VLSI efficient RNS scalars and arbitrary modulus residue generators

Low, Jeremy Yung Shern

2014

Low, J. Y. S. (2013). VLSI efficient RNS scalars and arbitrary modulus residue generators.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/59524>

<https://doi.org/10.32657/10356/59524>

**VLSI Efficient RNS Scalers And
Arbitrary Modulus Residue Generators**

Jeremy Low Yung Shern

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

June 2013

Acknowledgement

First and foremost, I would like to thank Associate Professor Chang Chip Hong for his supervision during my Ph.D candidature. His passion in research and his high standard propel me to strive for the best in my research work. I am also very grateful to him for his constructive criticism and insightful discussions in improving the manuscript of our publications and this thesis. His expertise in the research area as well as his broad technical knowledge are very important in paving me the correct research direction. Last but not least, I sincerely appreciate the effort and time he spent to instil the right working attitude in me.

I would also like to thank the technical staffs of ViRTUS research centre for their kind support and assistance. I am also grateful for the research facilities and environment provided by the centre.

I would also like to thank Professor Li Qiang for providing me an opportunity to do my research attachment in Aarhus University, Denmark during my final year. I am very grateful to him for funding my travel expenses and accommodation during my stay in Denmark. Also, I appreciate the support of the faculty and administrative staffs that make my experience in different research culture a memorable and meaningful one.

Table of Contents

Acknowledgement	i
Table of Contents	iii
Summary	vi
List of Figures	viii
List of Tables	x
List of Abbreviations	xii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	5
1.3 Main Contributions.....	6
1.4 Organization of the thesis	7
Chapter 2 Background and Literature Review.....	9
2.1 Overview of Residue Number System	9
2.2 Residue Arithmetic	12
2.3 Binary-To-RNS Conversion.....	13
2.4 RNS-To-Binary Conversion.....	14
2.5 Review of RNS Scalers	16
2.5.1 Two-Lookup Cycle Scaling Technique [Gar99]	18
2.5.2 Fast Scaling Scheme [Kon09]	21
2.5.3 Full-Adder Based Exact Scaling [Das08].....	22
2.5.4 RNS Scaling by 2^n for Moduli Set $\{2^n-1, 2^n, 2^n+1\}$ [Ye08].....	24
2.5.5 A 2^n RNS Scaling Scheme [Ma10].....	27
2.5.6 Programmable Power-Of-Two Scaler [Car05b].....	28
2.6 Review of Binary-To-RNS Converters	30

Chapter 3 Proposed Fixed Scaler for The Three-Moduli Set $\{2^n-1, 2^n, 2^{n+1}\}$ RNS	34
3.1 Introduction	35
3.2 Proposed Scaling Algorithm.....	37
3.2.1 Preliminaries	37
3.2.2 New Formulation of RNS Scaling.....	38
3.2.3 Error Analysis.....	42
3.3 Hardware Implementation	44
3.4 Evaluation and Comparison.....	51
3.5 Conclusion	57
Chapter 4 A VLSI Efficient Programmable Power-Of-Two Scaler for $\{2^n-1, 2^n, 2^{n+1}\}$ RNS	58
4.1 Introduction	58
4.2 Proposed Algorithm for Programmable Power-Of-Two Scaling in RNS	60
4.3 Hardware Implementation	67
4.4 Performance Evaluation and Comparison	71
4.4.1 Theoretical analysis based on unit-gate model.....	71
4.4.2 Synthesis Results	73
4.5 Conclusion	76
Chapter 5 A New Approach to the Design of Efficient Residue Generators for Arbitrary Moduli.....	78
5.1 Introduction	78
5.2 Proposed Residue Generators for Arbitrary Moduli.....	79
5.2.1 Partitioned Column Compression Of Partial Residue Bits.....	80
5.2.2 Elimination of Carry Propagation Addition	86
5.2.3 Modified Modulo m Adder.....	88
5.3 Performance Analysis and Comparison	91
5.3.1 Hardware Resource Analysis of Proposed Design Method for $L = 64$	91

5.3.2 Comparison with Published Results of Parallel Residue Generators for $L = 32$	93
5.3.3 Comparison of Synthesis Results with Memory-Free and Memory-Based Residue Generators for Arbitrary Moduli of Diverse Periodicities.....	95
5.4 Conclusion	97
Chapter 6 Conclusions and Future Work.....	99
6.1 Conclusions	99
6.2 Future Work.....	100

Summary

Carry propagation has been identified as the main timing bottleneck of the datapath elements of application-specific digital signal processors in the accustomed positional number system. Residue Number System (RNS), being a non-weighted number system, emerges as a good remedy to this problem. RNS gains phenomenal successes in speeding up the datapath of digital signal processing applications dominated by additions/subtractions and multiplications, such as digital filters, convolution, channelizers, equalizers, discrete transforms, etc. Despite the rapid development of RNS, it is still relatively inefficient in handling some inter-modulo operations such as data scaling directly in the residue domain. As a workaround, hybrid RNS-binary system has been widely used in the embedded inner-product-step-processor (IPSP) architecture of these applications to scale the intermediate results in binary domain to prevent overflow errors. This has resulted in a humongous amount of research on residue-to-binary conversion problems especially for special moduli set with good number theoretic properties. Such hybrid number system is not ideal as it relies heavily on the efficient RNS reverse converter, which is itself an inter-modulo operation that annihilates the modularity and parallelism of RNS. This thesis tackles the RNS scaling problem by eliminating the area intensive and slower residue-to-binary converter for a more efficient implementation of true RNS-based IPSP.

A novel area-efficient, high-speed and precise RNS scaler for the celebrated three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ is proposed in this thesis. The new scaling algorithm is formulated based on the Chinese Remainder Theorem and it uniquely exploits the number theoretic properties of this moduli set and the fixed scaling factor of 2^n to overcome the complexity associated with the hardware implementation of this inter-modulo operation. The proposed RNS scaler has an area complexity of $O(n \log_2 n)$ and a time complexity of $O(\log_2 n)$. The integer scaled output in normal binary number representation is also generated as a byproduct in this new formulation. Hence, the expensive residue-to-binary converter can be saved if the result after scaling is also required by a normal binary number system.

To extend the usefulness of this moduli set to RNS-based adaptive signal processing applications, another elegant algorithm is developed to enable programmable power-of-two RNS scaling for the first time. With a variable scaling factor of 2^r , $0 \leq r \leq n$, up to one-third

of the dynamic range of an integer can be arbitrarily scaled in the residue domain directly. The architecture can be implemented entirely in combinational circuits without lookup tables, making it easy to be merged and pipelined with other circuits within the RNS. Its simplicity contributes to the simultaneous reduction of area, delay and power consumption in its hardware implementation. As the binary scaled output can also be made available by the proposed method, it will also ease the magnitude comparison of scaled integers.

While RNS constructed by moduli of the form 2^n and $2^n \pm 1$ possesses good number theoretic properties, it has limited parallelism and asymmetrical modulus wordlengths due to the limited number of moduli that can be selected to fulfill the relative primality requirement of RNS. The size of one or more moduli has to be increased as the dynamic range increases, which leads to the degradation in overall system performance for high dynamic range computations. Rather than enlarging the sizes of one or more moduli to extend the dynamic range of an RNS, the cardinality of the RNS can be increased by the use of arbitrary moduli set as the complexity of the RNS-to-binary converter is cardinality insensitive once the cardinality exceeds certain threshold. For high dynamic range applications, a valid RNS can be formed with relative ease by selecting as many moduli as desired from plentiful small integers. This thesis also addresses the performance bottleneck of the generation of different arbitrary residues of diverse cyclic periodicity to prevent the advantages of balanced and high-cardinality general RNS from being offset by their hardware implementation overheads.

A new approach to the unitary design of efficient residue generators for arbitrary moduli is thus presented. The proposed design requires at most seven stages of carry save addition (CSA), one lookup table (LUT) of no more than seven-bit input and a small modular adder for input wordlength as large as 64 bits and modulus of up to six bits wide, making it the fastest, smallest and most power efficient residue generator architecture for any input and modulus within these ranges. It has significantly reduced the area of the fastest memory-based design and the timing of the most efficient memoryless design reported thus far. Moreover, the disparity due to the inconsistent periodicity of different moduli of the latter design has been minimized by the proposed depth-constrained CSA tree and periodicity independent LUT and modified modular adder. The latters are made possible by the ingenious use of distributive property in place of periodicity property to limit the width of the CSA tree so that large size LUTs and modular adder tree can be eliminated.

List of Figures

Figure 1.1: (a) IPSP with true RNS scalers (b) IPSP with binary scalers in hybrid RNS	4
Figure 2.1: Two look-up cycle RNS scaler.....	20
Figure 2.2: Two look-up cycle RNS scaler for RNS $\{29, 31, 32\}$ and $k = 29$	20
Figure 2.3: Architecture of fast RNS scaling scheme.....	21
Figure 2.4: Block diagram of RNS scaler design in [Das08]	23
Figure 2.5: 2^n RNS scaler for moduli set $\{2^n-1, 2^n, 2^n+1\}$	27
Figure 2.6: Architecture of parallel 2^n RNS scaler	28
Figure 2.7: Block diagram of the scaling-by-2 block	30
Figure 2.8: LUT-based residue generator	33
Figure 2.9: Block diagram of residue generator in [Pre06]	34
Figure 3.1: Reduction of addends	46
Figure 3.2: Simplification of the CSA tree for the computation of y_3	48
Figure 3.3: Architecture of the proposed RNS scaler	48
Figure 3.4: Calculation of scaled RNS output, y_1, y_2 and y_3 for Example 3.2	50
Figure 4.1: Architecture of proposed RNS programmable power-of-two scaler based on moduli set $\{2^n-1, 2^n, 2^n+1\}$	68
Figure 4.2: 8-bit 3-stage logarithmic cyclic right shifter for the generation of P_1	69
Figure 4.3: 8-bit 4-stage logarithmic left shifter for the generation of P_2	69
Figure 4.4: 8-bit 4-stage logarithmic complementary cyclic right shifter (output AND with $\overline{(x_3)_8}$) for the generation of Q_1	70
Figure 4.5: n -bit CSA with CEAC of modulus 2^n+1 channel.....	70
Figure 5.1: Block diagram of proposed residue generator.....	80
Figure 5.2: Dot matrix diagram of the CSA for the partial residue array of $X \bmod 29$ after Hamming weight reduction.....	84

Figure 5.3: Reduction of CSFA (left partition) and CSHA (right partition) to two r -bit operands for $m = 29$	86
Figure 5.4: (a) 2-bit CLA without carry input (b) Reduction of LUT size by 2-bit CLAs at level 9 of CSA for $m = 61$	88
Figure 5.5: Block diagram of the modulo 29 adder modified from [Hia02]	90
Figure 5.6: (a) The bit slices of SAC unit, (b) basic cells for CPG unit, and (c) gate-level structures for the CLAs that generate C_{out} and C_{out}^*	91
Figure 5.7: Proposed design approach to modulus m residue generator.....	92

List of Tables

Table 2.1: Computation of actual residue digits of scaled number.....	23
Table 2.2: Overview of RNS scalars.....	30
Table 2.3: Periodicity p for various moduli m_i	32
Table 3.1: Computation traces of scaling (80, 132, 29) by $k=256$ in RNS {255, 256, 257} ..	42
Table 3.2: Estimation of unit-gate area of proposed design	51
Table 3.3: Estimation of unit-gate delay of proposed design	52
Table 3.4: Estimated number of transistors of proposed design.....	53
Table 3.5: Comparison of estimated number of transistors (Value in parenthesis indicates the percentage area reduction of proposed design over its contenders)	53
Table 3.6: Comparison of estimated unit-gate delay (value in parenthesis indicates percentage delay reduction of proposed design over its contenders).....	54
Table 3.7: Comparison of estimated total number of transistors (A) and unit-gate delay (T) for the same three moduli sets	55
Table 3.8: Comparison of the synthesized area and delay of $\{2^n - 1, 2^n, 2^n + 1\}$ RNS scalers for $n = 5, 7$ and 8	56
Table 3.9: Comparison of total power and leakage power in mW (value in parenthesis is the percentage of leakage power) for $n = 5, 7$ and 8	57
Table 4.1: Scaling of (230, 97, 13) by 2, 4, 32 and 256 in RNS {255, 256, 257}	67
Table 4.2: Unit-gate area analysis of proposed design	72
Table 4.3: Unit-gate delay analysis of proposed design	72
Table 4.4: Unit-gate-area and delay analysis of hybrid design.....	73
Table 4.5: Comparison of estimated unit-gate areas and delays.....	73
Table 4.6: Comparison of the synthesized areas and delays of $\{2^n - 1, 2^n, 2^n + 1\}$ RNS scalers for $n = 5, 6, 7$ and 8	74

Table 4.7: Comparison of total power in mw and leakage power in μw @ $210MHz$ for $n = 5, 6, 7$ and 8	76
Table 4.8: Comparison of total power $\times$ delay in pJ and leakage power $\times$ delay in fJ for $n = 5, 6, 7$ and 8	76
Table 5.1: Coefficients and partial residues of $X \bmod 29$	81
Table 5.2: Columns height of the partial residue array for different modulus before and after Hamming weight reduction ($L = 64$)	85
Table 5.3: Column heights of left and right partitions for $m = 61$	88
Table 5.4: Hardware resource requirement of proposed residue generators before the final mod m adder for different m ($L = 64$)	93
Table 5.5: Comparison between [Pre06], [Pre02], [Moh94] for $m = 19$ and $L = 32$	94
Table 5.6: Area (μm^2) and delay (ns) comparison of different residue generators (number in parenthesis indicates percentage reduction of proposed design over other design in comparison).....	96
Table 5.7: Total power (μW) and Leakage power (μW) comparison of different residue generators (number in parenthesis indicates percentage reduction of proposed design over other design in comparison).....	97

List of Abbreviations

ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuits
CEAC	Complemented End Around Carry
CLA	Carry Look Ahead
CLAS	CLA and Summation
CLS	Circular Left Shift
CPA	Carry Propagating Adder
CPG	Carry Propagate and Generate
CRS	Circular Right Shift
CRT	Chinese Remainder Theorem
CSA	Carry Save Addition
CSFA	Carry Save Full Addition
CSHA	Carry Save Half Addition
DSP	Digital Signal Processing
DWT	Discrete Wavelet Transform
EAC	End Around Carry
FA	Full Adder
FF	Flip Flop
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
HA	Half Adder
HAL	Half Adder Like
IIR	Infinite Impulse Response
IPSP	Inner Product Step Processor
LSB	Least Significant bit
LUT	Look Up Table
MFA	Modified Full Adder
MOMA	Multi-Operand Modulo Adder
MRC	Mixed Radix Conversion
MUX	Multiplexer
PE	Processing Element

PFP	Pseudo Floating Point
RCA	Ripple Carry Adder
RNS	Residue Number System
ROM	Read Only Memory
SAC	Sum And Carry
TCS	Two's Complement number System

Chapter 1

Introduction

1.1 Motivation

Two's complement number system (TCS) has been used for more than 60 years to design digital blocks in various digital systems, such as mainframe computers, microprocessors, digital signal processors (DSP) and application specific integrated circuits (ASIC). Ever since then, humongous amount of research have been carried out to improve the performance of the two's complement digital blocks. Today, to keep up with the relentless pursuit for low power and high performance digital signal processors with higher application versatility, digital designers are compelled to look into all design abstraction levels, from algorithmic level down to logic or even transistor level, to reduce the power consumption and to increase the speed of the digital blocks. The latter has been the main motivation to forgo the ease of programmability for dedicated hardware implementation of complicated DSP algorithms, and it is still one of the key criteria in today's digital IC design. Carry propagation in the arithmetic block has been identified as the main culprit that slows down the datapath. This timing bottleneck is rooted from the native positional weighting of TCS. Over the years, researchers and designers have come up with many brilliant ideas to reduce the carry propagation delay associated with TCS. Some of the popular solutions to speed up the carry propagation addition of two operands include carry skip/bypass adder, carry select adder, carry lookahead adder, sparse tree adder and parallel prefix adder. All these high-speed adders trade area and power for speed. The hardware complexity as well as the fanouts of primitive logic gates used in the high-speed structures are usually very large, leading to high power dissipation.

Residue Number System (RNS), a non-weighted alternative algebraic structure, emerges as a savior for efficient hardware implementation of some commonly used arithmetic operations

in DSP algorithms. Representing numbers using remainders, RNS decomposes large wordlength number into several smaller wordlength numbers with respect to a set of moduli [Moh02], [Omo07], [Sza67]. Arithmetic operations, such as addition, subtraction and multiplication, that are operating on the reduced wordlength numbers are carried out independently and parallelly in their respective modulus channels. By breaking the carry propagation chain and introducing subwordlength parallelism into algorithms, RNS-based architectures exhibit greater performance than their TCS counterparts. Being relatively easier to meet a target timing specification, voltage scaling and multi-threshold voltage cells in dynamic voltage-frequency scaling design methodologies can be utilized more conveniently in RNS-based architecture to reduce the dynamic power and static power [Car00], [Car05a], [Car07], [Mah00], [Sto01]. Furthermore, due to its modularity and parallelism characteristic, RNS-based arithmetic units are more area-efficient (with lower complexity of localized interconnections) and lower switching activity, leading to further reduction of the total power.

Despite the advantages of RNS, its use for the design of general purpose digital signal processors is limited. Due to the non-weighted nature of the algebraic structure, RNS suffers from the drawback of inefficient inter-modulo operations. Unlike addition, subtraction and multiplication, inter-modulo operations, such as RNS-to-binary conversion, overflow and sign detection, magnitude comparison, division, and scaling (which is the division of an integer variable by a constant), require the information of all modulus channels to produce the correct outputs. The annihilation of the modularity and parallelism characteristics for these operations causes their hardware implementation to be relatively area intensive and slower. For this reason, RNS is well received in some applications for which the datapath is dominated by additions/subtractions and multiplications. An inner-product-step-processor (IPSP) consists of a series of multiply-add-delay operations. Such IPSP-like architectures [Con04], [Lim99], [Lim07], [Re01], [Sha85], [Sod86] are prevalent in finite impulse response (FIR) filter, infinite impulse response (IIR) filter, discrete wavelet transform (DWT), fast Fourier transform (FFT), convolution, etc. Although RNS has been proven to be useful in the datapath optimization of these DSP applications, the iterative IPSP computations often result in temporarily expansion of dynamic range. Hardware wastage will be excessive if the operators are designed for the worst case dynamic range. On the other hand, overflow errors will escalate if the sizes of all operators are fixed at the desired output precision. Scaling is a necessary evil in RNS to limit the dynamic range of the intermediate results to prevent overflow and to reduce the hardware complexity. In some applications, to minimize over-

scaling, adaptive scaling is preferred, whereby the scaling constant, usually a power-of-two, is determined during runtime [Tex99]. Besides, variable power-of-two scaling is an implicit operation in floating point system to manipulate real numbers of wide dynamic range [Par10].

Figure 1.1 (a) depicts the architecture of an RNS-based IPSP. The architecture can be divided into three main parts: binary-to-RNS converter and RNS-to-binary converter to interface with the front end TCS data source and back end TCS data receiver, respectively, and RNS computation stages. Each computation stage is followed by an RNS scaling block. As the scaling block is required more than once, bulky and slow scaling blocks will bloat the hardware complexity and become the performance bottleneck of the system. To avoid the use of complex and expensive RNS scaling block, a hybrid RNS architecture, as shown in Figure 1.1 (b), is one frequently considered resort. In this solution, the scaling operation is carried out in TCS and is sandwiched between the binary-to-RNS and RNS-to-binary converters. The effort to bypass the difficult scaling operation in the residue domain is in vain as the RNS-to-binary conversion is itself an equally complicated inter-modulo operation in RNS.

The choice of the base of RNS has a direct and significant impact on the complexity of the hardware implementation of its operations [Nav11], [Abd95]. Moduli of the form 2^n and $2^n \pm 1$ possess good number theoretic properties [Moh02], [Omo07], [Sza67] that enables efficient implementation of RNS operations, especially the problematic inter-modulo operations [Abd95], [Bak09], [Cha11], [Lot10], [Moh98], [Moh07a], [Moh07b], [Moh08], [Mur08], [Mur11], [Mur12], [Pie02], [She12], [Ver02], [Ver09], [Ver10]. On the downside, RNS constructed by the moduli of 2^n and/or $2^n \pm 1$ has limited dynamic range due to limited number of modulus channels (cardinality) that can be formed under the constraint of relatively prime criterion. To increase the dynamic range, the size of one or more moduli has to be increased, leading to the degradation in overall system performance. Recent studies indicate that RNS computations offer significant delay tolerance against process-induced parameter variations with properly selected bases [Kou10], [Kou12]. It is found that larger modulo operation increases circuit timing uncertainty and causes a larger delay variation of the datapath it resides. From variation-tolerant perspective, RNS constructed by a large number of small and balanced moduli is preferable. Besides, it is shown that the complexity of the RNS-to-binary converter is cardinality insensitive once the cardinality exceeds certain threshold [Bha99]. Hence, it is better to increase the cardinality rather than enlarging the sizes of one or more moduli to extend the dynamic range of an RNS. Fortunately, a valid RNS can be formed with

relative ease by selecting as many moduli as desired from plentiful small integers without restriction to fulfil the relative primality criterion and meet the dynamic range requirement. The modulo arithmetic operations on the residues are efficient enough to meet most requirements provided that the size of the arbitrary modulus is limited to a small word length of six bits or less, which can be easily met by increasing the number of moduli. The generation of many arbitrary residues for every input operand has now become the throughput bottleneck of such a balanced and high-cardinality general RNS. Due to the difference in cyclic periodicity, the time taken to generate the residues in different modulus channel and for different operand lengths tends to be highly inconsistent. This is a problem that has been overlooked in the existing architectures of arbitrary residue generators.

Figure 1.1: (a) IPSP with true RNS scalars (b) IPSP with binary scalars in hybrid RNS

1.2 Research Objectives

Motivated by the merits of arithmetic friendly special moduli and the significance of true RNS scaler in IPSP, this thesis aims to develop efficient RNS scaling algorithms and architectures for the three moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. An appropriate fixed scaling constant and a reasonable range of programmable scaling factor will be selected after an extensive review on past works of RNS scaling. The strength and drawback of different problem formulation methods as well as the implication of different types of scaling constants will be studied in order to propose new designs that will outperform existing state-of-the-art scalars in terms of one or more of the VLSI metrics, i.e., area, delay and power consumption for different dynamic ranges. In view of the limited parallelism of special moduli set, this research also aims to resolve one of the key components and a stumbling block of general moduli set RNS, which are the arbitrary modulus residue generators. A generalized approach to the design of efficient residue generators for arbitrary moduli will be proposed after carefully scrutinizing the existing approaches and architectures of residue generators.

The following key problems have been identified and investigation will be carried out to fulfill the main objective of this research.

- (a) To investigate RNS scaling problem formulation methods of Chinese Remainder Theorem (CRT) and least integer function of division.
- (b) To break the inter-modulus dependency of existing RNS scaling algorithms.
- (c) To formulate the first solution to programmable power-of-two scaling directly in RNS
- (d) To eliminate the bottleneck carry propagation additions and modular adder tree of existing residue generators.
- (e) To reduce the architectural disparity for different moduli of inconsistent cyclic periodicity.

1.3 Main Contributions

The main contributions of the research work performed are highlighted as follows:

(1) A simple, fast and precise scaling algorithm for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ is proposed. The complexity of inter-modulo operation has been resolved by a new formulation of scaling an integer in RNS domain by one of its moduli. By elegant exploitation of the Chinese Remainder Theorem (CRT) and the number theoretic properties for this moduli set, the complicated inter-modulo operation has been reduced to a circuit that is as simple as multi-operand modulo addition which can be readily implemented by a standard cell based design methodology. The low cost VLSI architecture without any read-only memory (ROM) makes it easier to be fused into and pipelined with other residue arithmetic operations of an RNS-based processor to increase the throughput rate. The proposed RNS scaler incurs no integer scaling error. Besides the scaled residue numbers, the scaled integer in binary representation is produced as a byproduct of this process, which saves the RNS-to-binary converter when the binary scaled integer is also needed.

(2) The first solution to the programmable power-of-two scaling problem in $\{2^n - 1, 2^n, 2^n + 1\}$ RNS is proposed which possesses the most efficient hardware architecture reported thus far. The proposed scaling algorithm breaks the inter-modulus dependency and produces a parallel architecture incurring no more than two logarithmic shifters, one-stage of carry-save adder and a modulo adder in any modulus channel. The proposed programmable power-of-two scaler can efficiently scale an RNS number by up to one third of the dynamic range, which is very useful for many applications. The architecture can be implemented entirely in combinational circuits without lookup tables, making it easy to be merged into and pipelined with other circuits within the RNS.

(3) A new design method of efficient residue generators is proposed and the design approach is demonstrated with large input wordlength of 64 bits for arbitrary moduli of up to six bits. The proposed design requires at most seven stages of CSA, one LUT of no more than seven-bit inputs and a small modular adder for input wordlength as large as 64 bits and modulus of up to six bits wide, making it the fastest, smallest and most power efficient residue generator for any input and modulus within these ranges. More importantly, the proposed design

eliminates the bottleneck carry propagation additions and modular adder tree of existing designs, and circumvents the undesirably high architectural disparity for different moduli of inconsistent cyclic periodicity by the ingenious use of distributive property in place of periodicity property to limit the width of the CSA tree.

The above contributions have led to the publications listed in the author's publications towards the end of the thesis.

1.4 Organization of the thesis

This thesis is organized into six chapters. In Chapter 1, the motivation, the objective and the key contributions of the research work are detailed.

In Chapter 2, an overview of RNS is presented. The fundamental concepts on residue arithmetic, binary-to-RNS converter and RNS-to-binary converter are described. Subsequently, the RNS scaling algorithms for general moduli set as well as the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ are comprehensively surveyed. An extensive literature review on residue generators for arbitrary moduli is also performed.

The main contributions of this thesis are presented in Chapters 3 to 5. In Chapter 3, a new and simple adder-based RNS scaler for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ is proposed. New formulation of RNS scaling to reduce the complexity of inter-modulo operation of RNS scaling is derived in detailed. A systematic analysis is made to validate the zero integer scaling error of the proposed RNS scaling algorithm. A low area, fast and power efficient RNS scaler architecture is derived from the proposed algorithm. The synthesis results are analyzed and compared with other memory-based and adder-based fixed scaler designs.

In Chapter 4, an elegant algorithm that performs programmable power-of-two scaling directly in RNS $\{2^n - 1, 2^n, 2^n + 1\}$ is presented. By applying the number theoretic properties shown in Chapter 2, the inter-modulus dependency of RNS variable scaling has been eliminated. The derivation steps of the algorithm that lead to the highly simplified architecture are elaborated.

As this is the first ever programmable scaler reported for this RNS, its performance is thoroughly evaluated, analyzed and compared against the hybrid solution (composed of the most efficient residue-to-binary and binary-to-residue converters and binary logarithmic shifter) to substantiate its remarkably lower area, delay and power consumption.

In Chapter 5, a new approach to the design of efficient residue generators for arbitrary moduli is described. A detailed flow chart is given to visually illustrate various design steps in arriving at the proposed residue generators. The block diagram of the proposed design is then provided to show that the disparity due to varying periodicity of different moduli has been eliminated. Likewise, performance evaluation and comparison are presented to prove that the proposed general moduli residue generators outperform the existing state-of-the-art designs in terms of area, delay and power consumption.

Finally, Chapter 6 summarizes the main results and findings in this thesis and presents several feasible ideas that are worthy of exploration in future.

Chapter 2

Background and Literature Review

2.1 Overview of Residue Number System

An integer numeral system is defined as a set of integers along with the arithmetic operations that can be performed on the integers. Any number X can be expressed with a limited number of digits from a finite digit set as follows:

$$X = \sum_{i=0}^{L-1} x_i w_i \quad (2.1)$$

where x_i and w_i are the digit and the weight, respectively at the i th position, and L is the wordlength of the integer X in this representation. As the value of the integer contributed by each digit x_i depends on its position, such a positional numeral representation is called a weighted number system. The weight w_i is the i th power of a base r_i , which is known as radix. The radix r_i is the number of unique digits that can be used to represent x_i . If the same radix r is used throughout the L digit positions of the integer X , then the numeral system is a fixed-radix system; otherwise, it is mixed-radix system. Decimal number system, which is a fixed-radix number system of radix 10, is the most widely used value system by human while binary number system, which is of radix 2, is used in computer or electronic computation. Other commonly used fixed-radix number systems include octal and hexadecimal number systems, which are of radix 8 and 16 respectively. The weighting information of positional number system is particularly useful for efficient hardware implementation of many fundamental operations. For example in binary number system, variable power-of-two scaling can be implemented simply by programmable shifter while overflow and sign detection can be carried out by observing the carry output bit and the most significant bit of the number, respectively.

In weighted number system, the speed of the arithmetic operation is limited by the carry propagation between digits, i.e., each digit of the results depends on all digits of the operands of equal or lower significance. RNS, in contrast, is a non-weighted number system. It breaks the long carry propagation chain into multiple shorter carry propagation chains to speed up the computation of addition, subtraction and multiplication.

RNS is defined by a base consisting of a set of integers $\{m_1, m_2, \dots, m_N\}$. m_i is known as a modulus. Any integer X can be represented in the RNS by N -tuple $(x_1, x_2, \dots, x_N)$, where x_i is the least positive remainder divided by m_i . The residue of $X \bmod m_i$ is commonly represented as

$$x_i = |X|_{m_i} \quad (2.2)$$

The i th residue digit x_i can only take the values of $[0, m_i-1]$. To ensure that the residue representation $(x_1, x_2, \dots, x_N)$ is unique within a given range of integers, the greatest common divisor between any two moduli of the base must be one, i.e., $\text{GCD}(m_i, m_j) = 1 \forall i \neq j$, and the dynamic range of the RNS is given by the product of all moduli, i.e., $M = \prod_{i=1}^N m_i$. An integer X can be calculated from its residue digits $(x_1, x_2, \dots, x_N)$ using the CRT [Moh02], [Omo07], [Sza67] as follows:

$$X = \left| \sum_{i=1}^N M_i |M_i^{-1}|_{m_i} x_i \right|_{m_i} \quad (2.3)$$

where $M_i = \frac{M}{m_i}$ and $|M_i^{-1}|_{m_i}$ is the multiplicative inverse of $|M_i|_{m_i}$. An integer a is said to be the multiplicative inverse of an integer b modulo m if $a \times b = 1 \bmod m$.

The choice of moduli has significant impact on the hardware implementation and hence the performance and complexity of the arithmetic operations in an RNS. In general, the moduli can be categorized into two groups: general and special moduli. General modulus, as the name implies, has no specific form. On the contrary, special moduli refer to those that possess arithmetic friendly modulo operations, such as integers of the 2^n-1 , 2^n or 2^n+1 forms. Special moduli are rich in good number theoretic properties. Various moduli sets based on these three specific forms of special moduli have been suggested in the literature:

$\{2^n - 1, 2^n, 2^n + 1\}$, $\{2^n - 1, 2^n, 2^{n-1} - 1\}$, $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$, $\{2^n - 1, 2^n, 2^n + 1, 2^{n-1} - 1\}$, $\{2^n - 1, 2^n, 2^n + 1, 2^{2n} + 1\}$, $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1, 2^{n-1} - 1\}$, to name a few. The most popular special moduli set is the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The latter moduli set is most balanced and well-established. The wordlength of each modulus is approximately the same and equal to n bits. Many efficient residue-to-binary converters, modulo arithmetic units, and other fundamental operations like sign detection and magnitude comparison have been proposed based on this special moduli set [Abd95], [Bak09], [Cha11], [Lot10], [Moh98], [Moh07a], [Moh07b], [Moh08], [Mur08], [Mur11], [Mur12], [Pie02], [She12], [Ver02], [Ver09], [Ver10]. The three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ has dynamic range of approximately $3n$ bits and the wordlength n is reasonably small for many commonly encountered DSP applications, which explains for its popularity. The first two contributions of this thesis target the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$.

Below are some properties [Moh02], [Omo07], [Sza67] that are used extensively in this thesis for the exposition of the proposed algorithms and architectures.

Property 2.1: $|mX|_m = 0$

Property 2.2: $|X + Y|_m = \||X|_m + |Y|_m|_m$

Property 2.3: $|X \cdot Y|_m = \||X|_m \cdot |Y|_m|_m$

Property 2.4: $|-X|_m = |m - X|_m$

Property 2.5: $|kX|_{km} = k|X|_m$

Property 2.6 [Cao03], [Cao05], [Moh02], [Omo07], [Zim99]: Multiplying an n -bit binary number X by two to the power of i in modulo $2^n - 1$ is equal to shifting X circularly to the left by i bits, denoted by

$$|2^i X|_{2^n - 1} = CLS_n(X, i)$$

Property 2.7 [Cao07], [Moh02], [Omo07], [Zim99]: Let X be an n -bit number. Since $|-X|_{2^n - 1} = |2^n - 1 - X| = \bar{X}$, where $\bar{X}$ denotes the one's complement of X , the following results is a corollary of Property 2.6.

$$|-2^i X|_{2^n - 1} = CLS_n(\bar{X}, i)$$

Property 2.8: Let $X_{n-1:0}$ denotes the least significant n bits of X , then

$$|X|_{2^n} = X_{n-1:0}$$

Property 2.9 [Moh02], [Mur08], [Omo07], [Zim99]: Let X be a single bit binary variable, then

$$|2^{n+i} X|_{2^{n+1}} = |2^i \bar{X} + 2^{n+i}|_{2^{n+1}}$$

2.2 Residue Arithmetic

In weighted number system, the arithmetic involved are saturated, which means that the result of an integer operation is either truncated or rounded to the maximum or minimum value when it exceeds either end of the range. On the other hand, arithmetic operation in RNS is modular and the result wraps around upon reaching the modulus. Such arithmetic operations are widely referred to as modulo arithmetic or residue arithmetic. Let $(x_1, x_2, \dots, x_N)$ and $(y_1, y_2, \dots, y_N)$ be the N -tuple residue digits of X and Y respectively, the residue representation of the result of the operation $Z = X \circ Y$ is given by

$$\begin{aligned} Z &\equiv (z_1, z_2, \dots, z_N) \\ &= (|x_1 \circ y_1|_{m_1}, |x_2 \circ y_2|_{m_2}, \dots, |x_N \circ y_N|_{m_N}) \end{aligned} \quad (2.4)$$

where $(z_1, z_2, \dots, z_N)$ is the N -tuple residue digits of Z , and ' $\circ$ ' denotes the arithmetic operation of addition, subtraction or multiplication. Modulo arithmetic operates on reduced-wordlength operands independently and in parallel in the respective modulus channel as shown in Figure 1.1. By introducing sub-word level parallelism into the operation, modulo arithmetic is faster than the binary counterpart. With the advantage of high-speed modulo arithmetic of special moduli, special moduli set RNS is well received in applications consisting of a series of additions/subtractions and multiplications. Sum-of-products kernels are ubiquitously found in many DSP algorithms, which make the modulo arithmetic units pivotal to the overall performance of RNS-based digital signal processor shown in Figure 1.1. Because of this, a vast amount of research works has been carried out to enhance the performance of the modulo arithmetic units [Mur11], [Mur12], [Ver09], [Ver02], [Mur08], [Pie02], [Hia02], [Efs03], [Efs94].

2.3 Binary-To-RNS Conversion

Prevailing DSP systems are built based on the accustomed binary number system due to its technological maturity and the huge industrial-wide investments in electronic design automation tools and design methodologies that have already been expended over years and will continue to be expended on this legacy number system. RNS-based arithmetic units or application-specific digital signal processor, as embedded hardware accelerators, cannot evade the communication incompatibility problem with the binary number system at their peripheral interfaces. A binary-to-RNS converter, also known as RNS forward converter and residue generators for a given RNS base, is required to convert the input data received from the binary bus interface to the RNS system. It is placed at the front end of the RNS architecture as depicted in Figure 1.1. To compute the residue for each modulus m_i based on (2.2), a residue generator is required for each modulus channel. The number of modulus channels of an RNS is known as its cardinality. For an N cardinality RNS, a total of N residue generators are employed, and they operate in parallel and independently. The type of moduli has direct implication to the implementation complexity of the residue generators. Special moduli of the forms $2^n \pm 1$ and 2^n enable efficient implementation of the residue generators. The latter residue generators can be constructed by using merely several modulo $2^n \pm 1$ adders or at no logic cost by mere hardwiring [Moh02], [Omo07], [Sza67]. However, RNS formed by such arithmetic friendly special moduli has limited cardinality. To increase the dynamic range, the wordlength of one or more modulus (i.e., n) has to increase, leading to degradation of the overall system performance. Thus the gain of parallelism diminishes for very high dynamic range applications such as those in the cryptographic digital signal processor. On the other hand, it is relatively easy to form large dynamic range RNS from general moduli by selecting as many integers as possible without restriction as long as the co-prime requirement is fulfilled. Furthermore, it is found that RNS formed by large number of small modulus is more tolerant to delay variation. Given the large cardinality, it is not only important to reduce the complexity of the residue generators so that they would not become the performance bottleneck of the RNS architecture, but also crucial to ensure the delay of the residue generator is balanced across all modulus channels. Large variability in delay will undermine the benefit of forming RNS with small moduli. Unfortunately, due to limited number

theoretic properties, a unified and systematic design approach to efficient binary-to-residue converter even with moduli of small wordlength for general moduli set is still lacking.

2.4 RNS-To-Binary Conversion

Likewise, an RNS-to-binary converter, which is also commonly known RNS reverse converter, is also required at the back end of the RNS architecture to communicate the result to the binary peripheral devices or binary processing modules in a hybrid binary-RNS system. It receives the residue digits from the final stage of the modulo computation unit and produces the corresponding binary integer.

The RNS-to-binary converter is typically designed based on the CRT stated in (2.3) and Mixed-Radix Conversion (MRC) algorithm whereby the resulting binary integer representation X is obtained from its residue digit representation of $(x_1, x_2, \dots, x_N)$ by

$$X = a_1 + a_2 m_1 + a_3 m_1 m_2 + \dots + a_N \prod_{i=1}^{N-1} m_i \quad (2.5)$$

where a_i is the mixed-radix digit. The a_i s are determined one digit at a time starting from a_1 as shown below:

$$\begin{aligned} a_1 &= x_1 \\ a_2 &= \left\lfloor (x_2 - a_1) \left\lfloor \frac{1}{m_1} \right\rfloor_{m_2} \right\rfloor_{m_2} \\ a_3 &= \left\lfloor \left((x_3 - a_1) \left\lfloor \frac{1}{m_1} \right\rfloor_{m_3} - a_2 \right) \left\lfloor \frac{1}{m_2} \right\rfloor_{m_3} \right\rfloor_{m_3} \\ &\vdots \\ a_N &= \left\lfloor \left(\left(\left((x_n - a_1) \left\lfloor \frac{1}{m_1} \right\rfloor_{m_N} - a_2 \right) \left\lfloor \frac{1}{m_2} \right\rfloor_{m_N} - a_3 \right) \left\lfloor \frac{1}{m_3} \right\rfloor_{m_N} - \dots - a_{N-1} \right) \left\lfloor \frac{1}{m_{N-1}} \right\rfloor_{m_N} \right\rfloor_{m_N} \end{aligned} \quad (2.6)$$

Improved version of CRTs, known as the New-CRT I and the New-CRT II, have been proposed by [Wan00]. Using New-CRT I, the integer number X is related to its residue digits $(x_1, x_2, \dots, x_N)$ by

$$X = x_1 + m_1 \left| k_1(x_2 - x_1) + k_2 m_2(x_3 - x_2) + \dots + k_{n-1} m_2 m_3 \dots m_{N-1}(x_N - x_{N-1}) \right|_{m_2 m_3 \dots m_N} \quad (2.7)$$

where k_i is the multiplicative inverse of $\left| \prod_{j=1}^i m_j \right|_{m_{i+1} m_{i+2} \dots m_N}$. By definition of multiplicative inverse,

$$\begin{aligned} |k_1 m_1|_{m_2 m_3 \dots m_N} &= 1 \\ |k_2 m_1 m_2|_{m_3 m_4 \dots m_N} &= 1 \\ &\vdots \\ |k_{N-1} m_1 m_2 \dots m_{N-1}|_{m_N} &= 1 \end{aligned} \quad (2.8)$$

For New-CRT II, the integer number X is computed from its residue digits $(x_1, x_2, \dots, x_N)$ using the algorithm *translate* as shown below:

Algorithm *translate* $((x_1, x_2, \dots, x_N), X)$

(1) If $N > 2$, let $t = \lfloor N/2 \rfloor$, then

translate $((x_1, x_2, \dots, x_t), N_1), M_1 = m_1 m_2 \dots m_t$

translate $((x_{t+1}, x_{t+2}, \dots, x_N), N_2), M_2 = m_{t+1} m_{t+2} \dots m_N$

findno (N_1, N_2, M_1, M_2, X)

(2) If $N = 2$, then *findno* (x_1, x_2, m_1, m_2, X)

Procedure *findno* (x_1, x_2, m_1, m_2, X)

(1) Find a k_0 such that $k_0 m_2 = 1 \pmod{m_1}$

(2) $X = x_2 + L_2 |k_0(x_1 - x_2)|_{L_2}$

Due to the lack of modularity and parallelism, RNS-to-binary converter usually incurs much larger hardware area and longer delay than a residue generator even for the special moduli set, as can be seen from the above computations. Thus hybrid RNS as shown in Figure 1.1(b) is not an attractive solution to resolve other inter-modulo operation problems in RNS. Fortunately, in most cases, the cost and performance penalty of the RNS-to-binary converter is greatly amortized over a large number of multiply-and-add computations directly in residue domain. The overhead of RNS-to-binary conversion, which is required only once at the final stage of a pure RNS IPSP system, represents only a very small fraction of the overall system cost.

2.5 Review of RNS Scalers

The earliest research on RNS scaling problem dated back to the sixties whence scaling an integer by a product of any subset of the moduli in RNS can be performed in N clock cycles [Sza67]. Since then, many new techniques have been proposed in the literature, and they can be categorized into three main groups: scaling by subset of the moduli set [Bar95], [Bur03], [Cha11], [Gar98], [Gar99], [Phi03], scaling by a factor that is co-prime with the moduli set [Ulm98], [Kon09], and scaling by the power-of-two [Car05b], [Cha11], [Ma10], [Mey03]. This section reviews some recent developments on RNS scaling algorithms that have an impact on hardware implementation.

RNS scaling problem is typically formulated based on modulo reduction of the least integer function of division or by CRT. The former problem is generally solved by the computationally intensive base extension methods [Bar95], [Das08], [Gar99], [Ma10] [Ye08], [Kon09], and the outputs are free from scaling error. On the other hand, CRT-based approaches [Gri88], [Gri89], [She89], [Ulm93], [Ulm98] usually produce scaled integer with some fractional error. The error could escalate and might not be tolerable for critical applications. All the aforementioned RNS scaling algorithms, except [Das08], [Ma10], [Ye08], are implemented using ROMs.

Shenoy and Kumaresan [She89] proposed a scaling technique based on the CRT. In their work, a redundant residue digit is introduced to enable the parallel computation of the scaled residues, and a novel decomposition of CRT is applied to reduce the maximum scaling error to unity. In [Gri88], Griffin *et al.* first used the least integer function $\lfloor \cdot \rfloor$ in CRT to produce the scaled output with error bounded by the number of moduli L used in the RNS (hence, it is known as L -CRT). Later, a generalization of L -CRT, called $L(\varepsilon + \delta)$ -CRT [Gri89] is proposed to provide the flexibility in choosing the scaling factor but the new reduced system moduli comes at a cost of potentially large errors. The catastrophic error band may be avoided with carefully selected reduced moduli for a pre-specified scaling factor. Another scaling algorithm that utilizes CRT was put forward by Ulman *et al.* [Ulm93]. The latter scaling method could be considered as an improved version of the design in [Sza67], as it

eliminates the use of redundant modulus without imposing any restrictions on the system moduli and on the scaling factor. Besides, the scaling error of this algorithm is not greater than 1.5. The aforementioned scaling algorithms use CRT in such a way that the residue representation of an integer in RNS is partially converted to its binary representation before producing the scaled output. Furthermore, they are ROM-based design which cannot be easily pipelined for high-throughput applications.

More recent works on scaling an integer X by a factor k for the moduli set $\{m_1, m_2, \dots, m_N\}$ is based on the scaling operation defined in [Gar99]. It states that if k is the scaling constant and Y is the result of scaling X by k , then $X = kY + |X|_k$, where $|X|_k$ denotes the remainder of X divided by k . Thus,

$$|Y|_{m_i} = \left\lfloor \frac{X}{k} \right\rfloor_{m_i} = \left\lfloor \frac{X - |X|_k}{k} \right\rfloor_{m_i} \quad (2.9)$$

Barsi and Pinotti [Bar95] derived a scaling algorithm by applying the base extension to (2.9) without approximation. The latter work has once again manipulated the CRT so that their design can perform both base extension and exact scaling without any system redundancy. Using almost similar scaling algorithm, Garcia and Lloris [Gar99] proposed a different look-up scheme, which leads to a faster and simpler implementation of the RNS scaling operation than that of [Bar95]. The drawback is its restriction on the number of moduli in the system – significantly large amount of memory is required for a large number of moduli. To circumvent large memory requirement when the number of moduli increases, Kong and Phillips [Kon09] proposed to avoid using the scaling constant that is equal to the product of a subset of moduli for (2.9). In their approach, scaling constant is chosen to be any integer coprime with the moduli set. Coupled with the efficient base extension algorithm proposed in [Bar95], this scaling scheme has better performance than the latter two scaling techniques which utilize the same problem formulation method. However, these designs also use memory for implementation. All these memory-based designs suffer from poor pipelinability and dramatic increase in hardware cost with the number of system moduli. Meyer-Base and Stouraitis [Mey03] came out with a scaling algorithm for small word length applications which does not rely on a conversion back to a weighted number system. Instead, it extended the *Division Remainder Zero Theorem* of [Sza67] to perform scaling directly on the residue digits.

The only valid adder-based RNS scaler design techniques were proposed in [Ye08] and [Ma10]. It is formulated based on (2.9). The scaler of [Ye08] is dedicated to the three moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ with scaling constant 2^n . Ma *et al.* [Ma10] extended it to general moduli sets with the proviso that the moduli are coprime with the scaling constant. The generalization has restricted its hardware simplification, leading to increased hardware cost and reduced performance over those of [Ye08]. All the above algorithms have their scaling factor predetermined at design time. The only variable scaling solution is proposed in [Car05b]. It is a workaround by cascading r (r is the maximum number of shifts) scaling-by-two blocks developed in [Mey03] to perform the programmable power-of-two scaling operation in RNS. Each of the basic scaling-by-two blocks consists of an RNS sign detector, an RNS parity checker, N (N is the number of moduli) modular adders and modular multipliers, and some 2-to-1 multiplexers. Besides, additional hardware is required for the scaling of the redundant modulus and error correction. It turns out that for the same dynamic range, a hybrid solution using efficient residue-to-binary [Wan02] and binary-to-residue [She04] converters for the special moduli sets like $\{2^n - 1, 2^n, 2^n + 1\}$ with a programmable binary shifter could do better than [Car05b]. Unfortunately, the design of [Car05b] is not applicable to such modulo-arithmetic friendly moduli set because it requires the existence of the multiplicative inverse $|2^{-1}|_{m_i}$ for all of its moduli m_i , which means that no even modulus is allowed in the moduli set. In what follows, the latest five major RNS scalers, [Car05b], [Gar99], [Ma10], [Ye08], [Kon09], will be further elaborated.

2.5.1 Two-Lookup Cycle Scaling Technique [Gar99]

The main contribution of this work is the proposal of a different lookup scheme for the RNS scaling operation. The RNS scaling algorithm used is derived based on the definition of scaling in (2.9). Given a set of moduli $\{m_1, m_2, \dots, m_N\}$, the corresponding residue digits of the scaled integer Y is determined by:

$$\begin{aligned}
y_i = Y \bmod m_i &= \left\| X - \left\| X \right\|_k \right\|_{m_i} \cdot \left\| k^{-1} \right\|_{m_i} \right\|_{m_i} \\
&= \left\| x_i - \left\| X \right\|_k \right\|_{m_i} \cdot \left\| k^{-1} \right\|_{m_i} \right\|_{m_i}
\end{aligned} \tag{2.10}$$

In (2.10), k can be chosen to be equal to the product of a subset of moduli

$$k = \prod_{i=1}^S m_i \tag{2.11}$$

where $S < N$. However, the multiplicative inverse of $\left\| k \right\|_{m_i}$ does not exist for m_i with $1 \leq i \leq S$.

Under the above premise, it is clear that

$$0 \leq Y < \prod_{i=S+1}^N m_i \tag{2.12}$$

In other words, as the RNS scaling operation is carried out with respect to the moduli set $\{m_{S+1}, m_{S+2}, \dots, m_N\}$, it can only obtain the residue digits $(y_{S+1}, y_{S+2}, \dots, y_N)$. As a consequence, residue digits $(y_1, y_2, \dots, y_S)$ must be generated after the calculation of residue digits $(y_{S+1}, y_{S+2}, \dots, y_N)$ using either the base extension or other iterative algorithms. The computation of $\left\| X \right\|_k \right\|_{m_i}$ is one of the major issues when using (2.10). If $k = \min\{m_i\}$ (e.g., m_1), then

$$\left\| \left\| X \right\|_{m_1} \right\|_{m_i} = \left\| X \right\|_{m_i} = x_i \tag{2.13}$$

for $i = S+1, S+2, \dots, N$. Otherwise, $\left\| X \right\|_k \right\|_{m_i}$ must be obtained using the base extension method based on the input residue digits $(x_1, x_2, \dots, x_S)$. Nevertheless, Look-Up Tables (LUTs) can be used to calculate $(y_{S+1}, y_{S+2}, \dots, y_N)$ and generate $(y_1, y_2, \dots, y_S)$.

From the above discussion, it is clear that the computation of y_i for $S+1 \leq i \leq N$ requires the values of x_i and $(x_1, x_2, \dots, x_S)$. Therefore, the computation of y_i for $S+1 \leq i \leq N$ can be implemented by looking up for $(x_1, x_2, \dots, x_S, x_i)$ which requires $N-S$ tables of $S+1$ inputs. Furthermore, the generation of $(y_1, y_2, \dots, y_S)$ depends solely on the values of $(y_{S+1}, y_{S+2}, \dots, y_N)$, and can therefore be implemented using a direct look-up for the base

extension. S tables of $N-S$ inputs are required for this purpose. The block diagram of the aforementioned look-up scheme for the RNS scaling operation is illustrated in Figure 2.1.

Figure 2.1: Two look-up cycle RNS scaler

Figure 2.2: Two look-up cycle RNS scaler for RNS $\{29, 31, 32\}$ and $k = 29$

This scaling technique is only practical for small N and S because look-up tables with more than two inputs are expensive in terms of silicon area. Nonetheless, there are certain sets of moduli in which the aforementioned scheme is feasible. One of which is the three-moduli set. A practical example is the three-moduli set $\{29, 31, 32\}$ which has a dynamic range of 14 bits. If the scaling constant k is chosen to be any one of the moduli (e.g., 29), the RNS scaling operation can be carried out by only three two-input LUTs, two for the calculation of y_2 and y_3 , and the other for the generation of y_1 from y_2 and y_3 (Figure 2.2).

2.5.2 Fast Scaling Scheme [Kon09]

The fast RNS scaler [Kon09] is also formulated based on the modulo reduction of the least integer of scaling in (2.10). The pitfall of the dependency of the computation of y_i for $1 \leq i \leq S$ on y_i for $S+1 \leq i \leq N$ is overcome by selecting the scaling constant k from only the coprime positive integers to the moduli of RNS. This constraint ensures that $|k^{-1}|_{m_i}$ exists for all values of i . Once k is selected, $|k^{-1}|_{m_i}$ can be pre-computed and the only unknown variable in (2.10) is $|X|_k$.

Base extension is applied in the first step of this scaling scheme to compute $|X|_k$ from $(x_1, x_2, \dots, x_N)$. Then, modulo subtraction and multiplication are applied in each residue channel to compute y_i in parallel. The architecture of the proposed scaling scheme is shown in Figure 2.3.

Figure 2.3: Architecture of the fast RNS scaling scheme

It can be observed that each of the LUTs that produces a final residue digit requires merely two inputs. The sizes of the two inputs correspond to the size of the modulus and the size of the scaling constant, respectively. While the size of the modulus is always considerably small (in an RNS architecture), the scaling constant is chosen such that the sizes of the required LUTs are reasonably small. Furthermore, efficient base extension algorithm, which has been

proposed and used in the design of RNS scaler in [Bar95], can be used to generate $|X|_k$. As a result, this scaling scheme exhibits better VLSI performance in terms of total delay and hardware cost in comparison with the two-lookup cycle scaling technique in [Gar99].

2.5.3 Full-Adder Based Exact Scaling [Das08]

The first and only full adder (FA) based exact scaling algorithm that operates directly on the residue digits was proposed by Dasygenis *et al.* [Das08]. It uses the axiom, $|X - |X|_k|_{m_i} =$

$$|x_i - ||X|_k|_{m_i}|_{m_i} \text{ to compute the scaled output, } y_i = ||x_i - ||X|_k|_{m_i}|_{m_i} \cdot |k^{-1}|_{m_i}|_{m_i} \text{ over each modulus}$$

independently, where $x_i = |X|_{m_i}$ is a residue of the RNS defined by the moduli set $\{m_1, m_2, \dots, m_N\}$. The computation is divided into several major stages as shown in Figure 2.4.

Each stage can be implemented with only FAs, which makes the hardware cost remarkably lower and throughput rate significantly higher than all the previous designs. Unfortunately, this algorithm has ignored the fact that $||X|_k|_{m_i}$ is an inter-modulo operation and it cannot be

simply resolved by treating X as an overflowed digit or a residue over each modulus independently. The authors of [Das08] also made a catastrophic assumption that the scaling constant, k can be chosen to be a product of various moduli, i.e., $k = \sum_{i=1}^S m_i$ with $S < N$, but it

has been proven that the multiplicative inverse of k for m_i ($1 \leq i \leq S$) does not exist [Gar99],

[Bar95]. Consequently, the output, $(y_1, y_2, \dots, y_N)$ calculated by the algorithm of [Das08] is not a valid RNS representation of scaling X by k . Using the first example from [Das08], this

error is illustrated by the numerical calculation in Table 2.1. The actual residue digits of the scaled value (254, 204, 154) are different from those calculated (4, 3, 3) using the scaling method in [Das08]. In fact, the condition $(k, m_i) = 1$ for the existence of $|k^{-1}|_{m_i}$ is not met

because of $(k, m_1) = (255, 255) = 255$. Besides, it may not be possible to access to the already overflowed digits because the modulo operators in RNS are usually designed such that the modulo reductions are performed to keep the final results and even their intermediate computations within the legitimate ranges of the respective moduli.

In view of the above, the only valid adder-based RNS scaler design technique that utilizes (2.9) was proposed in [Ye08] and [Ma10] which will be discussed in the subsequent sections.

Figure 2.4: Flowchart of RNS scaler design in [Das08]

Table 2.1: Computation of actual residue digits of scaled number

Modulus, m_i	255	256	257
Dynamic range, M	16776960		
M_i	65792	65535	65280
M_i^{-1}	128	255	129
$M_i \cdot M_i^{-1}$	8421376	16711425	8421120
overflowed $x_i = x_i'$	1100	900	800
$x_i = x_i' _{m_i}$	80	132	29
$x_i \cdot M_i \cdot M_i^{-1}$	673710080	2205908100	244212480
Integer X	3316100		
Scaler k	255		
$Y = \lfloor X/k \rfloor$	13004		
y_i	254	204	154

2.5.4 RNS Scaling by 2^n for Moduli Set $\{2^n-1, 2^n, 2^n+1\}$ [Ye08]

This work proposes an algorithm for scaling an RNS representation by 2^n for the special moduli set $\{2^n-1, 2^n, 2^n+1\}$. The scaling problem is formulated based on (2.9), which is the same as the two designs discussed in Section 2.5.1 and 2.5.2. With properly selected scaling constant and special moduli set with good number theoretic properties, this RNS scaler can be implemented by modulo adders and binary adders which are combinational logic circuits. The derivation of this RNS scaling algorithm is detailed as follows.

Without loss of generality, let $m_1 = 2^n - 1$, $m_2 = 2^n$, $m_3 = 2^n + 1$ and $k = 2^n$. Equation (2.9) can be re-expressed as:

$$y_i = \left| x_i - |X|_{2^n} \right|_{m_i} \cdot \left| (2^n)^{-1} \right|_{m_i} \quad (2.14)$$

where $i = 1, 2$ and 3 .

As emphasized in Section 2.5.1 and 2.5.2 that if $k = \prod_{i=1}^S m_i$ for $S < N$, then $|k^{-1}|_{m_i}$ does not exist for $1 \leq i \leq S$, and so do y_i for $1 \leq i \leq S$. In [Ye08], $k = 2^n$. Therefore, $|(2^n)^{-1}|_{2^n}$ does not exist and hence y_2 cannot be obtained directly using (2.14) while y_1 and y_3 can be easily computed using (2.14), where $|X|_{2^n} = x_2$, $|(2^n)^{-1}|_{2^n+1} = 2^n$ and $|(2^n)^{-1}|_{2^n-1} = 1$. Hence, the expressions for y_1 and y_3 are

$$y_1 = \left| (x_1 - x_2) \cdot 1 \right|_{2^n-1} = |x_1 - x_2|_{2^n-1} \quad (2.15)$$

$$y_3 = \left| (x_3 - x_2) \cdot 2^n \right|_{2^n+1} = |x_2 - x_3|_{2^n+1} \quad (2.16)$$

To obtain y_2 , information regarding the integer Y is required and it can be obtained using either CRT or MRC. In [Ye08], New CRT-I is applied to produce the integer Y from the known residue digits (y_1, y_3)

$$Y = y_3 + |k_1 (y_1 - y_3)|_{m_1} m_3 \quad (2.17)$$

where $k_1 = |m_3^{-1}|_{m_1} = |(2^n + 1)^{-1}|_{2^n - 1} = 2^{n-1}$. After that, modulus 2^n operation on the integer Y is carried out to generate the final residue digit y_2 ,

$$y_2 = |Y|_{m_2} = |y_3 + |k_1(y_1 - y_3)|_{m_1} m_3|_{m_2} \quad (2.18)$$

Replacing (2.18) with all the pre-determined or pre-computed constants, we have

$$\begin{aligned} y_2 &= |y_3 + |2^{n-1}(y_1 - y_3)|_{2^n - 1} (2^n + 1)|_{2^n} \\ &= |y_3 + |2^{n-1}(y_1 - y_3)|_{2^n - 1}|_{2^n} \end{aligned} \quad (2.19)$$

According to *Property 2.6*, $|2^{n-1}a|_{2^n - 1}$ is merely a circular left shift of $n-1$ bits of a . Therefore, (2.19) can be implemented by one modulo $2^n - 1$ adder and one n -bit binary adder. It should be noted that $y_3 \leq 2^n$ (i.e., y_3 is $(n+1)$ -bit long). From [Tom08],

$$|2^{n-1}y_3|_{2^n - 1} = |2^{n-1}|_{2^n - 1} |y_3|_{2^n - 1} \quad (2.20)$$

and,

$$|y_3|_{2^n - 1} = |y_{3,n-1}y_{3,n-2} \dots y_{3,0} + y_{3,n}|_{2^n - 1} \quad (2.21)$$

When $y_{3,n} = 1$,

$$|y_3|_{2^n - 1} = |00 \dots 0 + y_{3,n}|_{2^n - 1} = y_{3,n} \quad (2.22)$$

When $y_{3,n} = 0$,

$$|y_3|_{2^n - 1} = |y_{3,n-1}y_{3,n-2} \dots y_{3,0} + 0|_{2^n - 1} = y_{3,n-1}y_{3,n-2} \dots y_{3,0} \quad (2.23)$$

Hence,

$$|y_3|_{2^n - 1} = y_{3,n-1}y_{3,n-2} \dots (y_{3,0} + y_{3,n}) \quad (2.24)$$

Equations (2.15), (2.16) and (2.19) clearly reveal that this RNS scaler can be implemented efficiently with only modulo and binary adders, which are composed of combinational logic.

The RNS scalars discussed so far focus on unsigned numbers. Ye *et al.* [Ye08] also proposes a solution to handle sign numbers. The definition of signed numbers in RNS is similar to that in two's complement number system. Let the dynamic range of an RNS be M and the signed number be denoted as X' . If $X < M/2$, X' is positive; if $X \geq M/2$, X' is negative and

hence $-X' = M - X$ (i.e. absolute value). Therefore, if X' is negative, scaling should be performed first on its absolute value, $\langle X' \rangle$ ¹

$$\begin{aligned} Y_{absolute} &= (\langle X' \rangle - |X|_k) / k \\ &= (M - X - |M - X|_k) / k \\ &= (M - X - |M|_k + |X|_k) / k \end{aligned} \quad (2.25)$$

After that, the mapping of scaling result, $Y_{absolute}$ to the appropriate range that represents the negative number is carried out by

$$\begin{aligned} Y &= M - Y_{absolute} \\ &= ((k-1)M + X + |M|_k - |X|_k) / k \end{aligned} \quad (2.26)$$

Given that $k = m_2$ and $M = m_1 m_2 m_3$,

$$Y = (m_2 - 1)m_1 m_3 + \frac{X - |X|_{m_2}}{m_2} \quad (2.27)$$

Let $t = (m_2 - 1)m_1 m_3$ and $d = (X - |X|_{m_2}) / m_2$, it is obvious that d is the scaling result for a positive number. In other words, t is needed as a correction constant when dealing with negative numbers. Therefore, by including the correction constant t , the residue digit for a negative number is

$$y'_i = |y_i + t|_{m_i} = |y_i + |t|_{m_i}|_{m_i} \quad (2.28)$$

Since $t = (m_2 - 1)m_1 m_3$, it can be easily computed that $|t|_{m_1} = 0$, $|t|_{m_3} = 0$ and $|t|_{m_2} = 1$. As a result, the final scaled residue digits of a negative number are given by

$$y'_1 = |x_1 - x_2|_{2^{n-1}} \quad (2.29)$$

$$y'_2 = |y_3 + |2^{n-1}(y_1 - y_3)|_{2^{n-1}} + 1|_{2^n} \quad (2.30)$$

$$y'_3 = |x_2 - x_3|_{2^{n+1}} \quad (2.31)$$

The architecture of this RNS scaler is depicted in Figure 2.5. From Figure 2.5, it can be deduced that this RNS scaler is superior than the ROM-based RNS scalars due to the reduced hardware requirement. However, the drawback of this architecture is its longer delay. Moreover, a non-trivial RNS sign detection circuit has to be included in this architecture to

¹ $\langle \cdot \rangle$ is used to indicate absolute value instead of $|| \cdot ||$ because the latter has been widely and consistently used as modular operation.

determine the sign of the integer number from the given residue digits $(x_1, x_2, \dots, x_N)$ before the correct output is selected.

Figure 2.5: 2^n RNS scaler for moduli set $\{2^n-1, 2^n, 2^n+1\}$

2.5.5 A 2^n RNS Scaling Scheme [Ma10]

In general, this RNS scaling algorithm is similar to that of [Ye08]. First, the RNS scaling problem is formulated based on the modulo of the least integer of scaling. Second, signed numbers are considered using the same approach. Third, the scaling constant is 2^n . Lastly, the implementation is based on combinational logic. The only difference is that the design in [Ye08] is dedicated for the special moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ whereas this design has been extended to general moduli set $\{m_1, m_2, \dots, m_N\}$ with the proviso that the moduli are co-prime with the scaling constant 2^n . Since most of the derivation steps are similar, they will not be repeated here.

From (2.14) and (2.26), for scaling constant $k = 2^n$, the residues of the scaling results for positive and negative integers are given by:

$$y_i = \left\| x_i - |X|_{2^n} \right\|_{m_i} \left\| (2^n)^{-1} \right\|_{m_i} \Big|_{m_i}, \quad X' \geq 0 \quad (2.32)$$

$$y_i = \left\| x_i - |X|_{2^n} + |M|_{2^n} \right\|_{m_i} \left\| (2^n)^{-1} \right\|_{m_i} \Big|_{m_i}, \quad X' \geq 0 \quad (2.33)$$

where $i = 1, 2, \dots, N$. By comparing (2.32) with (2.33), $|M|_{2^n}$ is a correction constant for negative number. Due to the choice of moduli set in this scaling algorithm, the scaling constant may not be one of the moduli. Thus, the RNS scaling algorithm cannot take advantage of the same simplifications as in the design of [Ye08] described in Section 2.5.4. However, the dynamic range of this scaling algorithm can be much larger than that of the design in [Ye08]. The architecture of this RNS scaler is illustrated in Figure 2.6. RNS sign detection module is required to determine the sign of the integer while the base extension module is required to compute $|X|_{2^n}$.

Figure 2.6: Architecture of parallel 2^n RNS scaler

2.5.6 Programmable Power-Of-Two Scaler [Car05b]

RNS scaler designs discussed thus far have their scaling factor fixed during design stage. In some applications, variable scaling factor is preferred whereby the scaling factor is determined during runtime. [Car05b] presents an architecture that is able to scale an integer

in RNS with programmable power-of-two scaling factor. It is constructed by the scaling-by-two blocks of [Mey03] developed based on the *Division Remainder Zero Theorem*. The latter theorem as well as the architecture of the RNS programmable power-of-two scaler are briefly discussed below.

Given the N -tuple of residue digits $(x_1, x_2, \dots, x_N)$ of an integer X in RNS $\{m_1, m_2, \dots, m_N\}$ and a scaling factor of 2, if X is divisible by 2 (which means that X is an even number), then

$$\begin{aligned} \lfloor X/2 \rfloor_M &= \lfloor X \cdot 2^{-1} \rfloor_M \\ &\equiv \left(\lfloor x_1 \cdot 2^{-1} \rfloor_{m_1}, \lfloor x_2 \cdot 2^{-1} \rfloor_{m_2}, \dots, \lfloor x_N \cdot 2^{-1} \rfloor_{m_N} \right) \end{aligned} \quad (2.34)$$

where $2^{-1}_{m_i}$ is the multiplicative inverse of 2_{m_i} . The multiplicative inverse $2^{-1}_{m_i}$ exists only if $\text{GCD}(2, m_i) = 1 \ \forall i \in \{1, 2, \dots, N\}$. Due to this reason, the moduli set must be odd numbers.

In the case when X is not divisible by 2 (which means that X is an odd number), a value of one has to be added before scaling by 2, i.e.,

$$\begin{aligned} \lfloor (X+1)/2 \rfloor_M &= \lfloor (X+1) \cdot 2^{-1} \rfloor_M \\ &\equiv \left(\lfloor (x_1+1) \cdot 2^{-1} \rfloor_{m_1}, \lfloor (x_2+1) \cdot 2^{-1} \rfloor_{m_2}, \dots, \lfloor (x_N+1) \cdot 2^{-1} \rfloor_{m_N} \right) \end{aligned} \quad (2.35)$$

To determine if X is an even or odd number, a parity checker is needed. Parity checking is done by evaluating the value of $|X|_2$. If $|X|_2 = 1$, X is an odd number, otherwise X is an even number. This work employs base extension method to compute $|X|_2$. Based on (2.34) and (2.35), a scaling-by-two block is consisting of N modulo adders, 2-to-1 MUX and modulo multipliers, and an RNS parity checker. RNS programmable power-of-two scaler proposed by Cardarilli *et al.* [Car05b] is constructed by cascading as many scaling-by-two block as needed, with MUXes included in between the scaling-by-two blocks and for input routing.

Some modifications on the scaling-by-two block are also proposed by Cardarilli *et al.* [Car05b] to take care of the signed numbers. Since X is represented as $M - X$ and the dynamic range M is odd (as a result of the product of odd moduli), it follows that the even negative numbers are mapped through $M - X$ to odd positive numbers, and vice versa.

Therefore, for scaling signed numbers, additional sign detection block and *XOR* gate are needed. The scaling-by-two block for signed numbers is shown in Figure 2.7.

Figure 2.7: Block diagram of the scaling-by-two block

An overview of the above RNS scaling algorithms is summarized in Table 2.2.

Table 2.2: Overview of RNS scalars

Design	Scaling By	Type	Targeted Moduli
[Gar99]	subset of the moduli set	fixed	general
[Kon09]	factor coprime with the moduli set	fixed	general
[Das08]	subset of the moduli set	fixed	general
[Ye08]	2^n	fixed	special
[Ma10]	2^n	fixed	general, odd
[Car05b]	power of two	programmable	general, odd

2.6 Review of Binary-To-RNS Converters

Binary-to-RNS converters can be categorized into two main groups based on the type of the moduli: general moduli [Ali84], [Cap88], [Moh94], [Moh99], [Pie94], [Pre02], [Pre06], [Sto94], and special moduli of the forms, $2^n \pm 1$ and 2^n [Bi88], [Moh98], [Omo07], [Pou94], [She04]. In general, binary-to-RNS converters based on special moduli are much simpler than that based on the general moduli for the same word length of modulus and integer. However, due to the larger dynamic range and higher delay variability tolerance as discussed in Section

2.3, the use of general moduli of small modulus wordlength has its own merits, if the corresponding binary-to-RNS converters can be made more efficient. Alia and Martinelli [Ali84] proposed the general moduli binary-to-RNS converters based on the power of 2 mod m . Taking the modulus operation on a L -bit integer $X = \sum_{i=0}^{L-1} x_i 2^i$ yields

$$|X|_m = \left| \sum_{i=0}^{L-1} x_i 2^i \right|_m \quad (2.36)$$

Using *Property 2.2* and *Property 2.3*, (2.36) can be written as

$$|X|_m = \left| \sum_{i=0}^{L-1} x_i |2^i|_m \right|_m \quad (2.37)$$

The residue computation based on (2.37) employs $L/2$ Processing Elements (PEs) connected together through a data bus. Each PE performs modulo m addition using the binary addition, subtraction and sign detection operations. Two $\lceil \log_2 m \rceil$ -bit registers which are serially loaded with the values of $|2^i|_m$ and $|2^{i+1}|_m$ are contained in each PE. The final residue $|X|_m$ is obtained after $\log_2 L$ clock cycles. Capocelli and Giancarlo [Cap88] suggested storing the residue corresponding to the first bit in a group of bits, doubling this residue mod m and evaluating the residue of the next power of 2 in that group. With that, only $t = L / \log_2 L$ PEs are used, and the conversion time is faster than [Ali84].

Due to the cyclic periodicity p of $2^i \bmod m$, the integers 2^i and 2^{i+kp} for any integer k have the same residue modulo m [Moh94] (The periodicity p for various moduli m_i is given in Table 2.2). The binary-to-residue converter proposed in [Moh94] first partitions the L -bit integer X into p -bit words $x_{p-1} \dots x_1 x_0, x_{2p-1} \dots x_{p+1} x_p, x_{3p-1} \dots x_{2p+1} x_{2p}$, etc., which are then added by a tree of p -bit two-operand adders to yield a $(p+\log_2 p)$ -bit intermediate result. The intermediate result is reduced to p bits by adding the carries back to the result. In [Moh99], an improved p -bit carry-save addition (CSA) followed by carry-propagation with end-around carry additions are suggested to add the p -bit words. In the second stage of [Moh94] and [Moh99], the residue is computed by the techniques proposed in either [Ali84] or [Cap88]. The first stage of the residue generator of [Pie94] has the same CSA tree and CPA with EAC as that of [Moh99] and the second stage is realized by LUT or multi-operand modulo adder (MOMA).

Table 2.3: Periodicity p for various moduli m_i

m_i	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43
p	2	4	3	6	10	12	4	8	18	6	11	20	18	28	5	10	12	36	12	20	14

m_i	45	47	49	51	53	55	57	59	61	63	65	67	69	71	73	75	77	79	81	83	85
p	12	23	21	8	52	20	13	45	60	6	12	66	22	35	9	20	29	39	54	57	8

m_i	87	89	91	93	95	97	99	101	103	105	107	109	111	113	115	117	119	121	123	125	127
p	12	11	12	10	36	48	25	100	51	12	100	36	36	28	44	12	24	117	20	100	7

In [Sto94], the L -bit binary number is partitioned into c words of q bits each, i.e.,

$$X = \sum_{j=1}^c k_j \quad (2.38)$$

where

$$k_j = \sum_{i=(j-1)q}^{jq-1} 2^i x_i \quad (2.39)$$

where q can be chosen such that L is a multiple of q , and $c = L/q$. The residue $|X|_m$ is given by

$$|X|_m = \left| \sum_{j=1}^c k_j \right|_m \quad (2.40)$$

Using *Property 2.2*, (2.40) can be written as

$$|X|_m = \left| \sum_{j=1}^c |k_j|_m \right|_m \quad (2.41)$$

where $|k_j|_m$ is known as the partition residue. The partition residues are obtained by looking up the LUTs using the q -bit vectors. The partition residues are then added up by modulo m adders in a tree structure to produce the final residue $|X|_m$. The architecture of the LUT-based residue generator is given in Figure 2.8. c LUTs of size $2^q \times \lceil \log_2 m \rceil$ are used to compute the residues $|k_j|_m$. They are then summed by $c-1$ modular adders in $\lceil \log_2 c \rceil$ stages to produce

$$|X|_m = \left| \sum_{j=1}^c |k_j|_m \right|_m.$$

More recent binary-to-residue converter for an arbitrary modulus m [Pre02] computes the residues for every weighted bit of X by $2^i \bmod m$ and adds them using a modular adder tree. This converter has been improved recently by incorporating the property of cyclic periodicity [Pre06]. The first stage is the same as that in [Moh99], i.e., the L -bit binary input is partitioned into p -bit words which are then summed using CSAs and CPAs to produce an intermediate result of p -bit wide. In the second stage, $\lceil \log_2 p \rceil$ -stage of modulo adder tree is used to add the p -bit intermediate result to produce the final residue $|X|_m$. The number of modulo adders needed is $p-1$, which varies for different moduli of different periodicities. The corresponding architecture is shown in Figure 2.9. For large p (equal to L in some cases), the modular adder tree is very large, occupying substantially large hardware area. Not only that, the delay of this residue generator is higher than those in other modulus channels, and constitutes the critical path delay. The large disparity of performance of different moduli defeats the purpose of forming RNS with large number of small arbitrary moduli.

Figure 2.8: LUT-based residue generator

Figure 2.9: Block diagram of residue generator in [Pre06]

Chapter 3

Proposed Fixed Scaler for The Three-Moduli Set $\{2^n-1, 2^n, 2^n+1\}$ RNS

3.1 Introduction

The usefulness of RNS has been demonstrated especially in the design of IPSP-like architectures [Ber07], [Car00], [Con04], [Nan01], [Re01], as discussed in the introduction chapter. The most recent study [Ber07] based on the two's complement and the mixed RNS-binary architecture implementations of an adaptive channel equalization filter with a large number of taps for wireless communications shows that the latter offers remarkable savings in area and power consumption without any degradation in performance. The reason for using a hybrid-RNS in [Ber07] instead of a true RNS system is to avoid the use of complex and expensive scaling circuits.

Reverse conversion of an integer from its residue number representation [Cao03], [Cao05], [Cao07], [Sza67], [Vin94], [Wan02], [Wan03] and scaling of an integer number by a known constant [Bar95], [Das08], [Gar99], [Gri88], [Gri89], [Ma10], [Mey03], [She89], [Ulm93], [Ulm98], [Ye08], [Kon09] are two of the most important problems in RNS. Scaling is of particular significance for implementing IPSP and iterative digital processing systems dominated by a large number of additions and multiplications because it ensures the computed results of the preceding stages do not exceed the dynamic range of the system. Due to the inefficient inter-modular operations, implementing scaling operation in RNS domain entices considerably long delay and high hardware complexity, as in the reverse conversion. Unlike the reverse conversion, which is required only when the final result is needed to be communicated with a normal binary number system, scaling operations are executed repeatedly in IPSP-like architectures to avoid the severe consequence of overflow in modulo

arithmetic. In [Ber07], adaptation is carried out in the binary domain to avoid the speed bottleneck of RNS scaling, but this is viable provided that the channel variations in time are slower with respect to the data rate.

In the absence of an efficient RNS scaler, a binary scaler may be used in a hybrid RNS but it must be preceded by an expensive reverse converter, and followed by additional stage of forward converter, as shown in Figure 1.1(b). It is therefore imperative to have a high-speed scaler that operates directly in the RNS domain with lower complexity than an RNS-to-binary converter. Given that scaling in RNS is an inherently inter-modulo operation whereby knowledge of all of the residue digits is essential to produce the correct final result [Bar95], [Gri89], such a problem is better tackled with a moduli set that possesses good number theoretic properties.

This chapter presents a new and simple adder-based unsigned RNS scaler for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The choice of this RNS for overcoming the scaling problem is well justified by the relative simplicity and efficiency in implementing modulo addition, multiplication and shifter circuits for these three moduli proven by the humongous amount of research publications [Bak09], [Efs94], [Efs03], [Lin07], [Ver02], [Ver09], [Ver10], [Men06], [Mur08]. The conversion from the normal binary representation to residues for moduli of types 2^n and $2^n \pm 1$ is straightforward and the reverse conversion problem for this special three-moduli set has been well solved [Vin94], [Wan02]. Most importantly, the inputs to the proposed RNS scaler are in residue form, and the scaled output can be produced efficiently in both normal binary and residue forms. This has an additional advantage of eliminating the reverse converter in the last stage of the architecture shown in Figure 1.1(a).

The chapter is organized as follows: The new RNS scaling algorithm is derived together with its error analysis in Section 3.2. The architecture of the proposed scaler is presented in Section 3.3. Section 3.4 analyzes the area and time complexity of its implementation, and compares the synthesis results with other scaler designs. The chapter is concluded in Section 3.5.

3.2 Proposed Scaling Algorithm

3.2.1 Preliminaries

In RNS, an integer X is represented by an N -tuple $(x_1, x_2, \dots, x_N)$ with respect to a set of pairwise relatively prime numbers $\{m_1, m_2, \dots, m_N\}$, where $x_i = |X|_{m_i}$, $i = 1, 2, \dots, N$ and $|X|_{m_i}$ is defined as $X \bmod m_i$. The dynamic range of the selected moduli set $\{m_1, m_2, \dots, m_N\}$ is given by

$$M = \prod_{i=1}^N m_i \quad (3.1)$$

Based on CRT, X is related to its residue digits by

$$X = \left| \sum_{i=1}^N M_i |M_i^{-1}|_{m_i} x_i \right|_M \quad (3.2)$$

where $M_i = \frac{M}{m_i}$ and $|M_i^{-1}|$ is the multiplicative inverse of $|M_i|_{m_i}$.

The proposed scaling algorithm is designed for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, so $N = 3$ and (3.2) becomes

$$X = \left| m_2 m_3 |M_1^{-1}|_{m_1} x_1 + m_1 m_3 |M_2^{-1}|_{m_2} x_2 + m_1 m_2 |M_3^{-1}|_{m_3} x_3 \right|_M \quad (3.3)$$

The properties listed in Section 2.1 and the following theorems are used for the derivation of the scaling algorithm.

Theorem 3.1: Given $p = kq$, where k is an integer, $\left| |X|_p \right|_q = |X|_q$.

Proof: Based on the definition of modulo operation [She89],

$$|X|_p = X - \varepsilon p, \varepsilon = \lfloor X / p \rfloor \quad (3.4)$$

From *Properties 2.2* and *2.3*,

$$\left| |X|_p \right|_q = |X - \varepsilon p|_q = \left| |X|_q - |\varepsilon|_q \cdot |p|_q \right|_q \quad (3.5)$$

Since p is divisible by q , $|p|_q = 0$,

$$\left| |X|_p \right|_q = |X|_q \quad (3.6)$$

Theorem 3.2 [Con99]: Given the moduli set of $\{m_1, m_2, m_3\}$ with $m_1 = 2^n - 1$, $m_2 = 2^n$ and $m_3 = 2^n + 1$, the followings hold true.

$$\left| M_1^{-1} \right|_{m_1} = 2^{n-1} \quad (3.7)$$

$$\left| M_2^{-1} \right|_{m_2} = 2^n - 1 \quad (3.8)$$

$$\left| M_3^{-1} \right|_{m_3} = 2^{n-1} + 1 \quad (3.9)$$

3.2.2 New Formulation of RNS Scaling

Based on the above preliminaries, the following theorem is proposed to reduce the complexity of inter-modulo operation of RNS scaling.

Theorem 3.3: Let $x_i = |X|_{m_i}$ for $i = 1, 2, 3$ and $m_1 = 2^n - 1$, $m_2 = 2^n$ and $m_3 = 2^n + 1$. If $k = m_2 = 2^n$, then

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_1} = |x_1 - x_2|_{m_1} \quad (3.10)$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_2} = \left| \left((2^{2n-1} + 2^{n-1})x_1 - 2^n x_2 + (2^{2n-1} + 2^{n-1} - 1)x_3 \right) \right|_{m_1 m_3} \Big|_{m_2} \quad (3.11)$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_3} = |x_2 + 2^n x_3|_{m_3} \quad (3.12)$$

The formulation of this theorem and its proof is given as follows.

By definition, scaling an integer variable X by a constant integer k can be obtained by dividing both sides of (3.3) by k and then taking its floor value. Let Y be the integer result of the scaling operation, using *Property 2.5*,

$$\begin{aligned}
Y &= \left\lfloor \frac{X}{k} \right\rfloor \\
&= \left\lfloor \frac{1}{k} \left| m_2 m_3 \right| M_1^{-1} \Big|_{m_1} x_1 + m_1 m_3 \left| M_2^{-1} \right|_{m_2} x_2 + m_1 m_2 \left| M_3^{-1} \right|_{m_3} x_3 \right\rfloor_M \\
&= \left\lfloor \left| \frac{m_2 m_3}{k} \right| M_1^{-1} \Big|_{m_1} x_1 + \frac{m_1 m_3}{k} \left| M_2^{-1} \right|_{m_2} x_2 + \frac{m_1 m_2}{k} \left| M_3^{-1} \right|_{m_3} x_3 \right\rfloor_{\frac{M}{k}}
\end{aligned} \tag{3.13}$$

where $\lfloor \square \rfloor$ is the least integer function.

k is chosen to be equal to $m_2 = 2^n$ (The choice of k is crucial in ensuring that the truncation error is negligible and this will be discussed shortly), which produces

$$\left\lfloor \frac{X}{k} \right\rfloor = \left\lfloor \left| m_3 \right| M_1^{-1} \Big|_{m_1} x_1 + \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 + m_1 \left| M_3^{-1} \right|_{m_3} x_3 \right\rfloor_{m_1 m_3} \tag{3.14}$$

Using (3.14), the scaled integer can be computed directly from the RNS representation of X .

It is desired to have the output of the RNS scaler to be generated directly in the residue form so that it can be directly processed by the arithmetic unit in each residue channel. The residue digit in each channel can be computed by taking the corresponding modulo operation on both sides of (3.14) as follows:

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_i} = \left\| m_3 \left| M_1^{-1} \right|_{m_1} x_1 + \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 + m_1 \left| M_3^{-1} \right|_{m_3} x_3 \right\|_{m_1 m_3} \Big|_{m_i} \tag{3.15}$$

where $i = 1, 2$ and 3 .

For m_1 and m_3 channels, according to *Theorem 3.1*, (3.15) can be simplified to

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_i} = \left| m_3 \left| M_1^{-1} \right|_{m_1} x_1 + \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 + m_1 \left| M_3^{-1} \right|_{m_3} x_3 \right|_{m_i} \tag{3.16}$$

where $i = 1, 3$.

Each independently scaled residue of (3.16) can be further reduced to

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_1} = \left\lfloor m_3 \left| M_1^{-1} \right|_{m_1} x_1 + \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 \right\rfloor_{m_1} \quad (3.17)$$

and

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_3} = \left\lfloor \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 + m_1 \left| M_3^{-1} \right|_{m_3} x_3 \right\rfloor_{m_3} \quad (3.18)$$

For modulus m_2 ,

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_2} = \left\lfloor \left\lfloor m_3 \left| M_1^{-1} \right|_{m_1} x_1 + \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} x_2 + m_1 \left| M_3^{-1} \right|_{m_3} x_3 \right\rfloor_{m_1 m_3} \right\rfloor_{m_2} \quad (3.19)$$

As modulo 2^n of a variable is the same as keeping only the n least significant bits of it, the scaled residue digit, y_2 for the modulus m_2 can be obtained by simply hardwiring the n least significant bits of the result computed from the right hand side of (3.19). There is no additional logic and delay cost incurred by direct hardwiring.

It is observed that $\frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2}$ is a common term in (3.17), (3.18) and (3.19). Substituting the expressions of m_1 , m_2 , m_3 and $\left| M_2^{-1} \right|_{m_2}$ for this common term,

$$\begin{aligned} \left\lfloor \frac{m_1 m_3}{m_2} \left| M_2^{-1} \right|_{m_2} \right\rfloor &= \left\lfloor \frac{(2^n - 1)(2^n + 1)(2^n - 1)}{2^n} \right\rfloor \\ &= \left\lfloor \frac{2^{3n} - 2^{2n+1} + 2^{2n} + 2^n - 2^{n+1} + 1}{2^n} \right\rfloor \\ &\cong 2^{2n} - 2^{n+1} + 2^n - 1 \end{aligned} \quad (3.20)$$

The least integer function $\lfloor \cdot \rfloor$ of the scaling operation in RNS justifies the truncation of (3.20). The effect of this approximation will be analyzed in Section 3.2.3.

By substituting (3.20) into (3.17), (3.18) and (3.19),

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_1} = \left\lfloor (2^{2n-1} + 2^{n-1})x_1 + (2^{2n} - 2^{n+1} + 2^n - 1)x_2 \right\rfloor_{m_1} \quad (3.21)$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_2} = \left\| (2^{2n-1} + 2^{n-1})x_1 + (2^{2n} - 2^{n+1} + 2^n - 1)x_2 \right. \\ \left. + (2^{2n-1} + 2^n - 2^{n-1} - 1)x_3 \right\|_{m_1 m_3} \Big|_{m_2} \quad (3.22)$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_3} = \left\| (2^{2n} - 2^{n+1} + 2^n - 1)x_2 + (2^{2n-1} + 2^n - 2^{n-1} - 1)x_3 \right\|_{m_3} \quad (3.23)$$

Since $|2^n|_{2^n-1} = |1|_{2^n-1}$, as a corollary of *Properties 2.2* and *2.3*, the following periodic property for modulus m_1 can be defined [Pie94], [Pie02].

Corollary 1: For any non-negative integer, a

$$|2^{an+b}|_{m_1} = |2^b|_{m_1} \quad (3.24)$$

By applying *Properties 2.2* and *2.3* and *Corollary 1*, and using $m_1 m_3 = (2^n - 1)(2^n + 1) = 2^{2n} - 1$, (3.21), (3.22) and (3.23) can be reduced eventually to the following highly simplified expressions in *Theorem 3.3*.

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_1} = |x_1 - x_2|_{m_1}$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_2} = \left\| (2^{2n-1} + 2^{n-1})x_1 - 2^n x_2 + (2^{2n-1} + 2^{n-1} - 1)x_3 \right\|_{m_1 m_3} \Big|_{m_2}$$

$$\left\lfloor \frac{X}{k} \right\rfloor_{m_3} = |x_2 + 2^n x_3|_{m_3}$$

Example 3.1: Consider the integer $X = 3316100$ with the RNS representation of $(80, 132, 29)$ for the moduli set $\{255, 256, 257\}$. Using (3.10), (3.11) and (3.12), the scaled output in RNS representation is calculated in Table 3.1. It can be verified that the RNS number $(203, 153, 103)$ is equivalent to the integer, 12953, computed directly from $\lfloor 3316100 / 256 \rfloor$. Note that the binary representation of $12953 = 0011001010011001_2$ is a byproduct in the computation of the scaled residue y_2 before the final modulo of (3.11) is taken. Its eight least significant bits are $10011001_2 = 153$.

Table 3.1: Computation traces of scaling (80, 132, 29) by $k=256$ in RNS $\{255, 256, 257\}$

$\lfloor \lfloor X/k \rfloor \rfloor_{m_1}$	$ 80 - 132 _{255} = 203$
$\lfloor \lfloor X/k \rfloor \rfloor_{m_2}$	$ 32896 \times 80 - 256 \times 132 + 32895 \times 29 _{65535} _{256} = 12953 _{256} = 153$
$\lfloor \lfloor X/k \rfloor \rfloor_{m_3}$	$ 132 + 256 \times 29 _{257} = 103$

3.2.3 Error Analysis

The second term of the exact scaling equation of (3.14) is truncated to produce the approximation of

$$\left\lfloor \frac{X}{k} \right\rfloor' = \left\lfloor \begin{aligned} & \left((2^{2n-1} + 2^{n-1})x_1 + (2^{2n} - 2^{n+1} + 2^n - 1)x_2 \right) \\ & + (2^{2n-1} + 2^n - 2^{n-1} - 1)x_3 \end{aligned} \right\rfloor_{m_1 m_3} \quad (3.25)$$

If $k_1 = 2^{2n-1} + 2^{n-1}$, $k_2 = 2^{2n} - 2^{n+1} + 2^n - 1$, $k_3 = 2^{2n-1} + 2^n - 2^{n-1} - 1$, and $p = m_1 m_3$, then (3.25) can be rewritten as

$$\left\lfloor \frac{X}{k} \right\rfloor' = \left\lfloor k_1 x_1 + k_2 x_2 + k_3 x_3 \right\rfloor_p \quad (3.26)$$

The integer resulting from the above modulo operation can be expressed using the definition of (3.4).

$$\left\lfloor \frac{X}{k} \right\rfloor' = \left\lfloor k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p \right\rfloor \quad (3.27)$$

where ε is a non-negative integer.

Without truncation, the exact equation, (3.14), is given by

$$\begin{aligned}
\left\lfloor \frac{X}{k} \right\rfloor &= \left\lfloor \left| \left(2^{2n-1} + 2^{n-1} \right) x_1 + \left(2^{2n} - 2^{n+1} + 2^n - 1 + \frac{1}{2^n} \right) x_2 \right. \right. \\
&\quad \left. \left. + \left(2^{2n-1} + 2^n - 2^{n-1} - 1 \right) x_3 \right|_{m_1 m_3} \right\rfloor \\
&= \left\lfloor \left| k_1 x_1 + \left(k_2 + \frac{1}{2^n} \right) x_2 + k_3 x_3 \right|_p \right\rfloor \\
&= \left\lfloor \left| k_1 x_1 + k_2 x_2 + k_3 x_3 + \frac{x_2}{2^n} \right|_p \right\rfloor \\
&= \left\lfloor \left| k_1 x_1 + k_2 x_2 + k_3 x_3 + c \right|_p \right\rfloor \\
&= \left\lfloor k_1 x_1 + k_2 x_2 + k_3 x_3 + c - \varepsilon p \right\rfloor
\end{aligned} \tag{3.28}$$

where $c = \frac{x_2}{2^n}$.

Since $\left| k_1 x_1 + k_2 x_2 + k_3 x_3 \right|_p = k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p$ is an integer, based on the definition of the least integer function, (3.27) and (3.28) can be rewritten as:

$$\left\lfloor \frac{X}{k} \right\rfloor' = k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p \tag{3.29}$$

$$\left\lfloor \frac{X}{k} \right\rfloor = k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p + \lfloor c \rfloor \tag{3.30}$$

Since $0 \leq x_2 < 2^n$, it is true that $0 \leq c < 1$ and $\lfloor c \rfloor = 0$. Therefore,

$$\left\lfloor \frac{X}{k} \right\rfloor = k_1 x_1 + k_2 x_2 + k_3 x_3 - \varepsilon p = \left\lfloor \frac{X}{k} \right\rfloor' \tag{3.31}$$

This proves that the proposed RNS scaling algorithm is exact and does not introduce any scaling error.

3.3 Hardware Implementation

From (3.10), since $|-x_2|_{2^n-1} = |2^n - 1 - x_2|_{2^n-1} = \bar{x}_2$, the generation of the scaled RNS residue, y_1 requires a two-operand modulo $2^n - 1$ adder, which can be implemented using a Ling parallel-prefix modulo $2^n - 1$ adder [Dim05].

The generation of the scaled RNS residue, y_2 by (3.11) needs some attention. The two extensively used special properties of modulo $2^n - 1$ arithmetic, *Properties 2.6* and *2.7*, proposed by [Sza67] are exploited here to simplify the implementation.

Let the j -th bit of a residue x_i be represented as x_{ij} . The three residues to be scaled can then be represented in their binary forms as follows: $x_1 = (x_{1,n-1}x_{1,n-2} \cdots x_{1,0})_2$, $x_2 = (x_{2,n-1}x_{2,n-2} \cdots x_{2,0})_2$ and $x_3 = (x_{3,n}x_{3,n-1} \cdots x_{3,0})_2$.

By applying *Properties 2.6* and *2.7* to each term of (3.11),

$$\begin{aligned} P_1 &= |2^{2n-1} x_1|_{2^{2n}-1} = CLS_{2n}(x_1, 2n-1) \\ &= \left(x_{1,0} \underbrace{00 \cdots 0}_n x_{1,n-1} x_{1,n-2} \cdots x_{1,1} \right)_2 \end{aligned} \quad (3.32)$$

$$\begin{aligned} P_2 &= |2^{n-1} x_1|_{2^{2n}-1} = CLS_{2n}(x_1, n-1) \\ &= \left(0 x_{1,n-1} x_{1,n-2} \cdots x_{1,0} \underbrace{00 \cdots 0}_{n-1} \right)_2 \end{aligned} \quad (3.33)$$

$$\begin{aligned} P_3 &= |-2^n x_2|_{2^{2n}-1} = CLS_{2n}(\bar{x}_2, n) \\ &= \left(\bar{x}_{2,n-1} \bar{x}_{2,n-2} \cdots \bar{x}_{2,0} \underbrace{11 \cdots 1}_n \right)_2 \end{aligned} \quad (3.34)$$

$$\begin{aligned} P_4 &= |2^{2n-1} x_3|_{2^{2n}-1} = CLS_{2n}(x_3, 2n-1) \\ &= \left(x_{3,0} \underbrace{00 \cdots 0}_{n-1} x_{3,n} x_{3,n-1} \cdots x_{3,1} \right)_2 \end{aligned} \quad (3.35)$$

$$\begin{aligned}
P_5 &= \left| 2^{n-1} x_3 \right|_{2^{2n-1}} = CLS_{2n}(x_3, n-1) \\
&= \left(x_{3,n} x_{3,n-1} \cdots x_{3,0} \underbrace{00 \cdots 0}_{n-1} \right)_2
\end{aligned} \tag{3.36}$$

$$P_6 = \left| -x_3 \right|_{2^{2n-1}} = \left(\underbrace{11 \cdots 1}_{n-1} \bar{x}_{3,n} \bar{x}_{3,n-1} \cdots \bar{x}_{3,0} \right)_2 \tag{3.37}$$

The sum of the binary vectors $P_1 + P_2$ can be expressed as a single term $Q_1 = (x_{1,0} x_{1,n-1} x_{1,n-2} \cdots x_{1,0} x_{1,n-1} x_{1,n-2} \cdots x_{1,1})_2$. The remaining four addends can be reduced to three binary vectors by reordering the bits of different addends at the same position followed by logic minimization. As shown in Figure 3.1(a), the 1-string in P_3 is first swapped with $\bar{x}_{3,n} \bar{x}_{3,n-1} \cdots \bar{x}_{3,0}$ in P_6 . Then, the 0-string in P_5 is swapped with the corresponding 1-string that has just been transferred to P_6 . After that, $x_{3,n}$ in P_4 is swapped with $x_{3,0}$ in P_5 . Finally, the 0-string in P_4 , $x_{3,n-1} x_{3,n-2} \cdots x_{3,1}$ in P_5 and the 1-string in P_6 are interchanged. The strings to be swapped are boxed and annotated with its order of execution in Figure 3.1(a). The vectors after bit reordering are shown in Figure 3.1(b). The last two vectors (in the dotted box) of Figure 3.1(b) can be merged into a single constant vector of $(11 \cdots 1)_2$ when $x_{3,n} = 0$ and $(1011 \cdots 1011 \cdots 1)_2$ when $x_{3,n} = 1$. Since $\left| (1011 \cdots 1011 \cdots 1)_2 \right|_{2^{2n-1}} = (01 \cdots 110 \cdots 0)_2$, the merged term is represented by $Q_4 = (\bar{x}_{3,n} \underbrace{1 \cdots 1}_n \underbrace{\bar{x}_{3,n} \cdots \bar{x}_{3,n}}_{n-1})_2$ in Figure 3.1(c).

Since $x_3 \leq 2^n$, $x_{3,n} = 1$ if $x_3 = 2^n$. Also, when $x_3 = 2^n$, $x_{3,i} = 0 \forall i \leq n-1$, the n lower order bits of Q_2 becomes '1's, Q_3 becomes an all-zero string and $Q_4 = (01 \cdots 110 \cdots 00)_2$. The three nonzero binary vectors, Q_1 , Q_2 and Q_4 , can be summed using a $2n$ -bit carry save adder (CSA) with end-around carry (EAC). On the other hand, when $x_{3,n} = 0$, Q_4 is an all-one string. If Q_1 , Q_2 and Q_3 are summed using a $2n$ -bit CSA with EAC, the EAC from the CSA will propagate all the way to the carry out, and wrap around to the least significant bit (LSB). Hence, no additional $2n$ -bit CSA is needed to sum the all-one vector, Q_4 . The same $2n$ -bit CSA can therefore be reused by replacing Q_3 by Q_4 when $x_{3,n} = 1$. The n multiplexed addend bits to the CSA can be implemented by $x_{3,n} + x_{3,i}$, where $i = 0, 1, \dots, n-1$ and '+' denotes logical OR operation. Consequently, the four binary vectors, Q_1 to Q_4 can be added using only one $2n$ -bit

modulo $2^{2n} - 1$ CSA with EAC and n two-input *OR* gates, followed by a parallel-prefix modulo $2^{2n} - 1$ ladder. The four input vectors are formed by routing the binary bits from the input residues without using any hardware logic. It is important to note that the maximum delay of the CSA tree is kept constant at one FA delay regardless of the value of n . The residue y_2 of the scaled integer output Y for the modulus m_2 can be taken directly from the least significant n bits of the $2^{2n} - 1$ adder.

Figure 3.1: Reduction of addends

To produce y_3 from (3.12), the following axiom [Mur08] are needed.

$$\text{Axiom 3.1: } \left| 2^{2n} a \right|_{2^n+1} = |a|_{2^n+1}$$

where $a \in \{0, 1\}$.

By applying *Axioms 3.1* and *Property 2.9* to the second term of (3.12), the variable bits of this term are wrapped to the n least significant bit positions and its expression becomes:

$$\left| 2^n x_3 \right|_{2^n+1} = \left| \sum_{i=0}^n 2^{n+i} x_{3,i} \right|_{2^n+1} = \left| x_{3,n} + \sum_{i=0}^{n-1} 2^i \bar{x}_{3,i} + \sum_{i=n}^{2n-1} 2^i \right|_{2^n+1} \quad (3.38)$$

The addition of x_2 to (3.38) can be reduced by a CSA tree before a diminished-one mod $2^n + 1$ adder [Ver02] is used to produce the final output. A one needs to be subtracted from

each of the sum and carry outputs of the CSA tree to convert them into diminished-one representation before the diminished-one adder, and the output of the diminished-one adder needs to be incremented by one to obtain the actual output. This aggregate offset of -1 can be added upfront in the CSA tree. According to *Property 2.9*, the carry generated out of the most significant bit of each n -bit CSA will be complemented and added to the least significant bit of the next level of CSA together with a constant addition of 2^n . By assuming a three-level CSA tree, an offset of 3×2^n will be generated. Together with the compensation for diminished one addition and the last term of (3.38), the total correction constant, C to be added in the CSA tree is given by:

$$\begin{aligned}
C &= \left\lfloor \sum_n^{2^n-1} 2^i - 1 + 3 \times 2^n \right\rfloor_{2^n+1} = \left\lfloor 2^{2^n} - 2^n - 1 + 3 \times 2^n \right\rfloor_{2^n+1} \\
&= \left\lfloor 1 - (-1) - 1 + 3 \times (-1) \right\rfloor_{2^n-1} = \left\lfloor -2 \right\rfloor_{2^n+1} \\
&= \left\lfloor 2^n + 1 - 2 \right\rfloor_{2^n+1} = \left\lfloor 2^n - 1 \right\rfloor_{2^n+1}
\end{aligned} \tag{3.39}$$

The addends to be summed in the CSA tree are shown in Figure 3.2(a). The first n -bit CSA adds only the last two addends to produce the results of Figure 3.2(b), with the complementary end-around carry (CEAC) printed bold. Since $1 + a = 2a + \bar{a}$ for any binary variable a , no new signal variable is introduced. The result can be obtained by direct hardwiring without the need for the CSA. The second n -bit CSA adds the last three addends of Figure 3.2(b) to produce the results of Figure 3.2(c), with the CEAC printed bold. Based on the property of $1 + a = 2a + \bar{a}$, only two modified full adders (MFAs), instead of a complete n -bit CSA, are required to produce the unknown variables, s_0 , c_0 , s_1 and c_1 . Finally, an n -bit CSA is required to reduce Figure 3.2(c) to two addends to be fed into the final diminished-one mod $2^n + 1$ adder. Although it is started with a three-level CSA addition, most intermediate results can be directly hardwired and only one level of n -bit CSA with CEAC and two MFAs are needed eventually, which are highlighted by the shaded boxes of Figure 3.2.

The complete RNS scaling architecture is shown in Figure 3.3. Since the inner modulo $2^{2^n} - 1$ operation in (3.11) has to be computed any way to obtain the scaled output y_2 , the scaled integer, Y is also available as a byproduct of this computation. Thus, without introducing any hardware overhead, the scaled output is available in both RNS and binary forms. This flexibility to manipulate the scaled output in two domains has advantage in

interfacing the scaled output to computational units in either domain without the need for the expensive RNS-to-binary converter. For example, the bottleneck reverse converter in the last stage of Figure 1.1(a) can be eliminated to increase the throughput of the RNS IPSP.

$x_{2,n-1}$	$x_{2,n-2}$	...	$x_{2,2}$	$x_{2,1}$	$x_{2,0}$
$\overline{x_{3,n-1}}$	$\overline{x_{3,n-2}}$	...	$\overline{x_{3,2}}$	$\overline{x_{3,1}}$	$\overline{x_{3,0}}$
1	1	...	1	1	1
0	0	...	0	0	$x_{3,n}$

(a) Level 1 CSA with CEAC

$x_{2,n-1}$	$x_{2,n-2}$	...	$x_{2,2}$	$x_{2,1}$	$x_{2,0}$
$\overline{x_{3,n-1}}$	$\overline{x_{3,n-2}}$	...	$\overline{x_{3,2}}$	$\overline{x_{3,1}}$	$\overline{x_{3,0}}$
1	1	...	1	1	$\overline{x_{3,n}}$
0	0	...	0	$x_{3,n}$	1

(b) Level 2 CSA with CEAC

$x_{2,n-1}$	$x_{2,n-2}$	...	$x_{2,2}$	$x_{2,1}$	$x_{2,0}$
$x_{3,n-1}$	$x_{3,n-2}$	...	$x_{3,2}$	s_1	s_0
$\overline{x_{3,n-2}}$	$\overline{x_{3,n-3}}$	...	c_1	c_0	$x_{3,n-1}$

(c) Level 3 CSA with CEAC

Figure 3.2: Simplification of the CSA tree for the computation of y_3

Figure 3.3: Architecture of the proposed RNS scaler

Example 3.2: Consider the RNS representation of $X = (80, 132, 29)$ for the moduli set $\{255, 256, 257\}$ from Example 3.1. $x_1 = 01010000_2$, $x_2 = 10000100_2$ and $x_3 = 000011101_2$. $Q_1 =$

00101000 00101000₂, $Q_2 = 01111011$ 11100010₂, $Q_3 = 10001110$ 10001110₂ and $Q_4 = 11111111$ 11111111₂. The modulo $2^n - 1$ addition of x_1 and $\bar{x}_2$, carry save addition with end-around carry of the four $2n$ -bit binary vectors, Q_1 to Q_3 , and diminished-one modulo $2^n + 1$ addition of x_2 and $\bar{x}_3$ are illustrated in Figure 3.4. From Figure 3.4, $Y = 0011001010011001_2 = 12953$, $y_1 = 11001011_2 = 203$, $y_2 = 10011001_2 = 153$ and $y_3 = 01100111_2 = 103$ tally with the correct outputs from Table 3.1.

In the final part of this section, it is shown that the final modulo $2^{2n}-1$ adder in m_2 channel can be replaced by a much simpler modulo 2^n adder if only the residue output y_2 is needed.

From [Dim05],

$$|A+B|_{2^{2n}-1} = \begin{cases} |A+B|_{2^{2n}} + 1 & \text{if } A+B \geq 2^{2n} \\ |A+B|_{2^{2n}} & \text{Otherwise} \end{cases} \quad (3.40)$$

It follows that

$$\left\| |A+B|_{2^{2n}-1} \right\|_{2^n} = \begin{cases} \left\| |A+B|_{2^{2n}} + 1 \right\|_{2^n} & \text{if } A+B \geq 2^{2n} \\ \left\| |A+B|_{2^{2n}} \right\|_{2^n} & \text{Otherwise} \end{cases} \quad (3.41)$$

Using *Theorem 3.1*,

$$\left\| |A+B|_{2^{2n}-1} \right\|_{2^n} = \begin{cases} |A+B|_{2^n} + 1 & \text{if } A+B \geq 2^{2n} \\ |A+B|_{2^n} & \text{Otherwise} \end{cases} \quad (3.42)$$

Only a few additional logic gates are needed to compute the carry-in to the modulo 2^n adder based on the Ling's carry definition [Dim05].

X_1		0	1	0	1	0	0	0	0												
$\bar{x}_2$	+	0	1	1	1	1	0	1	1												
y_1		1	1	0	0	1	0	1	1												
Q_1		0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0				
Q_2		0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	0				
Q_3	+	1	0	0	0	1	1	1	0	1	0	0	0	1	1	1	0				
Sum		1	1	0	1	1	1	0	1	0	1	0	0	0	1	0	0				
Carry	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0					
EAC																	0				
		1	0	0	1	1	0	0	1	0	1	0	0	1	1	0	0				
EAC																	1				
Y		0	0	1	1	0	0	1	0	y_2				1	0	0	1	1	0	0	1

$$(c_0, s_0) = \overline{x_{3,0}} + \overline{x_{3,n}} + 1 = 0 + 1 + 1 = (1, 0)$$

$$(c_1, s_1) = \overline{x_{3,1}} + 1 + x_{3,n} = 1 + 1 + 0 = (1, 0)$$

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 CSA & & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 + & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
 \hline
 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \\
 \\
 \begin{array}{cccccccc}
 \text{CEAC} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \\
 + & & & & & & & & & 0 \\
 \hline
 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
 \text{CEAC} & + & & & & & & & & 1 \\
 \hline
 y_3 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 &
 \end{array}
 \end{array}$$

Figure 3.4: Calculation of scaled RNS output, y_1 , y_2 and y_3 for Example 3.2

3.4 Evaluation and Comparison

In this section, the total delay and hardware area of the proposed design are modelled and simulated in order to analyze its hardware implementation cost and complexity. In order to benchmark the proposed RNS scaler architecture against the FA based scaler architecture [Das08] and other RNS scaler architectures compared in [Das08], the unit-gate model used in [Ver02] is adopted for the analysis. In this analytical model, a two-input monotonic gate, such as *AND* or *NAND* gate, is said to have one unit of area and one unit of delay. An *XOR* gate is deemed to consume two units of area and have two units of delay. The area and delay of an inverter is a negligible fraction of a unit and hence it is assumed to have zero unit of area and delay. Based on the unit-gate model, a FA has seven units of area and 4 units of delay.

Based on the architecture (See Figure 3.3) described in Section 3.3, the area and delay for each residue channel can be independently evaluated by studying the area and time complexity of the logic gate implementation of the carry save adder tree, if any, and the carry propagating adder (CPA). Based on Figure 6 of [Dim05], the area and delay for the Ling modulo $2^n - 1$ adder are estimated to be $3n\lceil\log_2 n - 1\rceil + 12n$ and $2\lceil\log_2 n - 1\rceil + 3$ units, respectively. From Table 2 of [Ver02], the area and time complexity for the diminished-one mod $2^n + 1$ adder are $4.5n\lceil\log_2 n\rceil + 0.5n + 6$ and $2\lceil\log_2 n\rceil + 3$ units, respectively. For the m_1 channel, n inverters for the one-complement of an operand before the CPA are needed. A $2n$ -bit CSA with EAC and n *OR* gates are required in the m_2 channel. $2n$ FAs are required to implement the CSA. Finally, from Figure 3.2, two MFAs and n FAs are required to implement the CSA with CEAC for the m_3 channel. The decomposition of area and delay for each channel is tabulated in Table 3.2 and Table 3.3, respectively.

Table 3.2: Estimation of unit-gate area of proposed design

Modulus	A_{CPA}	A_{other}	A_{Total}
m_1	$3n\lceil\log_2 n - 1\rceil + 12n$	≈ 0	$3n\lceil\log_2 n - 1\rceil + 12n$
m_2	$6n\lceil\log_2 n\rceil + 24n$	$15n$	$6n\lceil\log_2 n\rceil + 39n$
m_3	$4.5n\lceil\log_2 n\rceil + 0.5n + 6$	$7n + 6$	$4.5n\lceil\log_2 n\rceil + 7.5n + 12$

Table 3.3: Estimation of unit-gate delay of proposed design

Modulus	D_{CPA}	D_{other}	D_{Total}
m_1	$2 \lceil \log_2 n - 1 \rceil + 3$	≈ 0	$2 \lceil \log_2 n - 1 \rceil + 3$
m_2	$2 \lceil \log_2 n \rceil + 3$	5	$2 \lceil \log_2 n \rceil + 8$
m_3	$2 \lceil \log_2 n \rceil + 3$	6	$2 \lceil \log_2 n \rceil + 9$

It is worth noting here that the delays of the non-CPA logic in all three modulus channels are independent of the word length n . In addition, the delay of the carry propagating addition is only logarithmically dependent on n . This means that the rate of increase in delay actually slows down as n grows larger. In fact, the speed is merely dependent on the performance of the modulo $2^n + 1$ and $2^n - 1$ adders. In other words, the proposed scaling algorithm has successfully circumvented the complexity of RNS scaling problem for $\{2^n - 1, 2^n, 2^n + 1\}$ by reducing it into an operation as simple as addition or constant multiplication in the same RNS.

The area and time complexity of the proposed RNS scaler are also benchmarked against other designs reported in the RNS scaling architecture [Das08], including [Das08] itself. All the implementations are based on ROMs except for the proposed scaler and [Das08]. It is worth noting that only the proposed design and [Gar99] have no scaling error. The other scaling methods introduce some non-zero scaling error. Even though the error may be small, it could escalate to an unacceptable magnitude after several iterations of computation as shown in Figure 1.1.

For consistency, the same method of evaluation as [Das08] is adopted so that the hardware area and time complexity of the contending designs can be excerpted directly from Table IV of [Das08] for comparison. Since the hardware complexity is expressed in terms of the number of transistors for all designs, the transistor count of the proposed design can be estimated from its unit-gate area by reasonably and conservatively assumed that a two-input monotonic gate can be implemented with six transistors using a classical CMOS implementation. In addition, an inverter is considered as equivalent to two transistors. The conversion of the unit-gate area from Table 3.2 to the equivalent number of transistors for the proposed design is shown in Table 3.4.

Table 3.4: Estimated number of transistors of proposed design

Modulus	A_{CPA}	A_{other}	A_{Total}
m_1	$18n \lceil \log_2 n - 1 \rceil + 72n$	$2n$	$18n \lceil \log_2 n - 1 \rceil + 74n$
m_2	$36n \lceil \log_2 n \rceil + 144n$	$90n$	$36n \lceil \log_2 n \rceil + 234n$
m_3	$27n \lceil \log_2 n \rceil + 3n + 36$	$42n + 36$	$27n \lceil \log_2 n \rceil + 45n + 72$

The same unit-gate delay model is used in [Das08] for the comparison of time complexity except that a classic FA is assumed to have only 2 units of delay in [Das08], which is equivalent to that of an *XOR* gate. This estimate is optimistic because the delay of a classic 28T FA implemented in static CMOS technology [Cha04], [Cha05], [Sha02] is at least 3 units. In other words, the total delay of [Das08] has been underestimated.

As the use of RNS scaler is to ensure that the computed result represents the real value instead of its residue modulo M , it is fairer to compare different RNS scalars based on the same dynamic range. As this is not always possible due to the odd values of moduli selected by [Das08], the value of n for the proposed design has been selected such that its dynamic range is at least several time larger than the dynamic range of other designs so that the comparison is always in favor of the contenders. Table 3.5 and Table 3.6 show the comparisons of the transistor counts and the delay, respectively, of six different RNS scaling architectures for four different dynamic ranges.

Table 3.5: Comparison of estimated number of transistors (Value in parenthesis indicates the percentage area reduction of proposed design over its contenders)

Proposed $\{m_1, m_2, m_3\}$ DR	[Das08] $\{m_1, m_2\}$ DR	Number of transistors ([contender-this]/contender×100%)					
		Proposed	[Das08]	[Gar99]	[Ulm93]	[She89]	[Gri88]
$\{7, 8, 9\}$ 504	$\{5, 17\}$ 85	1563	7476 (79.09)	22588 (93.08)	148862 (98.95)	20812 (92.49)	8618 (81.86)
$\{15, 16, 17\}$ 4080	$\{17, 113\}$ 1921	2060	10624 (80.61)	485684 (99.58)	320942 (99.36)	416816 (99.51)	87132 (97.64)
$\{31, 32, 33\}$ 32736	$\{97, 113\}$	2962	11940 (75.19)	928152 (99.68)	323438 (99.08)	1216816 (99.76)	89196 (96.68)
$\{255, 256, 257\}$ 16776960	10961	4696	11940 (60.67)	928152 (99.49)	323438 (98.55)	1216816 (99.61)	89196 (94.74)

Table 3.6: Comparison of estimated unit-gate delay (value in parenthesis indicates percentage delay reduction of proposed design over its contenders)

Proposed $\{m_1, m_2, m_3\}$ DR	[Das08] $\{m_1, m_2\}$ DR	Unit-gate delay ([contender–this]/contender $\times$ 100%)					
		Proposed	[Das08]	[Gar99]	[Ulm93]	[She89]	[Gri88]
$\{7, 8, 9\}$ 504	$\{5, 17\}$ 85	13	167 (92.22)	17 (23.53)	42 (69.05)	22 (40.91)	42 (69.05)
$\{15, 16, 17\}$ 4080	$\{17, 113\}$ 1921	13	307 (95.77)	21 (38.10)	50 (74)	22 (40.91)	54 (75.93)
$\{31, 32, 33\}$ 32736	$\{97, 113\}$ 10961	15	131 (88.55)	21 (28.57)	50 (70)	26 (42.31)	56 (73.21)
$\{255, 256, 257\}$ 16776960		15	131 (88.55)	21 (28.57)	50 (70)	26 (42.31)	56 (73.21)

It is obvious from Table 3.5 and Table 3.6 that the proposed design is significantly faster and has much lower hardware complexity than all the other designs. The proposed design reduces by 79.09%, 80.61%, 75.19% and 60.67% of the transistor counts required by the design of [Das08] for an extended DR of 504 ($\approx$ six times), 4080 ($\approx$ twice), 32736 ($\approx$ triple) and 16776960 ($>$ 1500 times), respectively. Although both implementations are FA based, many more number of stages of modulo additions are required by [Das08] (See Figure 2.4) in each channel to produce the residue digits of the scaled integer, which apparently also causes it to be the slowest design. Furthermore, the proposed design has also shortened the delay of the fastest two-lookup cycle design of [Gar99] by respectively 23.53%, 38.10% and 28.57% for approximately six times, twice and triple the dynamic range of its contender. Owing to the logarithmically dependence on the input bit width, the delay of the proposed design increases relatively slower with dynamic range than the two-lookup cycle design. In fact, the delay of the proposed design will remain constant at 15 units for up to $n = 8$ (equivalent to a DR of approximately 2^{24}).

Although Table 3.5 and Table 3.6 provide a good insight on the relative competitiveness of various designs based on the same method of evaluation as [Das08], the moduli sets used are different from the proposed one. For a more accurate evaluation and fairer comparison, the fastest ROM-based design [Gar99] from Table 3.6 has been selected for implementation using the same three moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ and scaling factor 2^n . In addition, another fast and precise base-extension ROM-based design [Bar95] and the current-art adder-based design [Ye08] that were not compared in [Das08] are also implemented using the same moduli set and scaling factor. It is worth noting that the designs of [Bar95] and [Ye08] have zero scaling error. Unit-gate area and delay of these designs are also analyzed. For this

theoretical analysis, the design [Ye08] without the sign number handling is evaluated against the proposed design without the scaled integer output.

Since the designs, [Bar95] and [Gar99], are implemented using ROMs, a ROM model based on [Ulm98] is used to estimate the number of transistors A_{ROM} and the unit-gate delay T_{ROM} . For a $2^n \times p$ ROM [Ulm98], $A_{ROM(2^n \times p)} = 2^{\lceil n/2 \rceil} (\lceil n/2 \rceil + 1) + 2^n p + 2^{\lfloor n/2 \rfloor} p (\lfloor n/2 \rfloor + 2) + p(2^{\lfloor p/2 \rfloor} + 1)$ and $T_{ROM(2^n \times p)} = (1 + \lceil \log_2 n \rceil + \lceil n/2 \rceil) \cdot t_{NAND}$. It is stated in [Gar99] that, for three moduli set with one of the modulus being the scaling constant, the design requires three double-input ROMs. Therefore, the total number of transistors of the design [Gar99] is estimated to be $A_{ROM(2^{2n} \times n)} + A_{ROM(2^{2n+1} \times \{n+1\})} + A_{ROM(2^{2n+1} \times n)}$, where n is the bit width of the moduli. Also, its total unit-gate delay is approximated to be $(T_{ROM(2^{2n+1} \times \{n+1\})} + T_{ROM(2^{2n+1} \times n)})$ units. Based on Fig. 2 of [Bar95], additional block of hardware should be included in each of the m_1 and m_3 channels of [Bar95] to perform the base extension operation of $\|X\|_k|_{m_i}$, for $i = 1$ and 3 . In this special case where $k = 2^n$ and $m_3 > m_1$, only one ROM is required in the m_1 channel. As a result, the total hardware of the design [Bar95] is estimated to be $A_{ROM(2^{2n} \times n)} + A_{ROM(2^{2n+1} \times \{n+1\})} + A_{ROM(2^{2n+1} \times n)} + A_{ROM(2^n \times n)}$ and its total unit-gate delay is $T_{ROM(2^{2n} \times n)} + T_{ROM(2^{2n+1} \times n)} + T_{ROM(2^n \times n)}$ units. For the design [Ye08], the main difference lies in the m_2 channel. As depicted in Fig. 1 of [Ye08], the m_2 channel consists of one modulo $2^n - 1$ adder and one modulo 2^n adder excluding the sign number handling. The former adder has been analyzed at the beginning of this section while the unit-gate area and delay of the latter adder are approximately $1.5n \lceil \log_2 n \rceil + 5n$ and $2 \lceil \log_2 n \rceil + 3$, respectively [Ver02]. For the proposed design without the additional scaled integer output feature, the modulo $2^{2n} - 1$ adder is replaced with a modulo 2^n adder and some glue logic for the Ling's carry. The estimated total number of transistors and unit-gate delay are tabulated in Table 3.7.

Table 3.7: Comparison of estimated total number of transistors (A) and unit-gate delay (T) for the same three moduli sets

$\{m_1, m_2, m_3\}$	Proposed		[Ye08]		[Gar99]		[Bar95]	
	A	T	A	T	A	T	A	T
{31, 32, 33}	2095	17	2077	31	32424	22	32721	29
{63, 64, 65}	2478	17	2478	31	143467	24	144177	31
{127, 128, 129}	2861	17	2879	31	637446	26	638765	34
{255, 256, 257}	3244	17	3280	31	2829481	30	2832513	38

All these full-feature designs (with sign handling as reported in [Ye08] and with both scaled residue and integer outputs for the proposed design) are coded using Verilog HDL and functionally verified using ModelSim. The designs are synthesized and mapped to STM 90nm standard cell library using Synopsys Design Compiler. Each design is optimized for speed to obtain their minimum achievable delay. The synthesized area in μm^2 and critical path delay in ns of these $\{2^n - 1, 2^n, 2^n + 1\}$ RNS scalars for $n = 5, 7$ and 8 are shown in Table 3.8. Due to the memory quota, the two LUT-based scalars are implemented only for $n = 5$ and 7 .

Table 3.8: Comparison of the synthesized area and delay of $\{2^n - 1, 2^n, 2^n + 1\}$ RNS scalars for $n = 5, 7$ and 8

Design	$n = 5$		$n = 7$		$n = 8$	
	<i>Area</i>	<i>Delay</i>	<i>Area</i>	<i>Delay</i>	<i>Area</i>	<i>Delay</i>
Proposed	3168.8	0.87	4590.2	0.88	4869.0	0.93
[Ye08]	4166.5	1.47	6786.5	1.58	7498.8	1.66
[Gar99]	12315.0	0.98	13782.6	1.67	-	-
[Bar95]	10147.3	1.22	15645.2	1.87	-	-

From Table 3.8, it is obvious that the proposed RNS scaler consumes the least area for all values of n . Comparing with the next most area efficient design [Ye08], the proposed design consumes 23.9%, 32.4% and 35.1% less area for $n = 5, 7$ and 8 , respectively. Besides, the proposed design is faster than it by 40.8%, 44.3% and 44.0% for $n = 5, 7$ and 8 , respectively. The significant performance improvement over [Ye08] is attributable to the different approaches to the formulation of scaling problem although both scaling algorithms are designed based on adders. From the synthesis results, it is observed that the critical path delay of the proposed design remains relatively constant for three different values of n . This is consistent with the inference from the unit-gate delay estimation in Table 3.6. Comparing with the fastest design [Gar99] from Table 3.6, the proposed design is faster than it by 11.2% and 47.3% for $n = 5$ and 7 respectively.

The power consumption of the above designs were also simulated using PrimeTime PX with the same technology library. The total power consumption as well as the leakage power for $n = 5, 7$ and 8 are listed in Table 3.9. The results show that the proposed design dissipates the least total power as well as leakage power for all values of n . It consumes 39.3%, 50.2% and

43.4% less total power and 37.0%, 44.1%, 43.1% less leakage power than the next most power-efficient design [Ye08] for $n = 5, 7$ and 8 , respectively.

Table 3.9: Comparison of total power and leakage power in mW (value in parenthesis is the percentage of leakage power) for $n = 5, 7$ and 8 .

Design	$n = 5$		$n = 7$		$n = 8$	
	<i>Total</i>	<i>Leakage</i>	<i>Total</i>	<i>Leakage</i>	<i>Total</i>	<i>Leakage</i>
Proposed	3.796	0.670 (17.7%)	4.955	0.955 (19.3%)	5.821	1.020 (17.5%)
[Ye08]	6.258	1.063 (17.0%)	9.948	1.708 (17.2%)	10.287	1.793 (17.4%)
[Gar99]	13.761	3.140 (22.8%)	16.607	3.144 (18.9%)	-	-
[Bar95]	21.083	2.473 (11.7%)	19.313	3.943 (20.4%)	-	-

3.5 Conclusion

In this chapter, a novel area-efficient, high-speed and precise RNS scaler for the three-moduli set RNS $\{2^n - 1, 2^n, 2^n + 1\}$ is proposed. The new scaling algorithm is formulated based on the Chinese Remainder Theorem and it uniquely exploits the number theoretic properties of this moduli set and the scaling factor of 2^n to overcome the complexity associated with the hardware implementation of inter-modulo operation. The proposed RNS scaler has an area complexity of $O(n \log_2 n)$ and a time complexity of $O(\log_2 n)$. The integer scaled output in normal binary number representation is also generated as a byproduct of this new formulation. Hence, the expensive residue-to-binary converter can be saved if the result after scaling is also required by a normal binary number system. The synthesis results based on 90nm standard-cell based implementation show that the proposed RNS scaler is approximately 30.5 % more area efficient and 43.0% faster than the state-of-the-art adder-based scaler and at least 66% more area efficient and 11.2% faster than the fastest ROM-based scaler design. On average, the total power consumption and leakage power of the proposed design are respectively 44.3% and 41.4% smaller than the most power-efficient adder-based scaler design.

In the next chapter, the problem for which the scaling constant $k = 2^i$, where i needs not be n will be addressed and for the first time, a new programmable RNS scaler for this three moduli set will be proposed and presented.

Chapter 4

A VLSI Efficient Programmable Power-Of-Two Scaler for $\{2^n-1, 2^n, 2^n+1\}$ RNS

4.1 Introduction

Variable power-of-two scaling is an implicit operation in floating point system to manipulate real numbers of wide dynamic range and ease its interfacing to the rest of the system [Par10]. In the case when an expensive floating-point system is unaffordable, the range and precision can also be changed by a single fixed-point system capable of adjusting its binary point position when desired. Scaling by different power-of-two factors in RNS, also known as truncation in a two's complement number system, has been widely used in different parts of the datapath of a large system to reduce the dynamic range and hardware complexity. It is also commonly used to prevent overflow error in DSP entailing an iterative structure of a large number of multiplications and additions, such as FIR and IIR filters. To minimize the risk of over-scaling, adaptive scaling is a preferred technique [Tex99] whereby the scaling factor, usually a power of two, is determined at run time.

While power-of-two scaling can be trivially programmed in conventional binary number system through shift registers in several cycles or combinational shifters in a single cycle, it is a fundamentally difficult operation in RNS because it is a non-weighted number system. Efficient scaling algorithms in RNS have been developed and can be categorized into three main groups [Ma10]: scaling by subset of the moduli set [Bar95], [Bur03], [Cha11], [Gar98], [Gar99], [Phi03], scaling by a factor that is co-prime with the moduli set [Ma10], [Ulm98], [Kon09], and scaling by the power-of-two [Car05b], [Cha11], [Mey03]. Although scaling by a fixed factor in RNS is not without difficulty when it comes to hardware implementation, the problem has been adequately solved for the three moduli set $\{2^n-1, 2^n, 2^n+1\}$ with a scaling

factor 2^n in Chapter 3. Nevertheless, scaling in the same moduli set by a programmable power-of-two factor remains a challenge as the divisor varies and may not be one of the moduli or a product of several moduli of the RNS [Sza67]. All the above algorithms, except [Car05b], have their scaling factor predetermined at design time. The only variable scaling solution [Car05b] is a workaround by cascading r (r is the maximum number of shifts) scaling-by-two blocks to perform the programmable power-of-two scaling operation in RNS. Each of the basic scaling-by-two blocks consists of an RNS sign detector, an RNS parity checker, N (N is the number of moduli) modular adders and N modular multipliers, and some 2-to-1 multiplexers. Besides, additional hardware is required for the scaling of the redundant modulus and error correction. It turns out that for the same dynamic range, a hybrid solution using efficient residue-to-binary [Wan02] and binary-to-residue [She04] converters for the special moduli sets like $\{2^n-1, 2^n, 2^n+1\}$ with a programmable binary shifter could do better than [Car05b]. Unfortunately, the design of [Car05b] is not applicable to such modulo-arithmetic friendly moduli set because it requires the existence of the multiplicative inverse $|2^{-1}|_{m_i}$ for all of its moduli m_i , which means that no even modulus is allowed in the moduli set.

In view of the above reasons, the work presented in this chapter is distinctive as the first solution to the programmable power-of-two scaling problem in $\{2^n-1, 2^n, 2^n+1\}$ RNS with the most efficient hardware architecture reported thus far. With a variable scaling factor of 2^r , $0 \leq r \leq n$, up to one-third of the dynamic range of an unsigned integer can be arbitrarily scaled in the RNS domain directly. The architecture can be implemented entirely in combinational circuits without lookup tables, making it easy to be merged and pipelined with other circuits within the RNS. As the binary scaled output can also be made available by the proposed method, it eases the magnitude comparison of scaled integers.

In what follows, the scaling algorithm is elegantly formulated in Section 4.2 based on the underpinning of CRT. By applying the number theoretic properties, the programmable scaler architecture is simplified and elaborated in Section 4.3. In Section 4.4, it is demonstrated that the area, delay and power consumption of the proposed scaler are remarkably lower than the hybrid solution (which is composed of a current-art residue-to-binary converter [Wan02], a programmable binary logarithmic shifter and a binary-to-residue converter [She04]) for programmable power-of-two scaling in $\{2^n-1, 2^n, 2^n+1\}$. Finally, the conclusion is drawn in Section 4.5.

4.2 Proposed Algorithm for Programmable Power-Of-Two Scaling in RNS

To support variable precision operation without requiring a full-fledged floating-point arithmetic unit, a pseudo-floating-point (PFP) representation is considered, whereby the least significant n -bit of the dynamic range can be truncated by a programmable power-of-two scaling operation while the precision of the remaining part for a given dynamic range of a binary integer representation is guaranteed. The fixed precision part of an integer X in PFP can be expressed as [And88], [Bha98], [Dhu98], [Pie95], [Wan02]

$$\left\lfloor \frac{X}{2^n} \right\rfloor = \frac{X - |X|_{2^n}}{2^n} \quad (4.1)$$

where $\lfloor X \rfloor$ is the least integer function of X .

From (4.1), an integer X can be re-expressed as

$$X = \left\lfloor \frac{X}{2^n} \right\rfloor 2^n + |X|_{2^n} \quad (4.2)$$

The latter term of (4.2) corresponds to the variable precision part of X and is equal to the residue x_2 of the target three moduli set RNS $\{2^n-1, 2^n, 2^n+1\}$.

Without sacrificing the guaranteed minimum precision, the PFP representation of X can be adaptively scaled by an arbitrary power-of-two factor 2^r , $0 \leq r \leq n$, to prevent overflow. This can be done by dividing both sides of (4.2) by 2^r followed by the least integer function, which gives

$$\left\lfloor \frac{X}{2^r} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{X}{2^n} \right\rfloor 2^n + x_2}{2^r} \right\rfloor \quad (4.3)$$

Since $\left\lfloor \frac{X}{2^n} \right\rfloor \frac{2^n}{2^r}$ is an integer, (4.3) becomes

$$\left\lfloor \frac{X}{2^r} \right\rfloor = \left\lfloor \frac{X}{2^n} \right\rfloor \frac{2^n}{2^r} + \left\lfloor \frac{x_2}{2^r} \right\rfloor \quad (4.4)$$

Note that $Y = \left\lfloor \frac{X}{2^r} \right\rfloor$ is the binary representation of the scaled output obtained by dividing the binary representation of X by 2^r . To perform this arbitrary power-of-two scaling directly in the selected RNS domain, the above result must be expressed entirely in the residue form involving only the variables, x_1, x_2 and x_3 , in $\{2^n-1, 2^n, 2^{n+1}\}$. The scaled residue digits y_i for $i = 1, 2$ and 3 can be computed by taking the corresponding modulo operations on both sides of (4.4) as follows:

$$y_i = \left\lfloor \frac{X}{2^r} \right\rfloor_{m_i} = \left\lfloor \frac{X}{2^n} \right\rfloor_{m_i} 2^{n-r} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_i} \quad (4.5)$$

where $i = 1, 2$ and 3 .

From *Properties 2.2* and *2.3*, the residue output in m_1 channel can be written as

$$y_1 = \left\lfloor \frac{X}{2^r} \right\rfloor_{m_1} = \left\lfloor \left\lfloor \frac{X}{2^n} \right\rfloor_{m_1} \cdot 2^{n-r} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_1} \right\rfloor_{m_1} \quad (4.6)$$

By substituting $\left\lfloor \frac{X}{2^n} \right\rfloor_{m_1} = |x_1 - x_2|_{m_1}$ from (3.10) into (4.6),

$$\begin{aligned} y_1 &= \left| |x_1 - x_2|_{m_1} \cdot 2^{n-r} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_1} \right|_{m_1} \\ &= \left| (2^{n-r})x_1 + -(2^{n-r})x_2 + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_1} \right|_{m_1} \end{aligned} \quad (4.7)$$

The generation of y_1 by (4.7) requires a CSA with EAC followed by a modulo 2^n-1 adder. The CSA can be eliminated to reduce the hardware and improve the speed by merging the second and third terms of (4.7) into a single operand. The sum of the last two terms of (4.7) can be expressed as:

$$\left| (2^{n-r})(-x_2) + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{2^n-1} \right|_{2^n-1} = \left| (2^{n-r})(-x_2) + \frac{x_2 - |x_2|_{2^r}}{2^r} \right|_{2^n-1} \quad (4.8)$$

It can be easily proven that the multiplicative inverse of 2^r modulo 2^n-1 is 2^{n-r} , i.e., $|2^{-r}|_{2^n-1} = 2^{n-r}$. Therefore, (4.8) can be simplified to

$$\left| (2^{n-r})(-x_2) + (2^{n-r})(x_2 - |x_2|_{2^r}) \right|_{2^{n-1}} = \left| -|x_2|_{2^r} \cdot 2^{n-r} \right|_{2^{n-1}} \quad (4.9)$$

Hence,

$$y_i = \left| (2^{n-r})x_1 - |x_2|_{2^r} \cdot 2^{n-r} \right|_{2^{n-1}} \quad (4.10)$$

According to *Property 2.8*, $|x_2|_{2^r} = (x_2)_{r-1:0}$, and by applying *Properties 2.6* and *2.7* to the first and second terms of (4.10), respectively, (4.10) can be further simplified to

$$y_1 = |P_1 + P_2|_{2^{n-1}} \quad (4.11)$$

where

$$P_1 = |x_1 \cdot 2^{n-r}|_{2^{n-1}} = CLS_n(x_1, n-r) = CRS_n(x_1, r) \quad (4.12)$$

and

$$\begin{aligned} P_2 &= \left| -(x_2)_{r-1:0} \cdot 2^{n-r} \right|_{2^{n-1}} \\ &= CLS_n \left(\underbrace{1 \cdots 1}_{n-r} \parallel \overline{(x_2)_{r-1:0}}, n-r \right) = \overline{(x_2)_{r-1:0}} \parallel \underbrace{1 \cdots 1}_{n-r} \end{aligned} \quad (4.13)$$

where $a \parallel b$ denotes the concatenation of two binary strings, a and b , and CRS denotes the circular right shift operation.

From (4.3) and (4.5), the scaled residue output y_2 is expressed as follows:

$$y_2 = \left\| \frac{X}{2^r} \right\|_{m_2} = \left\| \frac{\left\lfloor \frac{X}{2^n} \right\rfloor 2^n + x_2}{2^r} \right\|_{m_2} \quad (4.14)$$

The $2n$ -bit vector $\left\lfloor \frac{X}{2^n} \right\rfloor$ can be calculated from the residues using (3.11) and is given by

$\left| (2^{2n-1} + 2^{n-1})x_1 - 2^n x_2 + (2^{2n-1} + 2^{n-1} - 1)x_3 \right|_{m_1 m_3}$. Thus, the numerator of (4.14) is a $3n$ -bit

binary vector obtained by the concatenation of the result of $\left\lfloor \frac{X}{2^n} \right\rfloor$ and x_2 . Division by 2^r ,

which is simply a right shift by r bits of this $3n$ -bit binary vector, produces the binary scaled output Y . The modulo 2^n of Y completes the scaled output y_2 of (4.14). By *Property 2.8*, y_2 can be directly tapped from the least significant n bits of Y with no additional computation.

The scaled residue digit y_3 is derived from (4.5) as follows:

$$y_3 = \left\lfloor \frac{X}{2^r} \right\rfloor_{m_3} = \left\lfloor \frac{X}{2^n} \right\rfloor_{m_3} \cdot |2^{n-r}|_{m_3} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_3} \quad (4.15)$$

By substituting $\left\lfloor \frac{X}{2^n} \right\rfloor_{m_3} = |x_2 + 2^n x_3|_{m_3}$ from Section 3.2.2 ((3.12) can also be derived directly from MRC and were embraced in some solutions of the reverse conversion problem [And88], [Moh02]) into (4.15) and using the property $|2^n|_{2^n+1} = -1$,

$$\begin{aligned} y_3 &= \left| |x_2 + 2^n x_3|_{m_3} \cdot |2^{n-r}|_{m_3} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_3} \right|_{m_3} \\ &= \left| |x_2 - x_3|_{m_3} \cdot |2^{n-r}|_{m_3} + \left\lfloor \frac{x_2}{2^r} \right\rfloor_{m_3} \right|_{m_3} \end{aligned} \quad (4.16)$$

It can be easily shown that the multiplicative inverse of 2^r modulo 2^n+1 is -2^{n-r} . Hence, $|2^{-r}|_{2^n+1} = -2^{n-r}$. Consequently, the first and last terms of (4.16) involving the residue x_2 can be combined and simplified as follows.

$$\begin{aligned} &\left| (2^{n-r})(x_2) + \left\lfloor \frac{x_2}{2^r} \right\rfloor \right|_{2^n+1} \\ &= \left| (2^{n-r})(x_2) + \frac{x_2 - |x_2|_{2^r}}{2^r} \right|_{2^n+1} \\ &= \left| (2^{n-r})(x_2) + (-2^{n-r})(x_2 - |x_2|_{2^r}) \right|_{2^n+1} \\ &= \left| (2^{n-r})(|x_2|_{2^r}) \right|_{2^n+1} \end{aligned} \quad (4.17)$$

The expression can be further simplified by *Property 2.8* to

$$\left| (2^{n-r})(x_2) + \left\lfloor \frac{x_2}{2^r} \right\rfloor \right|_{2^n+1} = |(x_2)_{r-1:0} \cdot 2^{n-r}|_{2^n+1} \quad (4.18)$$

The expression in (4.18) is an r -bit binary number shifted to the left by $n-r$ bits. The result is an n -bit number with zeros at the least significant $n-r$ bit positions. Since this number is smaller than 2^n+1 , the modulus operator is redundant and can be omitted.

$\left| -(2^{n-r})x_3 \right|_{m_3}$ can be expressed as $\left| MS + \overline{LS} + 2 \right|_{2^{n+1}}$, where MS and LS denote the most significant and least significant n bits of $x_3 2^{n-r}$, respectively. However, this implementation results in an irreducible four-operand addition for (4.16) and two stages of CSA with CEAC are required. As x_3 is $(n+1)$ bits wide, this simple expression is not valid for $r = 0$ when $(x_3)_n = '1'$ and multiplexers are needed to select x_3 when $r = 0$. The following derivations show that $\left| -x_3 2^{n-r} \right|_{2^{n+1}}$ for $0 < r \leq n$ can be re-expressed to enable y_3 to be computed with only one stage of n -bit CSA with CEAC and the same architecture can be used for the direct flow-through of x_3 when $r = 0$ without multiplexing.

For $0 < r \leq n$,

$$\begin{aligned} \left| -(2^{n-r})x_3 \right|_{2^{n+1}} &= \left| 2^{2n-r} x_3 \right|_{2^{n+1}} \\ &= \left| \left((x_3)_{n:r} \cdot 2^r + (x_3)_{r-1:0} \right) (2^{2n-r}) \right|_{2^{n+1}} \\ &= \left| (x_3)_{n:r} + (x_3)_{r-1:0} \cdot 2^{n+(n-r)} \right|_{2^{n+1}} \end{aligned} \quad (4.19)$$

By decomposing the binary word $(x_3)_{r-1:0}$ of (4.19) into its weighted bit variables and applying *Property 2.9* to each binary bit variable,

$$\left| -(2^{n-r})x_3 \right|_{2^{n+1}} = \left| (x_3)_{n:r} + \overline{(x_3)_{r-1:0}} \cdot 2^{n-r} + \sum_{i=1}^r 2^{2n-i} \right|_{2^{n+1}} \quad (4.20)$$

The last term of (4.20) can be simplified to

$$\left| \sum_{i=1}^r 2^{2n-i} \right|_{2^{n+1}} = \left| 2^{2n} - 2^{2n-r} \right|_{2^{n+1}} = \left| 1 + 2^{n-r} \right|_{2^{n+1}} \quad (4.21)$$

By substituting (4.18) and (4.20) into (4.16),

$$y_3 = \left| (x_3)_{n:r} + \overline{(x_3)_{r-1:0}} \cdot 2^{n-r} + (x_2)_{r-1:0} \cdot 2^{n-r} + (1 + 2^{n-r}) \right|_{2^{n+1}} \quad (4.22)$$

It is noted that $(x_3)_n$ is asserted in only one out of the 2^n combinations and when it is asserted, the remaining bits of x_3 are zeros. According to *Property 2.9*, the CEAC in each stage of CSA will introduce a constant bias of $\left| 2^n \right|_{2^{n+1}} = \left| -1 \right|_{2^{n+1}}$. By including this addition of $\left| -1 \right|_{2^{n+1}}$ into (4.22),

$$y_3 = \left| (x_3)_{n:r} + \overline{(x_3)_{r-1:0}} \cdot 2^{n-r} + (x_2)_{r-1:0} \cdot 2^{n-r} + 2^{n-r} \right|_{2^n+1} \quad (4.23)$$

This expression can be simplified by considering the cases of $x_3 = 2^n$ and $x_3 < 2^n$ separately.

When $x_3 = 2^n$, $(x_3)_n = '1'$ and $(x_3)_{n-1:0} = \underbrace{0 \cdots 0}_n$,

$$y_3 = \left| 2^{n-r} + \sum_{i=0}^{r-1} 2^{n-r+i} + (x_2)_{r-1:0} \cdot 2^{n-r} + 2^{n-r} \right|_{2^n+1} \quad (4.24)$$

Since $\left| 2^{n-r} + \sum_{i=0}^{r-1} 2^{n-r+i} \right|_{2^n+1} = |2^n|_{2^n+1} = |-1|_{2^n+1}$, the constant terms in (4.24) summed up to

$2^{n-r} - 1 = \sum_{i=0}^{n-r-1} 2^i$. Equation (4.24) can be reduced to

$$y_3 = (x_2)_{r-1:0} \cdot 2^{n-r} + \sum_{i=0}^{n-r-1} 2^i \quad (4.25)$$

which is a single binary vector $\underbrace{(x_2)_{r-1:0}}_r \parallel \underbrace{1 \cdots 1}_{n-r}$.

When $x_3 < 2^n$, $(x_3)_n = '0'$ and $(x_3)_i \in \{0, 1\} \forall i < n$,

$$y_3 = \left| (x_3)_{n-1:r} + \overline{(x_3)_{r-1:0}} \cdot 2^{n-r} + (x_2)_{r-1:0} \cdot 2^{n-r} + 2^{n-r} \right|_{2^n+1} \quad (4.26)$$

The sum of the first two terms, $(x_3)_{n-1:r} + \overline{(x_3)_{r-1:0}} \cdot 2^{n-r} = \underbrace{\overline{(x_3)_{r-1:0}}}_r \parallel \underbrace{(x_3)_{n-1:r}}_{n-r}$ is a single binary

vector. Due to the occurrence of $(x_2)_0$ and '1' at the same $(n-r)$ -th bit position in the last two terms, (4.26) is a sum of only three binary vectors.

For $r = 0$,

$$|y_3|_{2^n+1} = \left| (x_3)_{n-1:0} + \overline{(x_3)_n} + 2^n \right|_{2^n+1} \quad (4.27)$$

By including the bias of $|-1|_{2^n+1}$ introduced by the CSA with CEAC, y_3 can be expressed as

$$\left| (x_3)_{n-1:0} + \overline{(x_3)_n} + (2^n - 1) \right|_{2^n+1}.$$

When $x_3 = 2^n$,

$$y_3 = \underbrace{1 \cdots 1}_n \quad (4.28)$$

When $x_3 < 2^n$,

$$y_3 = \left| (x_3)_{n-1:0} + (2^n - 1) + 1 \right|_{2^n+1} \quad (4.29)$$

Let

$$\begin{aligned} Q_1 &= \overline{(x_3)_n} \wedge \left(\overline{(x_3)_{r-1:0}} \parallel (x_3)_{n-1:r} \right) \\ &= \overline{(x_3)_n} \wedge CCRS_n \left((x_3)_{n-1:0}, r \right) \\ Q_2 &= \underbrace{(x_2)_{r-1:0}}_r \parallel \underbrace{1 \cdots 1}_{n-r} \\ Q_3 &= \overline{(x_3)_n} \end{aligned} \quad (4.30)$$

where $a \wedge b$ denotes the logical *AND* of the binary variable a with every bit of the binary vector b . $CCRS_n(X, r)$ denotes a complementary circular right shift of an n -bit binary variable X by r bits. By feeding Q_1 , Q_2 , and Q_3 into a CSA with CEAC, irrespective of the values of r and $(x_3)_n$, it can be easily shown that (4.25), (4.26), (4.28) and (4.29) are simultaneously fulfilled by

$$y_3 = \left| S + C_{n-2:0} \parallel \overline{C_{n-1}} \right|_{2^n+1} \quad (4.31)$$

where S and C denotes the n -bit sum and carry vectors of the CSA with CEAC.

As an example, consider RNS $\{2^n-1, 2^n, 2^n+1\}$ with $n = 8$, i.e., $\{m_1, m_2, m_3\} = \{255, 256, 257\}$. The residue representation (x_1, x_2, x_3) of an integer $X = 10021985$ is $(230, 97, 13)$. Scaling X by 2, 4, 32 and 256 produces $Y = 5010992, 2505496, 313187$ and 39148. Table 4.1 shows the numerical computations of the scaled residues (y_1, y_2, y_3) for $r = 1, 2, 5, 8$ from (4.3), (4.11), (4.14) and (4.31). The calculated results are equal to the residue representations of 5010992, 2505496, 313187 and 39148, respectively.

The integer part of scaling result is computed accurately by the proposed programmable power-of-two scaling algorithm because the simplified equations of y_1 , y_2 and y_3 are derived

from the basic definition of power-of-two scaling operation in RNS without approximating or pruning any term by assumption. The exactness of the scaling results has been verified by exhaustive Matlab simulation of the proposed algorithm with the full range of input values.

Table 4.1: Scaling of (230, 97, 13) by 2, 4, 32 and 256 in RNS {255, 256, 257}

r	i	$Sum\ before\ modulo\ m_i,\ a$	$y_i = a _{m_i}$
1	1	$ 128 \times 230 _{255} - 97 _2 \times 128 = -13$	242
	2	$\lfloor \{32896 \times 230 - 256 \times 97 + 32895 \times 13\}_{65535} \times 256 + 97 \} / 2 \rfloor = 5010992$	48
	3	$(6+0 \times 128) + (1 \times 128) + (128) = 6+0^*$	6
2	1	$ 64 \times 230 _{255} - 97 _4 \times 64 = 121$	121
	2	$\lfloor \{32896 \times 230 - 256 \times 97 + 32895 \times 13\}_{65535} \times 256 + 97 \} / 4 \rfloor = 2505496$	24
	3	$(3+2 \times 64) + (1 \times 64) + (64) = 131+129^*$	3
5	1	$ 8 \times 230 _{255} - 97 _{32} \times 8 = 47$	47
	2	$\lfloor \{32896 \times 230 - 256 \times 97 + 32895 \times 13\}_{65535} \times 256 + 97 \} / 32 \rfloor = 313187$	99
	3	$(0+18 \times 8) + (1 \times 8) + (8) = 144+17^*$	161
8	1	$ 1 \times 230 _{255} - 97 _{256} \times 1 = 133$	133
	2	$\lfloor \{32896 \times 230 - 256 \times 97 + 32895 \times 13\}_{65535} \times 256 + 97 \} / 256 \rfloor = 39148$	236
	3	$(0+242 \times 1) + (97 \times 1) + (1) = 146+195^*$	84

*They are the sum (S) and carry (C) outputs of the CSA with CEAC. Note that the most significant carry bit is complemented and wrapped around to the LSB position.

4.3 Hardware Implementation

Figure 4.1 shows the proposed architecture for the implementation of the programmable power-of-two scaling algorithm presented in the previous section. Its derivation is detailed in this section.

The scaled residue y_1 can be generated by (4.11). P_1 can be generated by a cyclic left shift of the n -bit residue x_1 by $n-r$ bits or an equivalent CRS of x_1 by r bits. This can be implemented using an n -bit logarithmic cyclic right shifter (Shifter 1 in Figure 4.1) with x_1 as the input. The logarithmic cyclic shifter has $\lceil \log_2 n \rceil$ stages to cater for the maximum number of shifts n . Since the width of the cyclic shifter is also n , the outputs are the same for $r = 0$ and n . P_2 can be generated by shifting the one's complement of x_2 to the left by $s = n-r$ bits and padded with '1'. This can be implemented by an n -bit logarithmic left shifter (Shifter 2 of Figure 4.1). The logarithmic shifter has $\lceil \log_2 (n+1) \rceil$ stages. The free input is fed with logic '1' for the

rightmost 2^{h-1} multiplexers at the h -th stage. Each of these multiplexers with a constant input of '1' can be reduced to a two-input *OR* gate. Figure 4.2 and Figure 4.3 show the logarithmic shifters for the generation of P_1 and P_2 , respectively for $n = 8$. The outputs of the shifters are added by a modulo $2^n - 1$ adder to produce y_1 .

- Shifter 1: $\lceil \log_2 n \rceil$ -stage logarithmic cyclic right shifter of an n -bit word by r bits
Shifter 2: $\lceil \log_2(n+1) \rceil$ -stage logarithmic left shifter of an n -bit word by r bits
Shifter 3: $\lceil \log_2(n+1) \rceil$ -stage logarithmic logical right shifter of a $3n$ -bit word by r bits
Shifter 4: $\lceil \log_2(n+1) \rceil$ -stage logarithmic complementary cyclic right shifter of an n -bit word by r bits

Figure 4.1: Architecture of proposed RNS programmable power-of-two scaler based on moduli set $\{2^n-1, 2^n, 2^n+1\}$

The scaled residue y_2 is obtained from the r -th to the $(n+r-1)$ -th bit of the concatenation of $\left\lfloor \frac{X}{2^n} \right\rfloor$ and x_2 , as indicated by (4.14). A modulo $2^{2n} - 1$ adder with $2n$ -bit CSA with EAC and a $3n$ -bit shifter, as shown in Figure 4.1, are used to implement (4.14). Based on the definition of Y given in Section 4.2, Y is $3n-r$ bits wide. As the right shift amount r is programmable from 0 to n , Shifter 3 is a $3n$ -bit wide logical right shifter, where the r vacant bit positions are filled in with zeros. Therefore, its output is the binary representation of the scaled output Y . Although only the least significant n bits of Y are needed for y_2 , it is useful and requires no additional cost to keep the byproduct Y of this computation.

Figure 4.2: 8-bit 3-stage logarithmic cyclic right shifter for the generation of P_1

Figure 4.3: 8-bit 4-stage logarithmic left shifter for the generation of P_2

The scaled residue y_3 is produced by a CSA with CEAC followed by a mod 2^n+1 adder, with the input vectors of (4.30). The complementary circular shift of x_3 by r bits to the right can be implemented by an n -bit $\lceil \log_2(n+1) \rceil$ -stage logarithmic complementary cyclic right shifter (Shifter 4 in Figure 4.1), as shown in Figure 4.4 for $n = 8$. The output is gated by $\overline{(x_3)_n}$ with n two-input *AND* gates to produce Q_1 .

Q_2 is similar to P_2 of m_1 channel except that the input x_2 are not inverted. Hence it can be implemented by the same type of n -bit logarithmic left shifter shown in Figure 4.3 without the input inversion.

Figure 4.5 shows the CSA with CEAC for the addition of Q_1 , Q_2 and Q_3 . It consists of one FA and $n-1$ half adder (HA) to produce the n -bit sum and carry outputs. These two outputs are then summed using a modulo 2^n+1 adder to produce the scaled residue output y_3 .

Figure 4.4: 8-bit 4-stage logarithmic complementary cyclic right shifter (output AND with $\overline{(x_3)_8}$) for the generation of Q_1

Figure 4.5: n -bit CSA with CEAC of modulus 2^n+1 channel

Another merit of the proposed scaling algorithm and architecture is that it can be adapted to perform fixed scaling by 2^r , for any arbitrary integer r less than or equal to n . As dedicated fixed scaler, the shifters are no longer needed and the input operands that are fed into the modulo adder in the m_1 and m_3 channels can be obtained either directly or by bit re-ordering (by hardwiring without logic gate) and bit inversion (with only inverters) of the corresponding residue inputs. y_2 can be obtained by tapping the least significant n bits of the output Y of modulo $2^{2n}-1$ adder.

4.4 Performance Evaluation and Comparison

The hardware complexity and performance of the proposed RNS programmable power-of-two scaler are theoretically analyzed and synthesized for comparison in this section. As mentioned in Section 4.1, this is the first programmable scaler for $\{2^n-1, 2^n, 2^n+1\}$ RNS and the only reported variable RNS scaler [Car05b] is not applicable to this special moduli set and is not better than a hybrid system involving efficient RNS forward and reverse converters and programmable binary logarithmic power-of-two shifter. Therefore, the most efficient implementation of the latter programmable RNS scaler is compared.

4.4.1 Theoretical analysis based on unit-gate model

The hardware requirement and delay of the designs in comparison are estimated here using the unit-gate model. For the architecture shown in Figure 4.1, the area and delay of its constituent modules, such as different types of shifters, CSAs and modulo CPAs, are estimated as follows.

The area and delay of Shifter 1 are $2n\lceil\log_2 n\rceil$ units and $2\lceil\log_2 n\rceil$ units, respectively; the area and delay of Shifter 2 are $2n\lceil\log_2(n+1)\rceil + (1 - 2^{\lceil\log_2(n+1)\rceil})$ units and $2\lceil\log_2(n+1)\rceil$ units, respectively; the area and delay of Shifter 3 are $6n\lceil\log_2(n+1)\rceil$ units and $2\lceil\log_2(n+1)\rceil$ units, respectively; and the area and delay of Shifter 4 are $2n\lceil\log_2(n+1)\rceil + n$ units and $2\lceil\log_2(n+1)\rceil + 1$ units, respectively. There is no CSA in the m_1 channel. n OR gates and $2n$ FAs are required to implement the $2n$ -bit CSA with EAC in the m_2 channel (as detailed in Section 3.3). From Figure 4.5, $n-1$ HAs and one FA are required to implement the n -bit CSA with CEAC in the m_3 channel. The area and delay for the Ling modulo $2^n - 1$ adder are $3n\lceil\log_2 n - 1\rceil + 12n$ units and $2\lceil\log_2 n - 1\rceil + 3$ units, respectively, as estimated in Section 3.4. Based on the details given in Section 3.2 and Section 4 of [Efs04], the area and

time complexity of a modulo 2^n+1 adder are estimated to be $4.5n\lceil\log_2 n\rceil+3.5n+10$ units and $2\lceil\log_2 n\rceil+6$ units, respectively. The area and delay broken down into three categories of Shifter, CSA and CPA in each modulus channel are tabulated and totalled in Table 4.2 and Table 4.3, respectively.

Table 4.2: Unit-gate area analysis of proposed design

	m_1	m_2	m_3
shifter	$2n\lceil\log_2 n\rceil+2n\lceil\log_2(n+1)\rceil+(1-2^{\lceil\log_2(n+1)\rceil})$	$6n\lceil\log_2(n+1)\rceil$	$2n\lceil\log_2(n+1)\rceil+2n\lceil\log_2(n+1)\rceil+n+(1-2^{\lceil\log_2(n+1)\rceil})$
CSA	0	$15n$	$3n+4$
CPA	$3n\lceil\log_2 n-1\rceil+12n$	$6n\lceil\log_2 n\rceil+24n$	$4.5n\lceil\log_2 n\rceil+3.5n+10$
Total	$5n\lceil\log_2 n\rceil+2n\lceil\log_2(n+1)\rceil+9n+(1-2^{\lceil\log_2(n+1)\rceil})$	$6n\lceil\log_2(n+1)\rceil+6n\lceil\log_2 n\rceil+39n$	$4.5n\lceil\log_2 n\rceil+4n\lceil\log_2(n+1)\rceil+7.5n+14+(1-2^{\lceil\log_2(n+1)\rceil})$

Table 4.3: Unit-gate delay analysis of proposed design

	m_1	m_2	m_3
shifter	$2\lceil\log_2(n+1)\rceil$	$2\lceil\log_2(n+1)\rceil$	$2\lceil\log_2(n+1)\rceil+1$
CSA	0	5	4
CPA	$2\lceil\log_2 n-1\rceil+3$	$2\lceil\log_2 n\rceil+3$	$2\lceil\log_2 n\rceil+6$
Total	$2\lceil\log_2(n+1)\rceil+2\lceil\log_2 n-1\rceil+3$	$2\lceil\log_2(n+1)\rceil+2\lceil\log_2 n\rceil+8$	$2\lceil\log_2(n+1)\rceil+2\lceil\log_2 n\rceil+11$

The single-stage CSA (if present), shifter and modulo adder operate parallelly and independently in each modulus channel. As CSA has a constant delay and the latter two arithmetic modules have their delay logarithmically dependent on n , the critical path delay of the proposed variable power-of-two RNS scaler increases logarithmically with n .

The hybrid variable power-of-two scaler to be compared is implemented with an efficient residue-to-binary converter [Wan02], a programmable power-of-two binary shifter and an efficient binary-to-residue converter of the same three moduli set [She04]. The unit-gate areas and delays of the residue-to-binary converter, shifter and binary-to-residue converter used to implement the hybrid programmable power-of-two RNS scaler and its total area and

delay are tabulated in Table 4.4. The comparison of the unit-gate areas and delays between the proposed scaler and the hybrid design for $n = 5$ to 8 are shown in Table 4.5.

Table 4.4: Unit-gate-area and delay analysis of hybrid design

module	area	delay
residue-to-binary	$6n\lceil \log_2 n \rceil + 44n + 9$	$2\lceil \log_2 n \rceil + 11$
shifter	$6n\lceil \log_2(n+1) \rceil$	$2\lceil \log_2(n+1) \rceil$
binary-to-residue	$3n\lceil \log_2 n \rceil + 63n + 54$	$2\lceil \log_2(n+2) \rceil + 2\lceil \log_2 n \rceil + n + 16$
total	$9n\lceil \log_2 n \rceil + 6n\lceil \log_2(n+1) \rceil + 107n + 63$	$2\lceil \log_2(n+2) \rceil + 2\lceil \log_2(n+1) \rceil + 4\lceil \log_2 n \rceil + n + 27$

Table 4.5: Comparison of estimated unit-gate areas and delays

$\{2^n-1, 2^n, 2^n+1\}$	proposed		hybrid		% reduction	
	area	delay	area	delay	area	delay
$\{31, 32, 33\}$	690	23	823	56	16.2	58.9
$\{63, 64, 65\}$	828	23	975	57	15.1	59.6
$\{127, 128, 129\}$	966	23	1127	60	14.3	61.7
$\{255, 256, 257\}$	1184	25	1327	63	10.8	60.3

From Table 4.5, the proposed variable power-of-two RNS scaler design is considerably faster and smaller than the hybrid design, with a significant 58.9%, 59.6%, 61.7% and 60.3% delay reduction and a discernible 16.2%, 15.1%, 14.3% and 10.8% area savings for $n = 5, 6, 7$ and 8, respectively. Owing to the logarithmic dependence on n of the critical path delay of the proposed design (see Table 4.3) versus the linear dependence on n of the delay of the hybrid design (see Table 4.4), the percentage improvement in the computational speed of the proposed design is expected to increase steadily with n .

4.4.2 Synthesis Results

The proposed and hybrid variable scaler architectures are described in Verilog HDL and functionally verified using ModelSim. The designs are synthesized and mapped to TSMC 0.18 μ m standard cell library using Synopsys Design Compiler under worst case scenario, *i.e.*, 125°C and 1.62V. Each design is optimized for speed to obtain their minimum achievable

delay. The synthesized area in μm^2 and critical path delay in ns of these RNS scalers for $n = 5, 6, 7$ and 8 are shown in Table 4.6. As a reference for assessing the cost of programmability, the fastest and least cost constant scaler with only one fixed scaling factor of 2^n for the same moduli set presented in Chapter 3, hereafter known as 2^n scaler, is also synthesized and optimized under the same design flow and setup. The results are included in the last column of Table 4.6.

Table 4.6 clearly indicates that the proposed scaler is significantly faster and consumes less area than the hybrid design for all values of n . The reduction in critical path delay is more than 50%. The critical path delay of the proposed design is found to be relatively constant for the four RNSs of different dynamic ranges. This observation is consistent with the results of theoretical analysis given by the unit-gate delay in Table 4.5.

Table 4.6: Comparison of the synthesized areas and delays of $\{2^n - 1, 2^n, 2^n + 1\}$ RNS scalers for $n = 5, 6, 7$ and 8

$\{2^n - 1, 2^n, 2^n + 1\}$	Proposed		Hybrid		2^n Scaler	
	<i>area</i>	<i>delay</i>	<i>area</i>	<i>delay</i>	<i>area</i>	<i>delay</i>
$\{31, 32, 33\}$	18315.2	1.95	22380.0 (18.2)	4.08 (52.2)	14429.9 (-26.9)	1.43 (-36.4)
$\{63, 64, 65\}$	21425.3	2.01	24565.5 (12.8)	4.26 (52.8)	15055.3 (-42.3)	1.49 (-34.9)
$\{127, 128, 129\}$	24329.3	2.10	28627.0 (15.0)	4.48 (53.1)	18122.2 (-34.3)	1.57 (-33.8)
$\{255, 256, 257\}$	30749.2	2.23	34271.9 (10.3)	4.76 (53.2)	19126.8 (-60.8)	1.62 (-37.7)

In the hybrid design, even though the binary scaling operation can be performed very fast, it is sandwiched between the slower inter-modulo residue-to-binary and the binary-to-residue converters and cannot be easily merged with either stage even though the computations in each residue channel can be executed in parallel for the last stage. The proposed variable power-of-two scaling algorithm breaks this dependency to enable the shifting operation to be distributed into each modulus channel so that each residue can be independently scaled by any power-of-two factor. The resultant operation is as simple as a programmable multi-operand modulo addition. This also explains why the remarkable speed improvement of the proposed parallel design does not come with an area penalty despite having three more shifters than the sequential hybrid design implemented with very efficient forward and

reverse converters. Comparing with the hybrid design, the area of the proposed design has in fact been reduced by 18.2%, 12.8%, 15.0% and 10.3% for $n = 5, 6, 7$ and 8 respectively.

A programmable scaler is used when at least two different scaling constants are needed. For this reason, a programmable scaler is considered area efficient if it consumes less than twice the area of the most cost effective constant scaler for the same moduli set. Although the RNS scaler presented in Chapter 3 can only be used to scale an integer variable by a fixed factor of 2^n for $\{2^n-1, 2^n, 2^n+1\}$, let us conservatively assume that a constant scaler of any scaling factor for this moduli set can also be realized at the same cost. Then, r such units will be required to scale a variable by r different pre-selected factors, and approximately $\lceil \log_2 r \rceil$ stages of 2-to-1 multiplexers are needed to select the output from one of these constant scalers. For $r = 2$, without accounting for the area of the output selection multiplexers, the area savings of the design over two such units reduces from 36.5% for $\{31, 32, 33\}$ to 19.6% for $\{255, 256, 257\}$ due to the increased overheads for programmability with n . As the required number of different scaling constants r increases, approximately $r-1$ times the area of 2^n scaler will be saved by using the proposed programmable scaler. The timing overhead due to the programmable scaling factor of the proposed design remains around 35% of 2^n scaler for all four dynamic ranges. As the delay of the output selection multiplexers increases logarithmically with r , this timing overhead for programmability will diminish as r increases and vanish when $r = n$. Overall, it is advantageous to use the proposed programmable scaler than multiple dedicated constant scalers to fulfil the need for variable integer scaling in this RNS.

The power consumptions of the proposed and the hybrid design are simulated by PrimeTime PX using the same technology library. The total power consumption as well as the leakage power of both designs for $n = 5, 6, 7$ and 8 are simulated with 500 randomly generated and normally distributed input data. The data are fed at a frequency of approximately 210 MHz determined by the longest delay of all designs in Table 4.6, which is 4.76 ns. The power simulation is performed based on a Monte Carlo method [Bur93] with 95% confidence that the error is bounded below 2%. The simulated total power consumption and the leakage power are listed in Table 4.7. The results show that the proposed design dissipates lower power for all values of n . It consumes 44.6%, 51.5%, 37.8% and 41.3% lower total power and 22.4%, 20.8%, 24.4% and 14.6% lower leakage power than the hybrid design for $n = 5, 6, 7$ and 8 respectively.

7 and 8, respectively. Each design is also simulated for their total power and leakage power at its individual maximum achievable clock speed, and its energy consumption is estimated by the product of the simulated power and its critical path delay. The total energy and leakage energy (power-delay product) so computed are listed in Table 4.8. Similar percentage energy reduction of 44.6%, 51.5%, 37.8% and 41.3% for $n = 5, 6, 7$ and 8 , respectively were observed for the proposed design. It is also noted that the energy consumption due to leakage of the proposed design is much smaller than that of the hybrid design for every values of n . These considerable power reduction and energy reduction are a direct consequence of the simultaneous reduction of hardware area and timing complexity made possible by the proposed algorithm.

Table 4.7: Comparison of total power in mw and leakage power in μw @ 210MHz for $n = 5, 6, 7$ and 8

$\{2^n-1, 2^n, 2^{n+1}\}$	Proposed		Hybrid		%reduction	
	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>
$\{31, 32, 33\}$	4.01	0.83	7.24	1.07	44.6%	22.4%
$\{63, 64, 65\}$	4.57	0.95	9.43	1.20	51.5%	20.8%
$\{127, 128, 129\}$	5.59	1.02	8.99	1.35	37.8%	24.4%
$\{255, 256, 257\}$	6.61	1.34	11.27	1.57	41.3%	14.6%

Table 4.8: Comparison of total power $\times$ delay in pJ and leakage power $\times$ delay in fJ for $n = 5, 6, 7$ and 8

$\{2^n-1, 2^n, 2^{n+1}\}$	Proposed			Hybrid		
	<i>Total</i>	<i>leakage</i>	<i>%leakage</i>	<i>total</i>	<i>leakage</i>	<i>%leakage</i>
$\{31, 32, 33\}$	19.04	1.62	8.51	34.44	4.37	12.69
$\{63, 64, 65\}$	21.75	1.91	8.78	44.85	5.11	11.39
$\{127, 128, 129\}$	26.56	2.14	8.06	42.79	6.05	14.14
$\{255, 256, 257\}$	31.48	2.99	9.50	53.67	7.47	13.92

4.5 Conclusion

An elegant algorithm has been proposed to overcome the daunting problem of performing programmable power-of-two scaling operation directly in the celebrated three moduli set RNS $\{2^n-1, 2^n, 2^{n+1}\}$. Its simplicity is a phenomenal contribution to the simultaneous reduction of area, delay and power consumption in its hardware implementation. The synthesis results are very promising. Exceptional computation speed improvements of 52.2%,

52.8%, 53.1% and 53.2% for $n = 5, 6, 7$ and 8 , respectively, combined with an average area saving of 14.1%, over the hybrid design have been demonstrated. Besides, remarkable savings of total power consumption by 44.6%, 51.5%, 37.8% and 41.3%, and leakage power by 22.4%, 20.8%, 24.4% and 14.6% for $n = 5, 6, 7$ and 8 , respectively, have been achieved. The proposed programmable power-of-two scaler can efficiently scale an RNS number by up to one third of the dynamic range, which is very useful for many applications.

In the next chapter, the front end conversion problem of general moduli set will be investigated.

Chapter 5

A New Approach to the Design of Efficient Residue Generators for Arbitrary Moduli

5.1 Introduction

Besides the frequently cited advantages over the conventional binary system, recent studies indicate that RNS computations also offer significant delay tolerance against process-induced parameter variations with properly selected bases [Kou10], [Kou12]. It is found that larger modulo operation dominates circuit behavior and exhibits increased delay variations. From variation-tolerant perspective, RNS constructed by a large number of small and balanced moduli is preferable. This finding is timely and has paradigmatic impact on the digital integrated circuit design and yield for the continued scaling of transistor dimensions.

Although moduli of the forms 2^n and $2^n \pm 1$ are modulo arithmetic friendly, it is very difficult to obtain more than five coprime integers of comparable wordlength in these forms. To the best of the author's knowledge, the highest reported cardinality of such balanced moduli set is five [Cao07]. As the dynamic range increases, the wordlengths of some or all of these moduli will have to increase. Except for the simplicity of forward and reverse conversions in RNS, it may not pay off to have a large modulo operation of special modulus than multiple small generic modulo operations. The study in [Bha99] shows that the area, delay and power costs contributed from the reverse conversion are cardinality insensitive once the cardinality exceeds certain threshold (usually between five to eight). Hence, it is better to increase the cardinality rather than enlarging the sizes of one or more moduli to extend the dynamic range of an RNS. Fortunately, a valid RNS can be formed with relative ease by selecting as many moduli as desired from plentiful small integers without restriction to fulfil the relative primality criterion and meet the dynamic range requirement.

The significance of general moduli sets is also reflected in the continuous research into their efficient reverse conversion problem [Bi04], [Dim03], [Mas07]. Hardware implementation of forward converter, also known as the residue generators, is not trivial for general moduli set. Unlike the reverse converter, as many residue generators as the cardinality of the moduli set are needed for each integer operand, which can become a performance bottleneck. In addition to serving as the forward converter in RNS, efficient residue generator for modulus of any form is also a basic component instrumental to the implementation of many online residue-based arithmetic operations. In this chapter, a new approach to the design of highly efficient residue generators for any arbitrary moduli of up to six bits wide is proposed.

The rest of this chapter is organized as follows. The proposed architecture and its design procedures are presented in Section 5.2. Synthesis results are compared and analyzed in Section 5.3, and the conclusion is given in Section 5.4.

5.2 Proposed Residue Generators for Arbitrary Moduli

Our approach to the design of residue generators for arbitrary moduli is a great departure from all existing solutions. The main ideas that distinguish the proposed architecture are: (1) depth-bounded carry-save addition, (2) carry-free wordlength reduction of the sum and carry vectors and (3) a single modified modulo m adder. An overview of the essential building blocks is shown in Figure 5.1. For the ease of exposition, wherever necessary, the computations are elaborated using a running example of modulus $m = 29$, and a binary input X of length $L = 64$. This input wordlength is twice that considered in [Pre06] and is sufficiently large for most digital signal processing algorithms.

Figure 5.1: Block diagram of proposed residue generator

5.2.1 Partitioned Column Compression Of Partial Residue Bits

To avoid the great delay disparity of CSA due to the differences in periodicity of different moduli, the residue is calculated using the distributive property in modular arithmetic [Das08] as follows:

$$|X|_m = \left| \sum_{i=0}^{L-1} 2^i x_i \right|_m = \left| \sum_{i=0}^{L-1} |2^i|_m \cdot x_i \right|_m \quad (5.1)$$

where $x_i \in \{0, 1\}$.

As opposed to the input partitioning of conventional designs discussed in Section 2.6, the number of partial sums is L instead of $\lceil L/p \rceil$, which is independent of the modulus m . For convenience, each partial sum $|2^i|_m \cdot x_i$ is called a partial residue, where $i = 0, 1, \dots, L-1$. Each partial residue is r -bit wide, where $r = \lceil \log_2 m \rceil$ is the wordlength of the arbitrarily chosen modulus m , and $\lceil \cdot \rceil$ denotes the smallest integer greater than or equal to $(\cdot)$. The number of variable bits in each partial residue cx_i is determined by the Hamming weight $d(c)$ of its coefficient $c = |2^i|_m$, which can be pre-computed. Table 5.1 shows some partial residues and their decimal coefficients for $m = 29$, which has a cyclic periodicity of $p = 28$.

Table 5.1: Coefficients and partial residues of $X \bmod 29$

i	$c = 2^i _{29}$ (dec)	cx_i	$d(c)$
0	1	0 0 0 0 x_0	1
1	2	0 0 0 x_1 0	1
2	4	0 0 x_2 0 0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
13	14	0 x_{13} x_{13} x_{13} 0	3
14	28	x_{14} x_{14} x_{14} 0 0	3
15	27	x_{15} x_{15} 0 x_{15} x_{15}	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$
27	15	0 x_{27} x_{27} x_{27} x_{27}	4
28	1	0 0 0 0 x_{28}	1
29	2	0 0 0 x_{29} 0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$

In conventional approach, the array of partial sums is reduced to two binary vectors by a CSA tree followed by a CPA to produce a p -bit or longer word as input operand to the next stage. Thus subsequent modular additions also become unwieldy due to its input dependency on p . To minimize the hardware complexity of the modulo reduction process, the wordlengths of the sum and carry vectors resulted from the CSA tree need to be minimized so as to reduce the wordlength of the CPA and hence the number of modulo m adders needed later. To completely do away with the CPA and reduce the number of modulo m adders to just one irrespective of the modulus m , an alternative construction of CSA tree is proposed, leveraging on the fact that the width of the proposed initial partial residue array is the same as the wordlength of the final residue r as opposed to p . The former is the smallest among the two for all modulus m from 3 to 127 [Moh94]. Very often, r is much smaller than p .

A compact dot matrix diagram, where each dot denotes a binary variable, is formed by vacating all the '0' bits in the $L \times r$ matrix of partial residue bits. The height of the dot matrix can be reduced by adding every three (two) dots in the same column by a FA (HA) in a carry save manner iteratively until it is two. The number of columns will expand in this process as the carry bits overflow to column r and beyond, where column 0 is the rightmost column. To minimize the number of overflow bits, which will be used as address bits to a LUT to generate a modulo m reduced r -bit carry vector in the next step, the CSA tree is divided into two parts so as to reduce the sum vector separately to r bits without carry propagation. The dots in every column in the right partition (column 0 to column $r-1$) and the dots in every

column in the left partition (column r and beyond) are summed in parallel but with different column compression strategies.

As soon as possible (ASAP) column reduction strategy [Tow03] is adopted in the right partition to minimize the depth of the CSA tree and shorten the length of its final carry vector so that at the last stage of the ASAP column compression, at least three least significant bit columns contain only one dot. Let $h_j(l)$ be the number of dots in column j at level l of the CSA. The number of FAs, $n_{FA,j}(l)$ required in column j of level l is given by

$$n_{FA,j}(l) = \left\lfloor \frac{h_j(l)}{3} \right\rfloor \quad (5.2)$$

$n_{HA,j}(l)$ (which is either 0 or 1) HA is then used to sum any balance of two dots after all three-bit groups in column j of level l have been summed by $n_{FA,j}(l)$ FAs, i.e.,

$$n_{HA,j}(l) = \left\lfloor \frac{\left\lfloor h_j(l) \right\rfloor_3}{2} \right\rfloor \quad (5.3)$$

The remaining dot, i.e.,

$$n_{d,j}(l) = \left\| h_j(l) \right\|_3 \Big|_2 \quad (5.4)$$

is passed on to the next level.

The number of FAs and HAs can be determined by (5.2) and (5.3) for $l = 1$ from the initial dot matrix, but to calculate them for $l > 1$, the column heights need to be determined first. Each FA or HA in the current level introduces into the next level one sum bit in the current column and one carry bit in the next column. For $j = 0$ to $r-1$, the height of each column after the ASAP column compression can be computed by

$$\begin{aligned} h_j(l) &= n_{FA,j}(l-1) + n_{HA,j}(l-1) + n_{d,j}(l-1) \\ &\quad + n_{FA,j-1}(l-1) + n_{HA,j-1}(l-1) \\ &= \left\lfloor \frac{h_j(l-1)}{3} \right\rfloor + \left\lfloor \frac{\left\lfloor h_j(l-1) \right\rfloor_3}{2} \right\rfloor + \left\| h_j(l-1) \right\|_3 \Big|_2 \\ &\quad + \left\lfloor \frac{h_{j-1}(l-1)}{3} \right\rfloor + \left\lfloor \frac{\left\lfloor h_{j-1}(l-1) \right\rfloor_3}{2} \right\rfloor \end{aligned} \quad (5.5)$$

To reduce the size of LUT, the total number of dots must be reduced as much as possible in the left partition. As the number of input and output bits are identical for a HA, only carry-save FAs are used for the column compression in the left partition. The number of FAs can still be computed by (5.2) but the number of free dots becomes $|h_j(l)|_3$. Therefore, the column height for $j > r$ can be computed by:

$$h_j(l) = \left\lfloor \frac{h_j(l-1)}{3} \right\rfloor + |h_j(l-1)|_3 + \left\lfloor \frac{h_{j-1}(l-1)}{3} \right\rfloor \quad (5.6)$$

For $j = r$, the carry from the HA, if any, in column $r-1$ of level $l-1$ will add to this column. Thus,

$$h_r(l) = \left\lfloor \frac{h_r(l-1)}{3} \right\rfloor + |h_r(l-1)|_3 + \left\lfloor \frac{h_{r-1}(l-1)}{3} \right\rfloor + \left\lfloor \frac{|h_{r-1}(l-1)|_3}{2} \right\rfloor \quad (5.7)$$

From (5.2) and (5.3), the number of FAs and HAs as well as the depth of the CSA tree can be minimized by reducing the heights of as many columns of the initial dot matrix as possible. This can be achieved by minimizing the Hamming weights of the partial residue coefficients using the following congruence.

Property 5.1: Let $c (< m)$ be an integer constant and $x \in \{0,1\}$ be a binary variable, then

$$|cx|_m = |c + c'x|_m \quad (5.8)$$

where $c' = m - c$.

Property 5.1 can be proved by $|cx|_m = |c(1 - \bar{x})|_m = |c - c\bar{x}|_m = |c + (m - c)\bar{x}|_m$.

As an example, $|2^{15}x_{15}|_{29} = |27x_{15}|_{29} = |27 + 2\bar{x}_{15}|_{29}$. The coefficient of x_{15} has changed from 27 to 2, causing its Hamming weight to be reduced by 3 after the transformation. *Property 5.1* is applied to each partial residue cx_i with Hamming weight $d(c) > 1$ and cx_i is replaced by $c'\bar{x}_i$ if $d(c) > d(c')$. Each replacement of cx_i to $c'\bar{x}_i$ generates a constant c . These known constants can be sum modulo m into a single constant vector and its '1' bits are inserted into the dot matrix. The dot matrix diagram of the partial residue array of $X \bmod 29$ after the

Hamming weight reduction by *Property 5.1* is illustrated in level 1 of the CSA tree of Figure 5.2. The two ‘1’ bits in the constant vector are added into columns 1 and 3.

Figure 5.2: Dot matrix diagram of the CSA for the partial residue array of $X \bmod 29$ after Hamming weight reduction

The heights $h_j(1)$ of every column of the dot matrix based on $L = 64$ for modulus up to six bits wide before and after the Hamming weight reduction are tabulated in Table 5.2. For each modulus, the total numbers of FAs and HAs required in the right partition until the height of the right partition reaches two based on the ASAP column compression scheme are also indicated. The moduli shown in Table 5.2 are odd integers. Any even modulus that is not a power-of-two can always be expressed as a power-of-two multiple of an odd modulus m , i.e.,

$2^k \times m$, where k is a positive integer. Its residue can be computed directly from the residue $|X|_m$ by adding to it the k least significant bits of X based on *Theorem 1* of [Pre06]. Special moduli of forms 2^n or $2^n \pm 1$ are also not shown as their residue generators are trivial. From Table 5.2, a significant amount of FAs can be saved from the CSA tree due to the column height reduction. If the cost of a HA is considered to be half of a FA, then the savings can be as high as 39.8% for $m = 43$ and 20.9% on average. Similar table can be easily generated for any other input wordlength based on (5.2)-(5.7). For $L = 32$, the savings was found to be 36.4% for $m = 43$ and 18.9% on average.

Table 5.2: Columns height of the partial residue array for different modulus before and after Hamming weight reduction ($L = 64$)

m	p	Before Hamming wgt reduction								After Hamming wgt reduction							
		column j						#FA	#HA	column j						#FA	#HA
		5	4	3	2	1	0			5	4	3	2	1	0		
11	10	-	-	19	25	31	31	79	7	-	-	14	25	19	20	56	6
13	12	-	-	26	26	31	31	79	7	-	-	17	22	31	16	61	8
19	18	-	11	28	28	31	31	103	11	-	7	21	28	18	19	74	3
21	6	-	10	21	11	21	21	63	10	-	10	21	12	21	12	56	10
23	11	-	12	23	22	23	24	81	9	-	12	24	16	29	19	77	9
25	20	-	24	22	28	31	31	105	11	-	12	20	23	32	17	81	13
27	18	-	27	25	28	30	30	107	10	-	18	18	28	15	22	77	6
29	28	-	27	28	29	31	30	111	11	-	17	21	25	32	18	87	11
35	12	5	25	21	21	26	26	102	10	5	20	21	21	21	11	79	13
37	36	10	28	27	29	31	33	130	12	6	20	24	22	32	16	98	12
39	12	5	20	21	21	21	21	86	11	6	20	21	16	16	26	84	9
41	20	18	21	19	28	19	31	106	10	10	18	13	20	19	10	70	7
43	14	22	18	22	18	30	30	109	8	9	18	9	18	14	16	65	6
45	12	15	25	11	21	31	26	104	11	15	20	11	21	31	16	89	15
47	23	15	22	24	25	25	25	110	8	15	22	21	23	16	23	95	9
49	21	21	21	30	33	30	31	133	14	9	21	22	22	30	20	101	11
51	8	8	16	24	16	16	16	75	9	8	16	24	16	16	16	75	9
53	52	25	24	27	28	31	30	131	11	15	21	24	22	31	19	106	9
55	20	18	24	25	25	25	25	113	10	13	22	25	22	23	17	97	13
57	18	26	30	28	25	21	29	125	8	10	18	22	25	21	12	86	9
59	58	28	28	28	29	30	30	136	14	19	21	23	30	20	19	103	12
61	60	29	29	30	30	31	31	140	18	19	20	23	26	31	19	109	14
Avg. FA cost (FA+0.5HA)								111.05		Avg. FA cost						87.86	

Of all moduli that are less than 64, the worst case is $m = 29$, which has the largest column height of 32 at $j = 1$ and also the second largest height at its adjacent column, i.e., 25 dots at $j = 2$. The application of the proposed column compression scheme to the left and right

partitions (demarcated by a vertical dash line) of the Hamming weight reduced dot matrix for this modulus is shown in Figure 5.2. It can be concluded that for all modulus less than 7 bits wide, at most 7 CSA levels are required to reduce the height of the dot matrix in the right partition to two.

5.2.2 Elimination of Carry Propagation Addition

In conventional designs, CPA is used after the CSA to generate a p -bit or longer word to be modulo reduced by a tree of modulo m adders. In this section, it is shown that the binary vectors generated in the left and right partitions after the partitioned column compression of Section 5.2.1 can be further reduced to two r -bit vectors without carry propagation.

As mentioned in the previous section, the use of ASAP column reduction strategy on the right hand side of the partial residue array leads to at least three least significant bit columns containing only one dot. This means that at most $r-3$ columns need to be further reduced to an r -bit vector in the right partition. This can be done fast and cost effectively by carry-save half addition (CSHA) since r is small. For a modulus of up to 6 bits, at most three levels of CSHA and 6 HAs are needed. The CSHA reduction of the right partition after the ASAP CSA for modulus 29 is shown in Figure 5.3. Only one stage of CSHA with one HA is needed to obtain the r -bit sum (henceforth denoted by A) in the right partition.

Figure 5.3: Reduction of CSFA (left partition) and CSHA (right partition) to two r -bit operands for $m = 29$

The carries in the left partition can also be modulo reduced to an r -bit word B by:

$$B = \left\lfloor \sum_{j=0}^{k-1} 2^{r+j} y_i \right\rfloor_m \quad (5.9)$$

where k is the maximum number of columns in the left partition and $y_i \in \{0,1\}$ is a binary variable. $i = 1, 2, \dots, i_{max}$ where i_{max} is the total number of dots at the final level of the left partition. B can be computed using a LUT of i_{max} inputs. To reduce i_{max} , carry-save full addition (CSFA) is continued to reduce the overflow carries in the left partition simultaneously with the CSHA in the right partition. As illustrated by the example of modulus 29 in Figure 5.3, B can be computed by $|2^8 y_7 + 2^7 (y_6 + y_5) + 2^6 y_4 + 2^5 (y_3 + y_2 + y_1)|_{29}$ using a 7-input LUT after continuing the CSFA for one level (i.e., level 8 of the CSA). If the CSFA is allowed to continue for up to $r-3$ levels, then B can be computed by $|2^8 y_6 + 2^7 (y_5 + y_4) + 2^6 (y_3 + y_2) + 2^5 y_1|_{29}$ using a 6-input LUT with 9 levels of CSA.

The number of inputs of the LUT is bounded by the maximum number of carry bits that can be generated from the right partition of ASAP CSA and CSHA. Equations (5.5) to (5.7) can be used to work out the total number of carries eventually generated in the left partition. An exhaustive computation for all moduli listed in Table 5.2 shows that the total number of CSA levels is limited to 9 and the maximum number of input bits of the LUT is 7, except for $m = 59$ and 61. For $m = 59$ and 61, an 8-input LUT is required. The complete column height reduction process for the latter case is illustrated in Table 5.3, where the final numbers of dots left in every column of the left and right partitions are printed in bold. To reduce the size of the LUT, 2-bit CLAs without input carry are used in place of the CSHAs and CSFAs at the last level of the CSA to reduce two pairs of dots in adjacent columns to one dot in each of the two columns and a carry into the next column. The logic structure of a 2-bit CLA with no carry input is shown in Figure 5.4(a). For the example of Table 5.3, the dot matrix diagram of applying two 2-bit CLAs at the last level of the CSA is shown in Figure 5.4(b). The two pairs of consecutive dots in $h_6 h_5$ and $h_8 h_7$ are each summed by a 2-bit CLA of Figure 5.4(a) and the output of each 2-bit CLA is represented by three connected dots. As the critical path delay of a 2-bit CLA is the same as that of a FA, the number of inputs of the LUT is reduced from 8 to 6 for both $m = 59$ and 61 without compromising the delay of the CSA. 2-bit CLAs are also used to reduce the size of the LUT when the column heights are of the form "...222|2...", where | denotes the line of demarcation of the left and right partitions.

Table 5.3: Column heights of left and right partitions for $m = 61$

l	Left (CSFA)				Right (ASAP CSA and CSHA)						
	$h_9(l)$	$h_8(l)$	$h_7(l)$	$h_6(l)$	$h_5(l)$	$h_4(l)$	$h_3(l)$	$h_2(l)$	$h_1(l)$	$h_0(l)$	
1	0	0	0	0	19	20	23	26	31	19	ASAP CSA
2	0	0	0	6	14	15	17	19	17	7	
3	0	0	2	7	10	11	12	13	8	3	
4	0	0	4	6	8	8	8	8	4	1	
5	0	1	4	5	6	6	6	4	2	1	
6	0	2	3	5	4	4	3	3	1	1	
7	0	3	2	4	3	3	2	1	1	1	
8	1	1	3	3	2	2	1	1	1	1	CSHA
9	1	2	2	2	2	1	1	1	1	1	
	1	2	2	3	1	1	1	1	1	1

(a)

(b)

Figure 5.4: (a) 2-bit CLA without carry input (b) Reduction of LUT size by 2-bit CLAs at level 9 of CSA for $m = 61$

5.2.3 Modified Modulo m Adder

The final result of $|X|_m$ is obtained by adding the two r -bit operands A and B modulo m . It should be noted, however, that the maximum values of operands A and B are $2^r - 1$ and $m - 1$, respectively. By definition of $r = \lceil \log_2 m \rceil$, $2^{r-1} < m \leq 2^r$. The maximum value of $A + B = m +$

$2^r - 2 = m + 2 \times 2^{r-1} - 2$ can exceed $2m$ for $m > 2$. Let $e = A + B - 2m$ be the sum of A and B in excess of $2m$. Then

$$\begin{aligned} e &\leq 2^r - 1 + m - 1 - 2m \\ &= 2^r - (m + 2) \\ &< m \end{aligned} \tag{5.10}$$

Thus, the modulo m addition of A and B can be fulfilled by:

$$|X|_m = |A + B|_m = \begin{cases} A + B - 2m, & \text{if } A + B \geq 2m \\ A + B - m, & \text{if } m \leq A + B < 2m \\ A + B, & \text{otherwise} \end{cases} \tag{5.11}$$

A direct implementation of (5.11) requires three r -bit carry lookahead (CLA) binary adders, a r -bit 3-1 multiplexer and two logic gates for its selector [Bay87]. Alternatively, the three additions can also be performed serially with one r -bit binary adder, two feedback $(r+1)$ -bit registers, three r -bit 3-1 multiplexers and two logic gates [Dug92]. More efficient implementation of modular addition could be found in [Ver05], which utilizes a carry save adder stage and two binary adders operating in parallel. Since A , B and $|X|_m$ are all r bits wide, by adapting the same concept of [Hia02], (5.11) can be rewritten as

$$|X|_m = \begin{cases} |A + B + 2Z|_{2^r}, & \text{if } A + B + 2Z \geq 2^{r+1} \\ |A + B + Z|_{2^r}, & \text{if } 2^r \leq A + B + Z < 2^{r+1} \\ A + B, & \text{otherwise} \end{cases} \tag{5.12}$$

where $Z = 2^r - m$.

The block diagram of the modified modulo 29 adder is shown in Figure 5.5. It has five basic units as in [Hia02], which are the Sum-And-Carry (SAC) unit, Carry Propagate and Generate (CPG) unit, CLA unit, multiplexer (MUX) unit, and CLA and Summation (CLAS) unit, except that there are two CLAs instead of one and the MUX unit consists of five 3-1 instead of 2-1 multiplexers.

Figure 5.5: Block diagram of the modulo 29 adder modified from [Hia02]

The fundamental components that make up these units are briefly described in order to explain the modifications made to the basic design of [Hia02]. The SAC unit generates the sum and carry vectors of $A+B$, $A+B+Z$ and $A+B+2Z$, respectively. The bit slices for their generation can be shared using two different types of cell. Since the i -bit of $2Z$ is the same as the $(i-1)$ -th bit of Z , a HA cell is used to generate s_i and c_i of A_i+B_i if z_i and z_{i-1} are 0. A half-adder-like (HAL) cell is used to generate both s_i and c_i of A_i+B_i if z_i and z_{i-1} are 0 as well as s'_i and c'_i of A_i+B_i+1 if z_i or z_{i-1} is 1. For $m = 29$, the values of Z and $2Z$ are 3 ("00011") and 6 ("00110"), respectively. The sum and carry bits of $A+B$, $A+B+Z$ and $A+B+2Z$ can therefore be realized with two HA cells for the two leading bits of which Z and $2Z$ are '0' and three HAL cells for the remaining bits of which Z and $2Z$ are either '0' or '1'. In Figure 5.6(a), the middle HAL is shaded to highlight that a different cell is used from the SAC unit of [Hia02]. The CPG consists of only HAs, which are used to generate the propagate and generate bits, P_i and G_i from s_i and c_{i-1} or p_i and g_i from s'_i and c'_{i-1} . Figure 5.6(b) shows the six HA cells from the CPG unit of [Hia02], and three additional HA cells (shaded) required for the calculation of the extra generate and propagate signals of $A+B+2Z$. The CLA used for the calculation of C_{out} of $A+B+Z$ is shown on the left of Figure 5.6(c), which is identical to that of [Hia02]. The additional CLA (shaded) for the computation of the output carry C_{out}^* for $A+B+2Z$ is shown on the right of Figure 5.6(c). The 3-1 MUX unit uses C_{out} and C_{out}^* to select the generate and propagate vectors from either P and G for $X+Y$, p and g for $X+Y+Z$ or p' and g' for $X+Y+2Z$. Similar to [Hia02], the CLAS unit can be implemented by any regular

CLA binary adder using the $2r$ generate and propagate bits received from the output of the MUX unit.

Figure 5.6: (a) The bit slices of SAC unit, (b) basic cells for CPG unit, and (c) gate-level structures for the CLAs that generate C_{out} and C_{out}^*

The flow chart in Figure 5.7 summarizes the above design steps for the proposed residue generator for any input word length L of X and arbitrary modulus m .

5.3 Performance Analysis and Comparison

5.3.1 Hardware Resource Analysis of Proposed Design Method for $L = 64$

The hardware resources required by the proposed residue generator for 64-bit integer and any value of m listed in Table 5.2 can be assessed from the total numbers of FAs and HAs, the sizes of the LUTs, and the logic gate counts of the 2-bit CLAs if used and the modified modulo m adder. Table 5.4 shows the derivation of the total numbers of FAs and HAs from

the numbers of FAs and HAs required by the left and right partitions in Section 5.2.1 and Section 5.2.2, and the number of inputs of the LUT in Section 5.2.2. In Table 5.4, ‘#2-bit CLA’ is the number of 2-bit CLAs used in the last level of the CSA and l_{max} is the maximum level of the CSA. The size of the LUT is given by $2^{l_{max}} \times r$ bits.

Figure 5.7: Proposed design approach to modulus m residue generator

Table 5.4: Hardware resource requirement of proposed residue generators before the final mod m adder for different m ($L = 64$)

m	r	l_{max}	Left	Right		#2-bit CLA	LUT	Total	
			#FA	#FA	#HA		i_{max}	#FA	#HA
11	4	8	12	56	7	0	6	68	7
13	4	8	15	61	9	0	6	76	9
19	5	7	8	74	3	0	6	82	3
21	5	6	8	56	10	2	5	64	10
23	5	9	12	77	10	0	6	89	10
25	5	9	12	81	14	0	6	93	14
27	5	9	13	77	7	0	6	90	7
29	5	9	15	87	12	0	6	102	12
35	6	8	8	79	14	0	6	87	14
37	6	9	9	98	15	0	7	107	15
39	6	8	8	84	14	2	5	92	14
41	6	8	8	70	10	0	6	78	10
43	6	9	8	65	9	0	5	73	9
45	6	9	12	89	18	0	7	101	18
47	6	9	12	95	12	0	7	107	12
49	6	9	10	101	14	0	7	111	14
51	6	9	8	75	11	2	5	83	11
53	6	9	13	106	12	0	7	119	12
55	6	9	12	97	16	0	7	109	16
57	6	9	10	86	10	0	6	96	10
59	6	8	15	103	14	2	6	118	14
61	6	8	15	109	16	2	6	124	16

5.3.2 Comparison with Published Results of Parallel Residue Generators for $L = 32$

The most recent comparison of parallel residue generators was reported in [Pre06]. The residue generator of [Pre06] is an improved version of [Pre02]. Its modular exponentiation is performed on p bits instead of L bits. The former requires p and the latter L r -bit $(q-1)$ -to-1 multiplexers with varying amount of logic gates for their forward translation functions, where $q = \lceil \log_2 L \rceil$ is the number of bits in the exponents. Unfortunately, the structures of modulo m adders were not described in [Pre06] and [Pre02]. It was only stated in [Pre02] that each of the L modulo m adders was realized using two r -bit RCAs of $2r$ FAs and one r -bit 2-1 multiplexer. The resource requirement is close to the adder scheme of [Bay87] except that the

OR gate for the multiplexer select input has been omitted. In [Pre06], it was only stated that each modulo m adder was realized using one r -bit RCA, r D flip flops (FFs), two r -bit 2-1 multiplexers and r inverters. To the best of the author's knowledge, the least cost implementation of a modulo m adder that uses only one RCA with approximately 10 FA delay is achievable with the adder scheme of [Dug92]. However, this scheme requires one r -bit 2-1 multiplexer, one FF and one OR gate more than what was reported in [Pre06]. The additional multiplexer is required for selecting the output from between the outputs of register and RCA, the FF for storing the carry signal and the OR gate for generating the multiplexer select signal. If the final modulo m adder of the proposed design is also realized using RCA and the same scheme as [Dug92], then the hardware resources of these parallel residue generators for $m = 19$ and $L = 32$ can be compared in Table 5.5 with the data excerpted from [Pre06] except that five 2-1 multiplexers, one FF and one basic logic gate have been added to each modulo m adder in the modular adder trees reported for all three parallel methods in Table II of [Pre06] for the reason mentioned above.

Table 5.5: Comparison between [Pre06], [Pre02], [Moh94] for $m = 19$ and $L = 32$

Method	[Pre06]	[Pre02]	[Moh94]	Proposed
No. of FAs	85	155	85	63
No. of HAs	0	0	0	11
No. of D FFs	102	186	102	12
ROM (bits)	0	0	19×5	16×5
No. of 3-1 MUX	0	0	0	15
No. of 2-1 MUX	525	935	85	0
No. of basic logic gates	149	163	1547	2

Based on the same timing analysis of [Pre06], the modulo m adder delay for the proposed design is approximately $3 \times 5 = 15$ FF delay. It is at least three times faster than the modular adder tree of the other three designs in comparison, which all have timing of $5 \times 2 \times 5 = 50$ FF delay due to the five levels of modular adder tree. The proposed design requires 7 FA delay for the CSA, CSHA and CSFA and a 7-bit LUT lookup time before the modified modulo m adder, but this is still much shorter than the delay for two levels of two-input multiplexers required by the modulo exponentiation of [Pre06] plus the delay of two CPAs (36 FAs if implemented by RCA) needed to add the two partitioned 18-bit words and their wrapped back carry for [Pre06] before the modulo exponentiation, which have been omitted in Table II of [Pre06] as well as Table 5.5.

5.3.3 Comparison of Synthesis Results with Memory-Free and Memory-Based Residue Generators for Arbitrary Moduli of Diverse Periodicities

In this section, the proposed residue generator is compared against the state-of-the-art memory-free parallel residue generator of [Pre06] with this thesis author's proposed improvements, [Moh94], and the most competitive ROM-based residue generator [Sto94] (which was not compared in [Pre06]) for six moduli $m = 19, 21, 29, 37, 53$ and 61 with periodicity $p = 18, 6, 28, 36, 52$ and 60 , respectively, and an input wordlength of 64 bits. Two different ways of partitioning the input operand are described in [Sto94]: (a) the binary input is partitioned into 16 words of 4 bits each, and (b) the binary input is partitioned into 8 words of 8 bits each. The method of [Pre06] partitions the input operand into $\lceil 64/p \rceil$ words of p bits each. For $p = 6, 18$ and 28 , CSA followed by CPAs are used to reduce the result to one p -bit word. The binary CPA is implemented using the fast parallel prefix adder [Lad80] and each modulo m adder in its modular adder tree is also implemented with the fastest modulo m adder detailed in [Hia02]. The author of this thesis has also further improved the residue generator presented in [Pre06] by removing the unnecessary modular exponentiation stage and mapping each output bit of the binary adder directly to the modular adder tree using the distributive property of modulo summation discussed in Section 5.2. Consequently, all the logic gates for the forward translation function and all the multiplexers required for the modular exponentiation can be eliminated completely. These improvements have not only sped up the original design of [Pre06] tremendously, but also reduced its area significantly. Similarly for [Moh94], each output bit of the binary adder is directly mapped to the modular adder tree using the distributive property, and hence, removing the ROMs and the multiplexers that select the input values to the modular adder tree.

All designs in comparison are described in Verilog HDL and functionally verified using ModelSim. The designs are synthesized and mapped to TSMC65nm standard cells using Synopsys Design Compiler under the worst case scenario, i.e., 125°C and 0.9V . The synthesized area in μm^2 and critical path delay in ns are shown in Table 5.6.

Table 5.6: Area (μm^2) and delay (ns) comparison of different residue generators (number in parenthesis indicates percentage reduction of proposed design over other design in comparison)

m	Proposed		Improved [Pre06]		Improved [Moh94]		[Sto94] (a)		[Sto94] (b)	
	area	delay	area	delay	area	delay	area	delay	area	delay
19	3181	1.65	3876 (17.9)	2.41 (31.5)	4450 (28.5)	2.58 (36.0)	6216 (48.8)	1.87 (11.8)	8699 (63.4)	1.68 (1.8)
21	2694	1.44	2630 (-2.4)	1.44 (0)	2592 (-3.9)	1.44 (0)	5550 (51.5)	1.85 (22.2)	7168 (62.4)	1.61 (10.6)
29	4340	1.55	5133 (15.4)	2.31 (32.9)	4690 (7.5)	2.19 (29.2)	5412 (19.8)	1.84 (15.8)	10171 (57.3)	1.58 (1.9)
37	3572	1.79	7846 (54.5)	2.94 (39.1)	7846 (54.5)	2.94 (39.1)	7438 (52.0)	2.25 (20.4)	10798 (66.9)	1.83 (2.2)
53	4560	1.75	7618 (40.1)	2.84 (38.4)	7618 (40.1)	2.84 (38.4)	6978 (34.7)	2.16 (19.0)	11896 (61.7)	1.83 (4.4)
61	4270	1.73	7998 (46.6)	2.28 (24.1)	7998 (46.6)	2.28 (24.1)	6818 (37.4)	1.93 (10.4)	10359 (58.8)	1.73 (0)

The results show that the proposed residue generator outperforms all designs in comparison. The proposed design is on average 27.7%, 27.8%, 16.6% and 3.5% faster than [Pre06], [Moh94], [Sto94] (a) and [Sto94] (b), respectively. The depth of the modular adder tree structure of [Pre06] and [Moh94], as well as the number of modular adders increase rapidly with p , causing the area reduction of the proposed design over the improved version of [Pre06] and [Moh94] to rise from -2.4% to 54.5% and -3.9% to 54.5% as p increases from 6 to 60. [Sto94] (b) is the most area intensive design because eight 8-bit LUTs are used to produce the partial sums to be added by the modular adder tree. On average, the proposed design reduces the area of [Sto94] (a) and [Sto94] (b) by 40.7% and 61.8%, respectively.

The power consumptions of the proposed and all other designs are simulated by PrimeTime PX using the same technology library. The total power consumption as well as the leakage power of all designs for $m = 19, 21, 29, 37, 53$ and 61 are simulated with 500 randomly generated and normally distributed input data. They are fed at a frequency of approximately $340MHz$ determined by the longest delay of all designs in Table 5.6, which is $2.94ns$. The power simulation is performed based on a Monte Carlo method [Bur93] with 95% confidence that the error is bounded below 1.5%. The simulated total power consumption and leakage power in μW are listed in Table 5.7. Overall, the results show that the proposed design dissipates the least power among the methods in comparison. It consumes on average 44.5%,

48.2%, 50.2% and 53.2% lower total power and 24.7%, 24.3%, 37.7% and 56.3% lower leakage power than [Pre06], [Moh94], [Sto94] (a) and [Sto94] (b), respectively.

Table 5.7: Total power (μW) and Leakage power (μW) comparison of different residue generators (number in parenthesis indicates percentage reduction of proposed design over other design in comparison)

m	Proposed		Improved [Pre06]		Improved [Moh94]		[Sto94] (a)		[Sto94] (b)	
	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>	<i>total</i>	<i>leakage</i>
19	1884	48.39	2817 (33.1)	54.77 (11.6)	3543 (46.8)	63.24 (23.5)	3878 (51.4)	89.15 (45.7)	3840 (50.9)	111.4 (56.6)
21	1435	40.19	1438 (0.2)	40.03 (-0.4)	1587 (9.6)	38.65 (-4.0)	3717 (61.4)	80.68 (50.2)	3472 (58.7)	96.65 (58.4)
29	2216	66.47	3615 (38.7)	72.80 (8.7)	3561 (37.8)	64.92 (-2.4)	3583 (38.2)	77.21 (13.9)	4247 (47.8)	136.7 (51.4)
37	2187	52.01	7106 (69.2)	108.5 (52.1)	7106 (69.2)	108.5 (52.1)	5091 (57.0)	103.1 (49.6)	5036 (56.6)	139.7 (62.8)
53	2328	67.27	7018 (66.8)	102.3 (34.2)	7018 (66.8)	102.3 (34.2)	4672 (50.2)	97.89 (31.3)	5182 (55.1)	156.8 (57.1)
61	2350	64.13	5698 (58.8)	110.7 (42.1)	5698 (58.8)	110.7 (42.1)	4111 (42.8)	99.62 (35.6)	4707 (50.1)	132.8 (51.7)

5.4 Conclusion

This chapter presents a new approach to the design of efficient residue generators for arbitrary moduli. The proposed design requires at most 7-stages of CSA, one LUT of no more than 7 inputs and a small modular adder for input wordlength as large as 64 bits and modulus of up to 6 bits wide, making it the fastest, smallest and most power efficient residue generator for any input and modulus within these ranges. It has significantly reduced the area of the fastest memory-based design and the timing of the most efficient memoryless design reported thus far. Moreover, the disparity due to inconsistent periodicity of different moduli of the latter design has been minimized by the proposed depth-constrained CSA tree and periodicity independent LUT and modified modular adder. The latters are made possible by the ingenious use of distributive property in place of periodicity property to limit the width of the CSA tree so that large size LUTs and modular adder tree can be eliminated. From the synthesis results of six moduli of diverse periodicities, the proposed design is found to be on average 27.7%, 27.8%, 16.6% and 3.5% faster than the improved version of [Pre06] and

[Moh94], [Sto94] (a) and [Sto94] (b), respectively with a considerable average area savings of 28.7%, 28.9%, 40.7% and 61.8%, average total power reduction of 44.5%, 48.2%, 50.2% and 53.2% and average leakage power reduction of 24.7%, 24.3%, 37.7% and 56.3%, respectively.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis focuses on solving the research problems of fixed and programmable scaling of the celebrated special moduli set, as well as the binary-to-residue conversion of general moduli sets, which have not been solved satisfactorily by the RNS community. With these objectives in mind, the main proposals made in this thesis are summarized here.

A novel area-efficient, high-speed and precise RNS scaler for the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ has been presented in this thesis. The new scaling algorithm is formulated based on the Chinese Remainder Theorem. The number theoretic properties of this special moduli set in relation to the scaling factor of 2^n have been exploited to eliminate the inter-modulus dependency of RNS scaling operation. The proposed scaling algorithm has successfully reduced the complicated inter-modulo operation into an architecture that is as simple as a multi-operand addition or constant multiplication in RNS.

Adaptive scaling, whereby the scaling factor is determined during runtime, is a useful technique often used in digital signal processing. An elegant algorithm for performing programmable power-of-two scaling operation directly in the three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ has been proposed for the first time. The proposed variable power-of-two scaling algorithm circumvents the inefficient method of performing variable power-of-two scaling in a sequential manner, which involves slow and costly reverse converter. In the proposed method, the shifting operation has been distributed into each modulus channel so that each scaled residue digit can be generated independently. The resultant architecture is as simple as a programmable multi-operand modulo addition, leading to tremendous speed improvement as

well as substantial hardware area reduction in comparison with the conventional design. The proposed programmable power-of-two scaler can efficiently scale an RNS number by up to one third of the dynamic range, which is very useful for many applications.

Furthermore, a new approach to the design of efficient residue generators for arbitrary moduli has been presented. The distributive property has been leveraged ingeniously in place of periodicity property to limit the width of the CSA and CPA. As a result, there is no need for multiple large size LUTs and modulo adder tree. The proposed design requires at most seven stages of CSA, one LUT of no more than seven-bit input and a small modular adder for input wordlength as large as 64 bits and modulus of up to six bits wide, making it the fastest, smallest and most power efficient residue generator for any input and modulus within these ranges. Moreover, the disparity due to inconsistent periodicity of different moduli of the latter design has been minimized by the proposed depth-constrained CSA tree and periodicity independent LUT.

6.2 Future Work

Based on the research results presented in this thesis, the following relevant topics are identified for future research.

1. The proposed programmable power-of-two scaling algorithm has scaling factor only up to one-third of the dynamic range. Larger scaling factor, preferably the complete dynamic range, will be very useful for the implementation of floating-point RNS digital signal processor. To accomplish the full range of scaling operation, additional control logic circuits are needed to enable the use of the proposed programmable scaler. The output of the proposed programmable scaler can be feedback in subsequent clock cycles to further scale the intermediate results. In this case, multiplexers and registers to select the appropriate inputs, and control logic to determine the selection are needed. Besides, more than one clock cycles are needed to complete the full range of scaling operation. Alternatively, a straightforward implementation will involve three such programmable scalers such that the input is passed through three scaling blocks in one clock cycle to produce the final scaled output. This is not an acceptable solution due to the long propagation time, and excessive hardware requirement

and may not be better than the hybrid solution. Consequently, a more elegant solution is required to develop an efficient RNS scaler that can cover the full range of variable scaling operation.

2. In this thesis, efficient RNS scalers, whether fixed or programmable, have been proposed for the most established three-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The dynamic range of the moduli set, which is of $3n$ bits, may not be sufficient to meet the frequency of operations required by some applications that have a large dynamic range. Lately, many high cardinality special moduli sets of conjugate moduli besides those of the forms 2^n and $2^n \pm 1$ have been proposed along with their corresponding reverse converters [Ska99a], [Ska99b], [Ska09]. However, to truly benefit from the use of these high cardinality RNS in implementing DSP applications, such as DWT, FFT, filter, etc., it is essential to solve the fundamental scaling problem in these RNS domains directly. In the absence of true RNS scaler, hybrid RNS in which the scaling operation is carried out in binary domain will be the only resort. The hybrid solution will definitely be worse for higher cardinality moduli set than for the three moduli set. Efficient algorithms and architectures for the inter-modulo scaling operation for high cardinality RNS is thus required to extend the usefulness of these new moduli sets. New CRT -I and -II could be exploited together with the good theoretic properties of these special moduli sets to devise the efficient architecture for multiple fixed scaling constants.

3. In binary number system, the result of a multiplication is often truncated to reduce the complexity of the subsequent operations in the datapath. The truncation is accomplished, for the simplest scheme, by tapping only the higher order bits to reduce the hardware cost and power consumption without compromising the speed of operation. However, it is impossible to do so in RNS without abandoning one or more modulus channels. The RNS scalers discussed thus far scale the value of an integer by keeping the dynamic range of the RNS system, and hence the complexity of the arithmetic operations remain unchanged. It is suggested to fulfill the envisioned truncated arithmetic operation in RNS by scaling the value of an integer and the dynamic range of the RNS simultaneously by reducing the size of individual modulus without discarding any modulus. In this way, the wordlengths of the modulo arithmetic operations in each modulus channel will not have to be uniform and confined by the fixed dynamic range of the chosen RNS throughout the IPSP architecture.

Significant hardware cost and power consumption can be saved particularly for some filtering and image processing applications where exact precision of the output is often not required.

Author's Publications

Journal Publications

[Low13] J. Y. S. Low and C. H. Chang, "A new approach to the design of efficient residue generators for arbitrary moduli," *IEEE Trans. Cir. Syst.-I: Regular Papers* (accepted for publication).

[Low12] J. Y. S. Low and C. H. Chang, "A VLSI efficient programmable power-of-two scaler for $\{2^n-1, 2^n, 2^n+1\}$ RNS," *IEEE Trans. Cir. Syst.-I: Regular Papers*, vol. 59, no. 12, pp. 2911-2919, Dec. 2012.

[Tay12] T. F. Tay, C. H. Chang, and 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 Integration (VLSI) Syst.* (in press, DOI 10.1109/TVLSI.2010.2221752).

[Cha11] C. H. Chang and 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. Cir. Syst.-I: Regular Papers*, vol. 58, no. 11, pp. 2686-2697, Nov. 2011.

Conference Publications

[Low13] J. Y. S. Low and C. H. Chang (Invited paper), "A survey on scaling in Residue Number System," in *Proc. The 8th Intelligent Living Technology Conference (ILT 2013) NCUT*, Taichung, Taiwan, Jun. 2013 (accepted for publication).

[Low13] J. Y. S. Low, T. F. Tay, and C. H. Chang, "A signed integer programmable power-of-two scaler for $\{2^n-1, 2^n, 2^n+1\}$ RNS," in *IEEE Int. Symp. Cir. Syst. (ISCAS)*, 2013, Beijing, China, May 2013 (accepted for publication).

[Low12] J. Y. S. Low, T. F. Tay, and C. H. Chang, "A unified $\{2^n-1, 2^n, 2^n+1\}$ RNS scaler with dual scaling constants," in *Proc. 2012 IEEE Asia Pacific Conf. Cir. Syst. (APCCAS)*, 2012, Taiwan, China, Dec. 2012, pp. 296-299.

[Low11] J. Y. S. Low and C. H. Chang, "A new RNS scaler for $\{2^n-1, 2^n, 2^n+1\}$," in *Proc. 2011 IEEE Int. Symp. Cir. Syst. (ISCAS)*, 2011, Rio de Janeiro, Brazil, May 2011, pp. 1431-1434.

[Low10] B. Cao, J. Y. S. Low, C. H. Chang, and T. Srikanthan, "Performance analysis of different special moduli sets for RNS-based inner product step processor," in *Proc. 2010 Int. Conf. Green Cir. Syst. (ICGCS)*, 2010, Shanghai, China, Jun. 2010, pp. 236-241.

References

- [Abd95] M. Abdallah and A. Skavantzios, "A systematic approach for selecting practical moduli sets for Residue Number Systems," in *Proc. 27th Southeastern Symp. Syst. Theory*, Mississippi, USA, Mar. 1995, pp. 445-449.
- [Ali84] G. Alia and E. Martinelli, "VLSI algorithm for direct and reverse conversion from weighted binary number to residue number system," *IEEE Trans. Cir. Syst.*, vol. 31, no. 12, pp. 1033-1039, Dec. 1984.
- [And88] S. Andraros and H. Ahmad, "A new efficient memory-less residue to binary converter," *IEEE Trans. Cir. Syst.*, vol. 35, no. 11, pp. 1441-1444, Nov. 1988.
- [Bak09] D. Bakalis and H. T. Vergos, "Shifter circuits for 2^n+1 , 2^n , 2^n-1 RNS," *Electronics Letters*, vol. 45, no. 1, pp. 27-29, Jan. 2009.
- [Bar95] F. Barsi and M.C. Pinotti, "Fast base extension and precise scaling in RNS for look-up table implementations," *IEEE Trans. on Signal Processing*, vol. 43, no. 10, pp. 2427-2430, Oct. 1995.
- [Bay87] M. Bayoumi and G. Jullien, "A VLSI implementation of residue adders," *IEEE Trans. Cir. Syst.*, vol. 34, no. 3, pp. 284-288, Mar. 1987.
- [Ber07] G. L. Bernocchi, G. C. Cardarili, A. Nannarelli and M. Re, "Low power adaptive filter based on RNS components," in *Proc. 2007 IEEE Int. Symp. Cir. Syst.*, New Orleans, USA, May 2007, pp. 3211-3214.
- [Bha98] M. Bharadwaj, A. B. Premkumar, and T. Srikanthan, "Breaking the $2n$ -bit carry propagation barrier in residue to binary conversion for the $\{2^n-1, 2^n, 2^n+1\}$ moduli set," *IEEE Trans. Cir. Syst. I*, vol. 45, no. 9, pp. 998-1002, Sep. 1998.

- [Bha99] M. Bhardwaj, T. Srikanthan, and C. T. Clark, "VLSI costs of arithmetic parallelism: a residue reverse conversion perspective," in *Proc. 14th IEEE Symp. Computer Arithmetic*, Adelaide, Australia, Apr. 1999, pp. 176-184.
- [Bi88] G. Bi and E. V. Jones, "Fast conversion between binary and residue numbers," *Electronic Letters*, vol. 24, no. 19, pp. 1195-1197, Sep. 1988.
- [Bi04] S. Bi, W. Wang, and A. Al-Khalili, "New modulo decomposed residue-to-binary algorithm for general moduli sets," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing, 2004*, Quebec City, Canada, May 2004, pp. 141-144.
- [Bur93] R. Burch, F. N. Najm, P. Yang, and T. N. Trick, "A Monte Carlo approach for power estimation," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 1, no. 1, pp. 63-71, Mar. 1993.
- [Bur03] N. Burgess, "Scaling an RNS number using the core function," in *Proc. 16th IEEE Symp. Computer Arithmetic*, Santiago de Compostela, Spain, Jun. 2003, pp. 262-269.
- [Cao03] B. Cao, C. H. Chang and 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. Cir. Syst.-I*, vol. 50, no. 10, pp. 1296-1303, Oct. 2003.
- [Cao05] B. Cao, T. Srikanthan and C. H. Chang, "Efficient reverse converters for the four-moduli sets $\{2^n-1, 2^n, 2^{n+1}, 2^{n+1}-1\}$ and $\{2^n-1, 2^n, 2^{n+1}, 2^{n-1}-1\}$," *IEE Proc. Computers and Digital Tech.*, vol. 152, no. 5, pp. 687-696, Sep. 2005.
- [Cao07] B. Cao, C. H. Chang, and T. Srikanthan, "A residue-to-binary converter for a new 5-moduli set," *IEEE Trans. Cir. Syst. I*, vol. 54, no. 5, pp. 1041-1049, May 2007.
- [Cap88] R. M. Capocelli and R. Giancarlo, "Efficient VLSI networks for converting an integer from binary system to Residue Number System and vice versa," *IEEE Trans. Cir. Syst.*, vol. 35, no. 11, pp. 1425-1430, Nov. 1988.

- [Car00] G. C. Cardarilli, A. Nannarelli, and M. Re, "Reducing power dissipation in FIR Filters using the residue number system," in *Proc. of the 43rd IEEE Midwest Symp. Cir. Syst.*, 2000, pp. 320-323.
- [Car05a] G. C. Cardarilli, A. D. Re, A. Nannarelli, and M. Re, "Low power and low leakage implementation of RNS FIR filters," in *Proc. 39th Asilomar Conf. Signals, Syst. Computer*, 2005, pp. 1620-1624.
- [Car05b] G. C. Cardarilli, A. Del Re, A. Nannarelli, and M. Re, "Programmable power-of-two RNS scaler and its application to a QRNS polyphase filter," in *Proc. IEEE Int. Symp. Cir. Syst.*, Kobe, Japan, May 2005, pp. 1102-1105.
- [Car07] G. C. Cardarilli, A. Nannarelli, and M. Re, "Residue number system for low-power DSP applications," in *Proc. 41st Asilomar Conf. Signals, Syst. Computers*, Pacific Grove, CA, USA, Nov. 2007, pp. 1412-1416.
- [Cha04] C. H. Chang, J. Gu, and M. Zhang, "Ultra low voltage, low power CMOS 4-2 and 5-2 compressors for fast arithmetic circuits," *IEEE Trans. Cir. Syst.-I*, vol. 51, no. 10, pp. 1985-1997, Oct. 2004.
- [Cha05] C. H. Chang, J. Gu, and M. Zhang, "A review of 0.18 μ m full adder performances for tree structured arithmetic circuits," *IEEE Trans. VLSI Syst.*, vol. 13, no. 6, pp. 686-695, Jun. 2005.
- [Cha11] C. H. Chang and 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. Cir. Syst. I*, vol. 58, no. 11, pp. 2686-2697, Nov. 2011.
- [Con99] R. Conway and J. Nelson, "Fast converter for 3 moduli RNS using new property of CRT," *IEEE Trans. Computers*, vol. 48, no. 8, pp. 852-860, Aug. 1999.
- [Con04] R. Conway and J. Nelson, "Improved RNS FIR filter architectures," *IEEE Trans. on Cir. Syst. II*, vol. 51, no. 1, pp. 26-28, Jan. 2004.

- [Das08] M. Dasygenis, K. Mitroglou, D. Soudris, and A. Thanailakis, "A full-adder-based methodology for the design of scaling operation in Residue Number System," *IEEE Trans. Cir. Syst. I, Reg. Papers*, vol. 55, no. 2, pp. 546-558, Mar. 2008.
- [Dhu98] A. Dhurkadas, "Comments on "A high-speed realization of a residue to binary number system converter",," *IEEE Trans. Cir. Syst. II*, vol. 45, no. 3, pp. 446-447, Mar. 1998.
- [Dim03] G. Dimauro, S. Impedovo, R. Modugno, G. Pirlo, and R. Stefanelli, "Residue-to-binary conversion by the "quotient function",," *IEEE Trans. Cir. Syst. II, Analog and Digit. Signal Processing*, vol. 50, no. 8, pp. 488-493, Aug. 2003.
- [Dim05] G. Dimitrakopoulos, D. G. Nikolos, H. T. Vergos, D. Nikolos, and C. Efstathiou, "New architectures for modulo 2^n-1 adders," in *Proc. IEEE Int. Conf. Electronics, Cir., Syst.*, Gammarth, Tunisia, Dec. 2005, pp. 1-4.
- [Dug92] M. Dugdale, "VLSI implementation of residue adders based on binary adders," *IEEE Trans. Cir. Syst. II, Analog and Digit. Signal Processing*, vol. 39, no. 5, pp. 325-329, May 1992.
- [Efs94] C. Efstathiou, D. Nikolos, and J. Kalamatianos, "Area-time efficient modulo 2^n-1 adder design," *IEEE Trans. Cir. Syst. II*, vol. 41, no. 7, pp. 463-467, Jul. 1994.
- [Efs03] C. Efstathiou, H. T. Vergos, and D. Nikolos, "Modulo $2^n\pm 1$ adder design using select-prefix blocks," *IEEE Trans. Computers*, vol. 52, no. 11, pp. 1399-1406, Nov. 2003.
- [Efs04] C. Efstathiou, H. T. Vergos, and D. Nikolos, "Fast parallel prefix modulo 2^n+1 adders," *IEEE Trans. Computers*, vol. 53, no. 9, pp. 1211-1216, Sep. 2004.
- [Gar98] A. Garcia and A. Lloris, "RNS scaling based on pipelined multipliers for prime moduli," in *Proc. IEEE Workshop on Signal Processing Syst.*, Piscataway, New Jersey, USA, Oct. 1998, pp. 459-468.

- [Gar99] A. Garcia and A. Lloris, "A look-up scheme for scaling in the RNS," *IEEE Trans. Computers*, vol. 48, no. 7, pp. 748-751, Jul. 1999.
- [Gri88] M. Griffin, F. Taylor, and M. Sousa, "New scaling algorithm for the Chinese Remainder Theorem," in *Proc. Conf. Rec. Twenty-Second Asilomar Conf. Signals, Syst. Computers*, 1988, pp. 375-378.
- [Gri89] M. Griffin, M. Sousa, and F. Taylor, "Efficient scaling in the residue number system," in *Proc. 1989 Int. Conf. Acoustics, Speech Signal Processing*, Glasgow, Scotland, UK, May 1989, pp. 1075-1078.
- [Hia02] A. A. Hiasat, "High-speed and reduced-area modular adder structures for RNS," *IEEE Trans. Computers*, vol. 51, no. 1, pp. 84-89, Jan. 2002.
- [Kon09] K. Yinan and B. Phillips, "Fast scaling in the residue number system," *IEEE Trans. VLSI Syst.*, vol. 17, no. 3, pp. 443-447, Mar. 2009.
- [Kou10] I. Kouretas and V. Paliouras, "Residue arithmetic bases for reducing delay variation," in *Proc. IEEE Int. Symp. Cir. Syst.*, Paris, France, Jun. 2010, pp. 3885-3888.
- [Kou12] I. Kouretas and V. Paliouras, "Residue arithmetic for designing multiply-add units in the presence of non-gaussian variation," in *Proc. IEEE Int. Symp. Cir. Syst.*, Seoul, Korea, May 2012, pp. 1231-1234.
- [Lad80] R. E. Ladner and M. J. Fisher, "Parallel Prefix Computation," *J. of the ACM*, vol. 27, no. 4, pp. 831-838, Oct. 1980.
- [Lim99] K. P. Lim and A. B. Premkumar, "A modular approach to the computation of convolution sum using distributed arithmetic principles," *IEEE Trans. Cir. Syst. II*, vol. 46, no. 1, pp. 92-96, Jan. 1999.
- [Lim07] Z. Lim and B. J. Phillips, "An RNS-enhanced microprocessor implementation of public key cryptography," in *Proc. 41st Asilomar Conf. Signals, Syst. Computers*, Pacific Grove, CA, USA, Nov. 2007, pp. 1430-1434.

- [Lin07] S. H. Lin, M. H. Sheu, K. H. Wang, J. J. Zhu, and S. Y. Chen, "Efficient VLSI design of modulo 2^n+1 adder using hybrid carry selection," in *Proc. IEEE Workshop on Signal Processing Syst., SiPS: Design and Implementation*, Shanghai, China, Oct. 2007, pp. 141-145.
- [Lot10] M. M. Lotfinejad, M. Mosleh, and H. Noori, "A novel high speed residue to binary converter design based on the three-moduli set $\{2^n, 2^{n+1}+1, 2^{n+1}-1\}$," in *Proc. East-West Design & Test Symp.*, St. Petersburg, Russia, Sep. 2010, pp. 524-526.
- [Ma10] S. Ma, J. Hu, Y. Ye, L. Zhang, and X. Ling, "A 2^n scaling scheme for signed RNS integers and its VLSI implementation," *Science China, Information Sciences*, vol. 53, no. 1, pp. 203-212, Jan. 2010.
- [Mah00] M. N. Mahesh and M. Mehendale, "Low power realization of residue number system based FIR filters," in *Proc. of 13th Int. Conf. VLSI Design*, Jan. 2000, pp. 30-33.
- [Mas07] O. Maslennikow, N. Maslennikowa, M. Rajewska, D. Gretkowski, and J. Lienou, "Design of FPGA-based Residue Number System converters for digital signal processing systems," in *Proc. 9th Int. Conf. Experience of Designing Applications of CAD Syst. in Microelectronics*, Lviv, Ukraine, Feb. 2007, pp. 194-201.
- [Men06] S. Menon and C. H. Chang, "A reconfigurable multi-modulus modulo multiplier," in *Proc. IEEE Asia Pacific Conf. Cir. Syst.*, Singapore, Dec. 2006, pp. 1168-1171.
- [Mey03] U. Meyer-Base and T. Stouraitis, "New power-of-2 RNS scaling scheme for cell-based IC design," *IEEE Trans. VLSI Syst.*, vol. 11, no. 2, pp. 280-283, Apr. 2003.
- [Moh94] P. V. A. Mohan, "Novel design for binary to RNS converters," in *Proc. IEEE Int. Symp. Cir. Syst.*, Bangalore, India, Jun. 1994, pp. 357-360.
- [Moh98] P. V. A. Mohan, "Evaluation of fast conversion techniques for binary-residue number systems," *IEEE Trans. Cir. Syst. I*, vol. 45, no. 10, pp. 1107-1109, Oct. 1998.

- [Moh99] P. V. A. Mohan, "Efficient design of binary to RNS converters," *J. of Cir., Syst., Comput.*, vol. 9, no. 3-4, pp. 145-154, Jun & Aug. 1999.
- [Moh02] P. V. A. Mohan, *Residue Number Systems: Algorithms and Architectures*. Kluwer Academic Publishers, New York, 2002.
- [Moh07a] 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. Cir. Syst. II*, vol. 54, no. 9, pp. 775-779, Sep. 2007.
- [Moh07b] P. V. A. Mohan and A. B. Premkumar, "RNS-to-binary converter for two four-moduli sets $\{2^n-1, 2^n, 2^{n+1}, 2^{n+1}-1\}$ and $\{2^n-1, 2^n, 2^{n+1}, 2^{n+1}+1\}$," *IEEE Trans. Cir. Syst. I*, vol. 54, no. 6, pp. 1245-1254, Jun. 2007.
- [Moh08] P. V. A. Mohan, "New residue to binary converters for the moduli set $\{2^k, 2^{k-1}, 2^{k-1}-1\}$," in *Proc. IEEE Region 10 Conf.*, Hyderabad, India, Nov. 2008, pp. 1-6.
- [Mur08] R. Muralidharan, C. H. Chang, and C. C. Jong, "A low complexity modulo 2^n+1 squarer design," in *Proc. IEEE Asia Pacific Conf. Cir. Syst.*, Macao, China, Dec. 2008, pp. 1296-1299.
- [Mur11] R. Muralidharan and C. H. Chang, "Radix-8 Booth encoded modulo 2^n-1 multipliers with adaptive delay for high dynamic range residue number system," *IEEE Trans. Cir. Syst. I*, vol. 58, no. 5, pp. 982-993, May 2011.
- [Mur12] R. Muralidharan and 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. Cir. Syst. I*, 2012 (in press).
- [Nan01] A. Nannarelli, M. Re, and G. C. Cardarilli, "Tradeoffs between residue number system and traditional FIR filters," in *Proc. 2001 IEEE Int. Symp. Cir. and Syst. (ISCAS2001)*, Sydney, Australia, May 2001, pp. 305-308.

- [Nav11] K. Navi, A. S. Molahosseini, and M. Esmaeildoust, "How to teach residue number system to computer scientists and engineers," *IEEE Trans. Education*, vol. 54, no.1, pp. 156-163, 2011.
- [Omo07] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*. London: Imperial College Press, 2007.
- [Par10] B. Parhami, *Computer Arithmetic : Algorithms and Hardware Designs*. New York: Oxford University Press, 2010, ch. 17.
- [Phi03] B. Phillips, "Scaling and reduction in the residue number system with pairs of conjugate moduli," in *Proc. 37th Asilomar Conf. Signals, Syst. Computers*, Pacific Grove, CA, USA, Nov. 2003, pp. 2247-2251.
- [Pie94] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Comput.*, vol. 43, no. pp. 68-77, Jan. 1994.
- [Pie95] S. J. Piestrak, "A high-speed realization of a residue to binary number system converter," *IEEE Trans. Cir. Syst. II*, vol. 42, no. 10, pp. 661-663, Oct. 1995.
- [Pie02] S. J. Piestrak, "Design of squarers modulo A with low-level pipelining," *IEEE Trans. Cir. Syst. – II*, vol. 49, no. 1, pp. 31-41, Jan. 2002.
- [Pou94] F. Pourbigharaz and H. M. Yassine, "Modulo-free architecture for binary to residue transformation with respect to $(2^m-1, 2^m, 2^m+1)$ moduli set," in *Proc. IEEE Int. Symp. Cir. Syst. (ISCAS94)*, London, May 1994, pp. 317-320.
- [Pre02] A. B. Premkumar, "A formal framework for conversion from binary to residue numbers," *IEEE Trans. Cir. Syst. II, Analog and Digit. Signal Processing*, vol. 49, no. 2, pp. 135-144, Feb. 2002.
- [Pre06] A. B. Premkumar, E. L. Ang, and E. M.-K. Lai, "Improved memoryless RNS forward converter based on the periodicity of residues," *IEEE Trans. Cir. Syst. II, Express Briefs*, vol. 53, no. 2, pp. 133-137, Feb. 2006.

- [Re01] A. D. Re, A. Nannarelli, and M. Re, "Implementation of digital filters in carry-save residue number system," in *Proc. Conf. Rec. of the Asilomar Conf. Signals, Syst. Computers*, Pacific Grove, California, USA, Nov. 2001, pp. 1309-1313.
- [Sha85] A. Shah, M. Sid-Ahmed, and G. Jullien, "A proposed hardware structure for two-dimensional recursive digital filters using the residue number system," *IEEE Trans. Cir. and Syst. I*, vol. 32, no. 3, pp. 285-288, Mar. 1985.
- [Sha02] A. Shams, T. Darwish, and M. Bayoumi, "Performance analysis of low-power 1-bit CMOS full adder cells," *IEEE Trans. VLSI Syst.*, vol. 10, no. 1, pp. 20-29, Feb. 2002.
- [She89] M. A. P. Shenoy and R. Kumaresan, "A fast and accurate RNS scaling technique for high speed signal processing," *IEEE Trans. Acoustics, Speech Signal Processing*, vol. 37, no. 6, pp. 929-937, Jun. 1989.
- [She04] M.-H. Sheu, S.-H. Lin, Y.-T. Chen, and Y.-C. Chang, "High-speed and reduced-area RNS forward converter based on $(2^n-1, 2^n, 2^n+1)$ moduli set," in *Proc. 2004 IEEE Asia-Pacific Conf. Cir. Syst.*, Tayi, Paraguay, Dec. 2004, pp. 821-824.
- [She12] R. Shende, P. Zode, and P. Zode, "Efficient design 2^k-1 binary to residue converter," in *Proc. Int. Conf. Devices, Cir. Syst.*, Tamilnadu, India, Mar. 2012, pp. 482-285.
- [Ska99a] A. Skavantzios and M. Abdallah, "Implementation issues of the two-level residue number system with pairs of conjugate moduli," *IEEE Trans. Signal Processing*, vol. 47, no. 3, pp. 826-838, Mar. 1999.
- [Ska99b] A. Skavantzios and Y. Wang, "New efficient RNS-to-weighted decoders for conjugate-pair-moduli residue number systems," in *Proc. 33rd Asilomar Conf. Signals, Syst., Comput.*, 1999, vol. 2, Pacific Grove, CA, USA, Oct. 1999, pp. 1345-1350.

- [Ska09] A. Skavantzios, M. Abdallah, T. Stouraitis, and D. Shinianakis, "Design of a balanced 8-modulus RNS," in *Proc. IEEE 16th Int. Conf. Electronics, Cir. Syst., 2009*, Yasmine Hammamet, Dec. 2009, pp. 61-64.
- [Sod86] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, *Residue Number System Arithmetic: Modern applications in Digital Signal Processing*. New York: IEEE Press, 1986
- [Sto94] T. Stouraitis, "Analogue- and binary-to-residue conversion schemes," *IEE Proc. - Cir., Devices Syst.*, Apr. 1994, pp. 135-139.
- [Sto01] T. Stouraitis and V. Paliouras, "Considering the alternatives in low-power design," *IEEE Circ. Devices Magazine*, vol. 17, pp. 22-29, Jul. 2001.
- [Sza67] N. Szabo and R. Tanaka, *Residue Arithmetic and Its Application to Computer Technology*. New York: McGraw-Hill, 1967.
- [Tex99] *Texas Instrument Application Report SPRA509*, Texas Instrument, May 1999.
- [Tom08] T. Tomczak, "Fast sign detection for RNS $(2^n-1, 2^n, 2^n+1)$," *IEEE Trans. Cir. Syst. I: Regular Papers*, vol. 55, no. 6, pp. 1502-1511, Jul. 2008.
- [Tow03] W. J. Townsend, E. E. Swartzlander, Jr., and J. A. Abraham, "A comparison of Dadda and Wallace multiplier delays," F. T. Luk Ed., *Proc. SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, vol. 5205, pp. 552-560, Dec. 2003.
- [Tya93] A. Tyagi, "A reduced-area scheme for carry-select adders," *IEEE Trans. Computers*, vol. 42, no. 10, pp. 1163-1170, Oct. 1993.
- [Ulm93] Z. D. Ulman, M. Czyzak, and J. M. Zurada, "Effective RNS scaling algorithm with the Chinese Remainder Theorem decomposition," in *Proc. IEEE Pacific Rim Conf. Communications, Computers Signal Processing*, Victoria, BC, Canada, May 1993, pp. 528-531.

- [Ulm98] Z.D. Ulman and M. Czyzak, "Highly parallel, fast scaling of numbers in nonredundant residue arithmetic," *IEEE Trans. Signal Processing*, vol. 46, no. 2, pp. 487-496, Feb. 1998.
- [Ver02] H. T. Vergos, C. Efstathiou, and D. Nikolos, "Diminished-one modulo 2^n+1 adder design," *IEEE Trans. Computers*, vol. 51, no. 12, pp. 1389-1399, Dec. 2002.
- [Ver05] H. T. Vergos and C. Efstathiou, "On the design of efficient modular adders," *J. Cir., Syst. Computers*, vol. 14, no.5, pp. 965-972, Oct. 2005.
- [Ver09] H. T. Vergos and C. Efstathiou, "Efficient modulo 2^n+1 adder architectures," *Integration, the VLSI J.*, vol. 42, no. 2, pp. 149-157, Feb. 2009.
- [Ver10] H. T. Vergos, D. Bakalis, and C. Efstathiou, "Fast modulo 2^n+1 multi-operand adders and residue generators," *Integration, the VLSI J.*, vol. 43, no. 1, pp. 42-48, Jan. 2010.
- [Vin94] B. Vinnakota and V. V. Bapeswara Rao, "Fast conversion technique for binary-residue number system," *IEEE Trans. Cir. Syst. I*, vol. 41, no. 12, pp. 927-929, Dec. 1994.
- [Wan00] Y. Wang, "Residue-to-binary converters based on new chinese remainder theorems," *IEEE Trans. Cir. Syst. II: Analog and Digital Signal Processing*, vol. 47, no. 3, pp. 197-205, Mar. 2000.
- [Wan02] Y. Wang, X. Song, M. Aboulhamid, and H. Shen, "Adder based residue to binary number converters for $(2^n-1, 2^n, 2^n+1)$," *IEEE Trans. Signal Processing*, vol. 50, no. 7, pp. 1772-1779, Jul. 2002.
- [Wan03] W. Wang, M. N. S. Swamy, M. O. Ahmad, and Y. Wang, "A study of the residue-to-binary converters for the three-moduli sets," *IEEE Trans. Cir. Syst.-I*, vol. 50, no. 2, pp. 235-243, Feb. 2003.

[Ye08] Y. Ye, S. Ma, and J. Hu, “An efficient 2^n RNS scaler for moduli set $\{2^n-1, 2^n, 2^n+1\}$,” in *IEEE Symp. Information Science and Engineering (ISISE 2008)*, Shanghai, China, Dec. 2008, pp. 511-515.

[Zim99] R. Zimmermann, “Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication,” in *Proc. of 14th IEEE Symp. Computer Arithmetic*, Adelaide, Australia, Apr. 1999, pp. 158-167.