hardware architecture, 190
parallel/serial FIR filter, 196–198
polyphase (complex) filter, 198–202
real FIR filters, 195–196
TCS, 189
RNS perspectives, 212–213
Digital-to-analog converter (DAC), 12
Direct isomorphic transformation (DIT), 183
Discrete cosine transform (DCT), 233
Discrete fourier transform (DFT), 233
Discrete wavelet transform (DWT), 233
Distinct multicast routing protocols (DVMRP), 370
Double-base coefficient generator (DBCG), 296, 302
Double-base product selector (DBPS) block, 288
DSE. *See* Design space exploration (DSE)
DSP. *See* Digital signal processing (DSP)

E

EDBNS. *See* Extended double based number system (EDBNS)
Elementary modular multiplication (EMM), 362
Elliptic Curve Cryptography (ECC), 312
GF (p), 324–326

Elliptic Curve Cryptography (ECC) (*cont.*)
point multiplication, 327
RNS application
conversion from base *B*' to base *B*, 335–336
hardware architecture, RNSMMM, 337–338
modular adders and multipliers, 334–335
RNS bases use, 333
RNSMMM algorithm, 333
Elliptic curve discrete logarithm problem (ECDLP), 324
Energy-delay product (EDP), 140
European Logarithmic Processor (ELM), 248
Exponential diophantine equation (EDE), 299–301
Extended double based number system (EDBNS)
binary and CSD representations, 293
integer *c*, 293
power-of-*b* integers, 296
programmable FIR filter design
overall design flow, 301–302
POBG, 296–298
POBS, EDBNS reduction properties and EDE, 299–301
transposed form FIR filter, 297
search algorithm, 295
w-bit integers, 294, 295

F
Fast fourier transform (FFT), 271–272
Fault-tolerant RNS, FIR filter
CRT block, 205
erroneous module, 205
hardware implementation, 208–209
implementation, 206
microelectronic technologies, 205
minority voter, 206
output converters, 207–208
Fault-tolerant techniques, redundant RNS
background and preliminaries
residue arithmetic, 68
residue-to-binary conversion, 68–69
RNS, 66–67
correction algorithms
multiple residue digit error detection and, 77–84
single residue digit error detection and, 71–77
erroneous residue digits, 71
error detection and correction, 85–87
legitimate and illegitimate ranges, 70
motivations, 65–66
residue representation, 69
Field programmable gate arrays (FPGAs), 13, 278
Finite impulse response (FIR) filter
ASIC, 277
binary number systems
ASIC synthesis results, 302–307
FPGA synthesis, 307–308
conventional number systems, 283–285
double-base number system
CDBNSs, 286–288
design method and examples, 288–292
DSP applications and public key cryptography, 285
EDBNS (*see* Extended double based number system (EDBNS))
implementations
area-delay and power-delay complexity, 267
normalized delay, 268, 270
Wallace-tree and carry-save (CSA) structures, 269
LNS MAC architectures
retimed LNS MAC unit, 267
single-MAC architecture, 265
SNR, 266
switching activity, 265
two-MAC architecture, 265
multiple constant multiplication block, 280–281
optimization of representation
logarithmic representation, 254
simulation-based methodology, 256–257
SNR, 255, 256
SDR, 277
TMCM block, 281–283
transpose direct form, 279–280
Floating point numbers
decimal floating point number system (*see* Decimal floating point number system)
IEEE 754-2008 standard, 89
single/double precision, 89
Forward converters, 5–7, 11
Forwarding Group Bitmap (FGB), 373
Full-Adder (FA) model, 42
Fused multiply add (FMA), 103–104

G
Graph-dependence (GD) algorithms, 280

H
Hardware description language (HDL), 13
Homomorphic encryption scheme, 219
Horner scheme, 363
Hybrid RNS architecture, 203–204

I
Image processing
digital image and numerical representation, 218
edge detection and sharpening filters
addition and multiplication of integers, 228
binary number system, 221
convolution operation, 220
data processing, 224
frequency and power consumption, 222
grayscale images, 220
image pixels, 225
MATLAB, 225
moduli sets, dynamic range for, 222
peak signal to noise ratio, 226–227
RNS dynamic range., 223
simulation results, 224
SSIM, 227–228
image digitization, 218
operations, 219
smoothing filters
binary number system, 232
correct operation, MATLAB, 229
division operation, subtraction and multiplication, 231
hardware implementation, HDL, 229
image denoising and quality improvement, 229
noisy images, 231
replacement operation, 229
reverse conversion operation, 230
RNS architectures, 232
RNS modulo, 230
wavelets
construction scheme, 240
cryptographic information protection systems, 242
Daubechies wavelet Db4, 234
DWT, 233
finite-field wavelets, 236
Fourier transform, 233
image filtering scheme, 235
mathematic microscopes, 236
MATLAB, 234
matrix, 239
RGB representation, 242
signal processing systems, 243
three-level hierarchy wavelet decomposition, 237
trivial filterbank, 242
Indirect isomorphic transformation (IIT), 183
Instruction Set Architecture (ISA), 20
arithmetic and conversion operations, 22
32-bit instruction size, 22
definition, 22
instruction format, 22–23
single and multi-cycle instructions, 22–24
Integrated Circuit Compiler (ICC), 190
Inverse Discrete Fourier Transform (IDFT) unit, 199

J
Joint Photographic Experts Group (JPEG), 233

K
Karatsuba's algorithm, 364
"Kawamura et al.", base conversion, 321–322

L
Lagrange's interpolation, 350
Lattice-based cryptography (LBC)
asymptotic computational efficiency, 346
Babai's rounding-off algorithm, 348–349
complexity analysis
multi-precision approach, 364–365
RNS-based approaches, 362–364
CPU/GPU, RNS/MRS approach, 365–366
general integer lattices, 346
hard problems, 346–348
key-exchange protocol and RSA, 345
mixed RNS-MRS approach, Babai's rounding-off algorithm, 353–356
RNS arithmetic, Babai's rounding-off algorithm
correction technique, 359–361
decryption process, 361
fast RNS modular reduction, 358–359
Montgomery reduction, 357
MRS-based extensions, 357
transcription, RNS
adapting Babai's rounding-off algorithm, 351–353
notations, 350–351
trapdoor functions, 349–350
Lattice Vector Quantizer, 233
Least significant bit (LSB), 121, 257, 283

Logarithmic Maximum A Posteriori (Log-MAP) algorithm, 248
Logarithmic number system (LNS)
base optimization, 252–254
basics of, 250–252
circuits design
LNS adder/subtractor organization, 257, 258
LUT subsystem, 258–261
sub-LUT selection strategies, 261–263
data representation, 247
design methodology, 249–250
and FFT, 271–272
FIR filters
implementations, 267–270
LNS MAC architectures, 263–267
optimization of representation, 254–257
look-up tables, 248
power dissipation, 249
schemes, 272–273
symbol-by-symbol Log-MAP algorithm, 248
Look-up tables (LUTs)
control signals, 301
input converters, 185
LUT subsystem
active sub-LUT, 260
complexity reduction, 261
design-space exploration, 261
LSBs, 259
memory subsystem, 258
MSBs, 259
stored values, 259
sub-LUT selection strategies, 261–263
Low-Density Parity Check (LDPC), 248

M

Maximum likelihood decoding (MLD), 80
Maximum likelihood estimation (MLE), 339
Mentor Graphics LeonardoSpectrum, 303
Minimal signed digit (MSD), 284, 285
Mitchell's algorithm (MA), 248
Mixed-radix conversion (MRC), 6, 20, 31–36, 185
hardware-fault tolerance, 340
non-modular operations, 50–52, 62–63
residue representation, 68
RNS-to-binary conversion, 319–320
Mixed radix system (MRS), 24, 319, 351
Mobile Ad Hoc Network (MANETs), 370
ALEX
achievements, 375
approach, 373–375
CRT, 372
efficient routing protocol, 372
Modular Digit Generation, 152
Modular multiplication and-accumulation (MMAC) units, 40–41, 43–44
Monte Carlo simulation, 173, 175–176
Montgomery modular multiplication (MMM), 312, 328
Most significant bits (MSBs), 6, 121, 194, 257
Most significant digit (MSD), 120
Moving Picture Experts Group (MPEG), 233
MRC. *See* Mixed-radix conversion (MRC)
Multicast Channel-specific IDentifier (MCID), 374
Multicast Routing Table (MRT), 370
Multi-operand modular adders (MOMA), 50, 53
Multiple constant multiplication (MCM), 278, 280–281
Multiple-Valued-Logic (MVL), 149
Multiply-and-accumulate (MAC), 114, 197, 202, 332

N

NBBE. *See* Normal binary Booth encoding (NBBE)
New Chinese Remainder Theorem I (New-CRT-I), 31
New Chinese Remainder Theorem II (New-CRT-II), 31
Newton-Raphson iterations, 106
Newton's interpolation, 351
Non-modular operations in RNS
core functions, magnitude comparison, 50
CRT
magnitude comparison, 53
moduli set, constraints on, 50
MOMA, 50, 53
performance analysis, 62–63
residue-to-binary conversion, 52–53
diagonal function, 50, 57–58
coefficients, property of, 56
diagonal modulus, characterization of, 54–55
integers, 53–54
labels of diagonals, 53–54
monotonicity, 56–57
performance analysis, 62–63
mixed-radix conversion process, 50–52, 62–63
quotient function, 50–51
binary representations of X, 62
labels of diagonals, 58–59

magnitude comparison, 62
not relatively prime moduli, 61–62
pairwise relatively prime moduli, 59–61
performance analysis, 62–63
residue-to-binary conversion, 62
residue-to-binary conversion, 50
Normal binary Booth encoding (NBBE), 125–126

O
Output Port Bitmap (OPB), 371
Output Port Index (OPI), 371

P
Parallel-prefix computations, 3
Parallel/serial FIR filter, 196–198
Partially redundant biased Booth encoding (PRBBE), 126
Partial product generator (PPG), 125
Peak signal to noise ratio (PSNR), 226–227
Polyphase (complex) filter
clock gating, 200
complex TCS, 199
EXP-3, 200, 201
EXP-4, single serial complex filter, 202
IDFT unit, 199
operational frequency, 201
power dissipation, 200
QRNS base, 199
structure, 201
Power-of-*b* generator (POBG), 296–298
Power-of-*b* selector (POBS), 299–301
Processors, RNS
architecture, 20–22
arithmetic operations
arithmetic channels, 25–30
binary-to-RNS conversion, 25
RNS-to-binary conversion, 31–36
control units
global control unit, 36
multi-cycle operations, condensed micro-code for, 36–37
relative co-prime numbers, 37–38
ISA, 20
arithmetic and conversion operations, 22
32-bit instruction size, 22
definition, 22
instruction format, 22–23
single and multi-cycle instructions, 22–24
state-of-the-art analysis
Cox–Rower architecture, 39–40, 42–44
MMAC units, uRNS, 40–41, 43–44
RDSP, 38–39
TTA approach, 40
Public-key cryptography
ECC
GF (p), 324–326
point multiplication, 327
residue arithmetic
binary-to-RNS conversion, 314–316
CRT, 311
cryptosystem designs, 312
mathematical background, 312–314
modular multiplication, 312
RNS-to-binary conversion, 316–322
RNS application
ECC, 333–338
RSA, 330–332
RNS modular multiplication, 328–330
RSA
digital signatures, 324
encryption/decryption, 323
key establishment, 323
RSA-CRT
digital signature operation, 338
fault-infective techniques, 339
hardware-fault tolerance, 340, 341
modulus expansion, 339
safe-error concept, 339

Q
Quadratic Residue Number System (QRNS), 183
Quality of Service (QOS), 375

R
RB full adders (RBFA), 131
RB partial products (RBPP), 124
Reconfigurable modulo (RM) adder, 335
Reconfigurable modulo (RM) multiplier, 335
Reduced Instruction Set Computer (RISC), 39
Redundant binary adders (RBAs), 114
Redundant binary Booth encoding (RBBE), 119, 125
Redundant binary representation, booth-encoded multipliers
analyses and discussions
high-radix booth encoding *vs.* simple booth encoding, 141–142
NBBE and RBBE, 139–141

Redundant binary representation, booth-encoded multipliers (*cont.*)
redundant binary coding efficiency, 142–144
booth algorithms, redundant binary multiplier, 116–120
converters for RBA interface, 133–134
digital multiplication, 114
existing RB multipliers, 122–124
N x N-bit RB multipliers, BEPPG, 129–131
one-digit BEPPG module, 126–128
one-digit RB adder cells, 131–133
performance evaluation
numerical simulation results, 136–138
RB booth multipliers configurations, 134–136
RBA and carry-free adding rule, 115–116
redundant binary coding interface component, 120–122
taxonomy, booth encoders and partial product generators
NBBE, 125–126
RBBE scheme, 126
VLSI performance factors, 114
Redundant Residue Number System (RRNS), 186, 205
Reed–Solomon (RS) codes, 85
Residue arithmetic
binary-to-RNS conversion, 314–316
CRT, 311
cryptosystem designs, 312
mathematical background, 312–314
modular multiplication, 312
RNS-to-binary conversion
base conversion, 320–322
CRT, 316–319
MRC, 319–320
Residue digital signal processor (RDSP), 38–39
Residue number system (RNS), 49–50
abilities, 3, 13
applications, 3–4
approach in (*see* Lattice-based cryptography (LBC))
components, 4
computer networks (*see* Computer networks)
CVNS (*see* Continuous Valued Number System (CVNS))
DSP (*see* Digital signal processing (DSP))
dynamic range, 4
image processing (*see* Image processing)
non-modular operations (*see* Non-modular operations in RNS)
processor design (*see* Processors, RNS)
properties, 4
public-key cryptography (*see* Public-key cryptography)
structure
arithmetic-friendly moduli sets, 4
conversion-friendly moduli sets, 4
extended moduli sets, 5
forward converters, 5–7
magnitude comparators, 6, 7
modulo adders and multipliers, 6, 7
overflow detection, 6, 7
prime moduli sets, 5
reverse converters, 6, 7
scaling, 6, 7
sign detection, 6, 7
special moduli sets, 4, 6
teaching methodology, 9
application, embedded systems design, 12
ASIC/FPGA implementation, 13
basic concepts, 10
courses, 8
forward converter, 11
hard RNS operations, 12
modular adders and multipliers, 10–11
phases, 8
reverse converters, 11–12
Residue Number System Product Code (RNS-PC), 186
Reverse converters, 6, 7, 11–12
Reverse Evolution, 155
Ripple-carry adder (RCA), 10, 11
Rivest-Shamir-Adleman-system (RSA)
digital signatures, 324
encryption/decryption, 323
key establishment, 323
RNS. *See* Residue number system (RNS)
Round to Nearest with ties Away from zero (RNA), 96
Round to Nearest with ties to Even (RNE), 96
Round To Negative (RTN), 96
Round To Positive (RTP), 96
Round To Zero (RTZ), 96
Round trip time (RTT), 376
Rower architecture, 39–40
RRNS. *See* Redundant Residue Number System (RRNS)
RSA. *See* Rivest-Shamir-Adleman-system (RSA)

S
Scalar-pair vectors routing and forwarding (SVRF), 370–372
Shenoy and Kumaresan's technique, 358
Shortest vector problem (SVP), 346–348
Signal-to-noise ratio (SNR), 85, 249, 255, 266
Signed digit (SD) representation, 284
Signed-Power-of-Two (SPT), 284
Single event upset (SEU), 205
Single instruction multiple data (SIMD), 366
Six-Moduli RRNS (6 M-RRNS), 85, 86
Software-defined network (SDN), 370–372
Software defined radio (SDR), 277
Structural SIMilarity index (SSIM), 227–228
Synapse multiplier, CVNS
  current-mode circuits, 172–173
  CVNS registers, 176
  layout of proposed CVNS multiplier, 173–176
  post-layout simulation results, 173–176
  sigmoid activation function, 172
  TSMC CMOS 0.18 μm technology, 171
Synopsys Design Compiler, 136

T
Time-multiplexed multiple constants multiplication (TMCM), 281–283
Transport triggered architecture (TTA), 40
Triple modular redundancy (TMR), 205
Two's complement system (TCS), 66, 189, 204

W
Wallace-tree, 269
Watermarking algorithms, 219
Weighted number system, 6
Wireless sensor network (WSN), 369, 376–378

X
Xilinx Virtex-E FPGA, 211–212