ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Modulo adders, multipliers and shared-moduli architectures for moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$

Shibu Menon

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University

in fulfilment of the requirement for the degree of

Master of Engineering

2007

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Acknowledgements

This work would not have been possible wihout the constant guidance and continuous support of my supervisor Assoc. Prof. Chang Chip Hong. Many have been his late night reading and correction sessions to see this work through. I would like to convey my heartfelt gratitude for all his support.

Thanks are also due to Dr. Cao Bin for his helpful insights into RNS and for conveying in simple terms what is essentially a very involved subject.

All the staff and students of CHiPES (Centre for High Performance Embedded Systems) have been very helpful and accomodating and for that I thank them.

A final word of gratitude is also due to the two people who have been very understanding and encouraging during periods of great stress - Mads and Pips.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Contents

Table of Contents ii

List of Figures v

List of Tables viii

Summary viii

1 Introduction 1

1.1 Motivation 1
1.2 Research Objectives 3
1.3 Contribution 5
1.4 Organization of the thesis 6

2 Background 7

2.1 Binary arithmetic units 7
2.1.1 Dot Notation 7
2.1.2 Binary Adders 7
2.1.3 Binary Multipliers 17
2.2 Residue Number Representation 22
2.2.1 Properties of RNS 22
2.2.2 Dynamic Range and Pairwise relatively prime moduli . . . 23
2.2.3 Common RNS Operations 24
2.2.4 Binary/RNS Conversion 24
2.2.5 Special Moduli set 25
2.3 RNS Code Generation and the OO paradigm 26
2.3.1 PERL 28
2.3.2 Verilog 28
2.4 Experimental Methodology 28

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3 Modulo Adders **30**
3.1 Modulo Addition 30
3.2 Modulo adders for the general moduli set 31
3.3 Modulo adders for moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$ 35
3.4 Modulo adders for moduli of type $2^n - 1$ 35
3.4.1 Direct EAC implementations 35
3.4.2 Implementations based on unwrapping EAC 39
3.4.3 Dual-representation of zero 44
3.5 Modulo Adders for type $2^n + 1$ 45
3.6 Modulo $2^n + 1$ Adders for diminished-one representation 48
3.6.1 Direct CEAC implementations 48
3.6.2 Implementations based on unwrapping CEAC 50
3.6.3 Prefix Adders with CEAC Unwrapped 52
3.7 Modulo $2^n + 1$ Adders for normal representation 54
3.7.1 PPFCI 56
3.7.2 TPP 58
3.8 Correction for diminished-one addition 60
3.9 Modified diminished-one representation 61
3.9.1 CLA Adders with zero handling 61
3.9.2 Parallel-Prefix adders with zero handling 62
3.10 Modulo adder Generation using BuRNS 63
3.11 Experimental evaluation of modulo adders 64
3.11.1 Cell Area 66
3.11.2 Timing 68
3.11.3 Dynamic Power 71
3.11.4 Interconnect Area 73
3.12 Summary 76

4 Modulo Multipliers **77**
4.1 Modulo Multiplication 77
4.2 Modulo multiplier for the general moduli set 79
4.3 Modulo multiplier for moduli set belonging to type $\{2^n-1, 2^n, 2^n+1\}$ 82
4.4 Modulo multiplier for moduli set of type $2^n - 1$ 83
4.4.1 MOMA for moduli of type $2^n - 1$ 84
4.5 Modulo multiplier for moduli set of type $2^n + 1$ 86
4.5.1 MOMA for moduli of type $2^n + 1$ 88
4.5.2 Review of modulo $2^n + 1$ multipliers 88
4.6 Proposed modulo $2^n + 1$ multiplier 92

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.7 Modulo multiplier generation using BuRNS 96
4.8 Experimental Evaluation 96
4.8.1 Cell Area 98
4.8.2 Timing 99
4.8.3 Dynamic Power 100
4.9 Summary 103

5 Shared moduli architectures 104
5.1 Definition of shared moduli architectures 104
5.2 Motivation for shared moduli architectures 104
5.3 Shared moduli architectures 107
5.3.1 Shared moduli architecture for modulo adder 107
5.3.2 Shared moduli architecture for MOMA 108
5.3.3 Shared moduli architectures for modulo multipliers 110
5.4 Experimental Results 111
5.5 Summary 112

6 Conclusion and Recommendations 113
6.1 Conclusion 113
6.2 Future Work 114

Author's Publication 116

References 117

Appendices 125

A Synthesis scripts 125

B Power Analysis scripts 127

C Simulation Scripts 128

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Figures

1.1 General architecture of an n-channel (n-moduli) RNS system . . . 3
1.2 Building blocks of RNS systems 3

2.1 Dot notation for computer arithmetic 8
2.2 A generic (m,k)-counter 8
2.3 Half Adder (a)Dot notation, (b)Logic Symbol , (c) Schematic and Full Adder (d)Dot notation, (e)Logic Symbol and (f) Schematic . 9
2.4 A k-bit Ripple Carry Adder (RCA) 9
2.5 A 3-bit Carry Look-ahead Adder (CLA) 11
2.6 A 16-bit Multilevel Carry Look-ahead Adder (CLA) 13
2.7 Prefix Operator for prefix addition 14
2.8 Prefix Trees for 16-bit prefix computation 15
2.9 Prefix Adder for 16-bit operands 16
2.10 A 16-bit sparse adder 18
2.11 A 4-bit Carry Select Block 19
2.12 General structure of a binary multiplier 20
2.13 An example of a Wallace tree for reduction of 7 operands 21
2.14 Dot notation of a Wallace tree for reduction of 7 operands 21
2.15 Booth recoding (a)Recoding rule (b)Example 22
2.16 Modularity and hierarchy in modulo arithmetic units 27

3.1 Modulo adders using binary adders as proposed in (a)[1] and (b)[2] 32
3.2 Modulo adders using CSA as proposed in [3] 33
3.3 Modulo adders as proposed in [4] 34
3.4 Example of EAC modulo $2^n - 1$ adder for $n = 3$ 36
3.5 Example of direct EAC implementation for a modulo $2^n - 1$ adder for $n = 16$ using multilevel CLA 37
3.6 Direct EAC implementation using Prefix adders 38
3.7 Direct EAC implementation using sparse prefix adder 39

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.8 Two-level CLA implementation of modulo $2^n - 1$ adder with EAC unwrapping . . . 41
3.9 Prefix tree for a modulo $2^n - 1$ adder using unwrapped EAC [5] . 43
3.10 Reduced Prefix Tree for a sparse adder . . . 43
3.11 Ling modulo $2^n - 1$ adder . . . 45
3.12 EAC leading to dual representation in modulo $2^n - 1$ addition : An example . . . 45
3.13 Solving the problem of dual representation of zero for modulo $2^n - 1$ adders . . . 46
3.14 Modulo $2^n + 1$ addition using CEAC : An example . . . 48
3.15 General structure of a modulo $2^n + 1$ adder using the diminished-one representation . . . 49
3.16 Prefix based modulo $2^n + 1$ adder with direct CEAC implementation 50
3.17 CLA based modulo $2^n + 1$ adder with direct CEAC implementation 51
3.18 A modulo $2^n + 1$ adder using two-level CLA with CEAC unwrapped for n=8 . . . 53
3.19 Prefix tree using modified prefix operators for a $2^n + 1$ adder with CEAC unwrapped for n=8 [6] . . . 55
3.20 Modulo $2^n + 1$ addition for the normal representation . . . 57
3.21 PPFCI Modulo $2^n + 1$ addition for the normal representation . . . 58
3.22 TPP Modulo $2^n + 1$ addition for the normal representation . . . 59
3.23 Diminished-one modulo $2^n + 1$ addition with correction . . . 60
3.24 Parallel-prefix modulo $2^n + 1$ adder with correct zero handling [7] 62
3.25 Class diagram for modulo adder generation in BuRNS . . . 65
3.26 Area evaluation of modulo adders for the special moduli set . . . 69
3.27 Delay evaluation of modulo adders for the special moduli set . . . 72
3.28 Dynamic Power evaluation of modulo adders for the special moduli set . . . 76

4.1 General structure of a modulo multiplier . . . 79
4.2 Modulo multiplier for general moduli set as proposed in [8] . . . 80
4.3 Modulo multiplier for a general moduli set as proposed in [9] . . . 81
4.4 Modulo multiplier for a general moduli set as proposed in [10] . . 82
4.5 CSA with EAC for moduli of type $2^n - 1$ . . . 84
4.6 MOMA for 8 operands for moduli of type $2^n - 1$ . . . 85
4.7 Modulo $2^n - 1$ multiplier for n=4 . . . 87
4.8 CSA with CEAC . . . 88
4.9 General structure of a modulo $2^n + 1$ multiplier . . . 89

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.10 Modulo $2^n + 1$ multiplier proposed in [11] 90
4.11 Modulo $2^n + 1$ multiplier proposed in [12] 90
4.12 Partial product terms for the modulo $2^n + 1$ multiplier proposed in [12] 91
4.13 Modulo $2^n + 1$ multiplier proposed in [13] 91
4.14 Proposed novel modulo $2^n + 1$ multiplier 95
4.15 Class diagram for BuRNS for generation of modulo multipliers . . 97
4.16 Cell area in μm^2 for modulo multipliers 100
4.17 Delay in ns for modulo multipliers 101
4.18 Dynamic power consumption (in μW) for modulo multipliers . . . 102

5.1 Programmable RNS datapaths in a reconfigurable system as proposed in [14] 105
5.2 A modulo multiplier with error detection and coding 106
5.3 Shared moduli architecture for modulo adders using a prefix adder based implementation 108
5.4 Shared moduli architecture for modulo adders using a CLA based implementation 109
5.5 Architecture of a CCSA 109
5.6 Shared moduli architecture for the PPG unit 110
5.7 Shared moduli architecture for modulo multiplier 111

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Tables

3.1 Cell area (in μm^2) for modulo adders 68
3.2 Critical path delay (in ns) for modulo adders 71
3.3 Dynamic power (in uW) for modulo adders 74
3.4 Interconnect area (in μm^2) for modulo adders 75

4.1 Cell area (in μm^2)for modulo multipliers 99
4.2 Critical path delay (in ns) for modulo multipliers 101
4.3 Dynamic power consumption (in ns) for modulo multipliers 102

5.1 Cell area for modulo multipliers (in μm^2): shared moduli vs. modulo $2^n + 1$ 112
5.2 Critical path delay for modulo multipliers (in ns): shared moduli vs. modulo $2^n + 1$ 112

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Summary

Modulo arithmetic circuits are ubiquitous in Residue Number System (RNS) architectures. The basic arithmetic units used are modulo adders and multipliers. The efficient implementation of modulo adders and multipliers thus form the cornerstone of high performance RNS.

Special moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ are known to lead to efficient implementation of moduli arithmetic units and converters for binary-to-residue and residue-to-binary transformation. Most modern implementations of RNS thus concentrate on moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$. For this reason, moduli arithmetic units for special moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$ form an important and growing area of research.

A multitude of architectures exist for modulo arithmetic units for moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$. From the perspective of RNS system design, this translates to a bewildering choice of architectures to consider. An informed decision on the type of modulo arithmetic unit to choose can be made only with the availability of standard VLSI performance metrics such as area/power/timing. This work provides such a standardized evaluation of commonly known modulo adders and multipliers. Pre-layout simulation of VLSI metrics for a standard cell implementation based on the TSMC $0.18um$ process model were used to evaluate well known modulo adders and multipliers for moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$. As part of the evaluation methodology, a code generator for modulo adders and multipliers using the Object Oriented Paradigm (OOP) has been developed.

The evaluation shows a clear performance gap between modulo multipliers for moduli of type $2^n + 1$ and moduli of types 2^n and $2^n - 1$. A novel modulo multiplier architecture has been proposed to reduce this performance gap. The proposed multiplier outperforms other well known modulo $2^n + 1$ multipliers for high operand sizes.

Shared moduli architectures are a growing area of research in RNS seeking to make arithmetic units reusable for multiple moduli. In particular, reusable arithmetic units for moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$ can have varied

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

applications. This work details some of these applications as well as proposes an architectural framework to implement shared architectures for modulo addition and multiplication with moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 1

Introduction

This chapter gives a succinct introduction to the work done as part of this project and its presentation in this thesis. The first part of this chapter deals with the motivation and goals of this project, and lays out the significant achievements for the same. The second part sets out the organization of the thesis.

1.1 Motivation

Modern Digital Signal Processing (DSP) applications, operating at bandwidths that exceed 100 MHz [15] place a huge premium on the performance of its basic building blocks, namely arithmetic circuits. The bottleneck for computation using the normal binary system arises from its need for carry propagation [16]. An n-bit system depends on carry propagation from bit 0 to bit $n-1$. While methods exist to parallelize the carry propagate mechanism in the form of Carry Look Aheads (CLA) [17], the inherent bottlenecks in the binary system need to be addressed further.

Residue Number Systems (RNS) involve breaking down an n-bit number into its component residues of m-bits, with $m < n$. Computation of particular arithmetic operations for each residue can proceed along independent channels without the need for inter-channel communication. This limits carry propagation to within channels of much smaller bit-width. While conversion to and from the standard binary representation constitute overheads, some applications have been shown to achieve overall performance improvements by the use of RNS [18]. Additionally, modulo arithmetic lends itself admirably to other applications like error-detection, error correction and cryptography. Modulo arithmetic, defined as arithmetic operations on the individual residue channels are thus an important and growing area of research.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

A RNS consists of three basic building blocks: Forward Converter (FC), Residue Arithmetic Unit (RAU) and Reverse Converter (RC). This is shown in Fig. 1.1. While the only constraint placed on the moduli set from a number theoretical standpoint is that they be pairwise relatively prime, efficient VLSI implementation requires the usage of special moduli sets. It has been well established that moduli sets belonging to the form $\{2^n-1, 2^n, 2^n+1\}$ have special properties such as End Around Carry [19], Complemented End Around Carry [6], etc. that lead to the most efficient forms of converters and moduli arithmetic units [18, 20, 21]. For this reason, the usage of moduli belonging to one of the three types is very common in modern RNS implementations [11, 22, 23]. Additionally, the moduli arithmetic units for many proposals of special four and five moduli sets are derivatives of the special three moduli set [24, 25, 26]. Thus, moduli arithmetic units for modulus of type $\{2^n-1, 2^n, 2^n+1\}$ have a critical role to play in RNS architecture design.

The three basic structures that constitute an RNS system all have common building blocks of modulo adders and modulo multipliers. The design of RNS systems can thus be considered as a cascading of basic modulo arithmetic building blocks as shown in Fig. 1.2. A multitude of architectures exist for designing modulo adders and multipliers for the special triple moduli set [27, 19, 11, 5, 4]. When designing at the system level, the selection of the right architectures for the building blocks has a very large impact on the VLSI performance metrics of area, delay and power. What is lacking at the present moment is a systematic analysis of these modulo arithmetic units that will assist a designer to make an informed choice about which moduli set to choose and the particular architecture for the chosen moduli. A study of the fundamental arithmetic blocks with an emphasis on moduli belonging to the special three moduli sets will thus help in architectural design of Residue Number Systems.

One common disadvantage of using RNS in terms of VLSI performance metrics is the area overhead incurred. This comes about due to the need for converters, both forward as well as reverse. A possibility for area reduction is the use of shared-moduli architectures. The extensive usage of moduli sets of the form $\{2^n-1, 2^n, 2^n+1\}$ in modern RNS thus motivates us to look at the possibility of using shared-moduli architectures for the special triple moduli set. A formal framework for moduli sharing also needs to be articulated.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

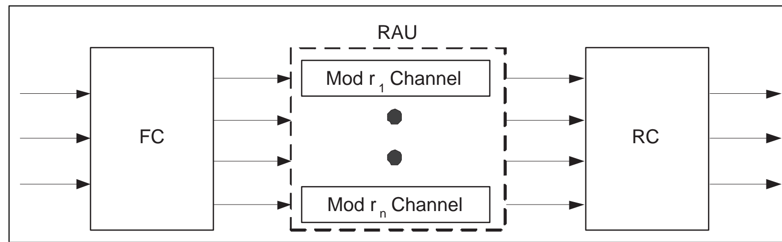

Figure 1.1: General architecture of an n-channel (n-moduli) RNS system

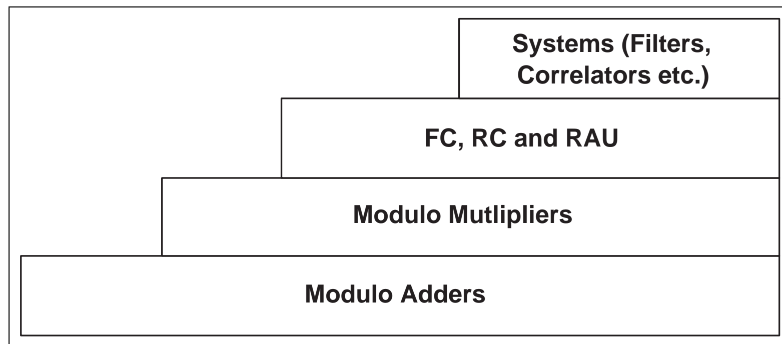

Figure 1.2: Building blocks of RNS systems

1.2 Research Objectives

The objectives of this research are as follows.

1. Implementation and performance evaluation of modulo adders and multipliers for the special moduli set

   Moduli adders and multipliers are the basic building blocks for all three main components of a RNS system. A multitude of architectures and design methodologies are known for modulo adders and multipliers for the special moduli set. However, most existing studies of modulo adders and multipliers focus primarily on the analytical area-time complexity for the general moduli or on specific realization of a particular modulus. One incentive of this research is to provide an appraisal of existing modulo adder and multiplier architectures for the special triple moduli set in terms of the VLSI area, delay and power consumption. Our plan is to code all designs in structural Verilog at Register Transfer Level (RTL) and synthesize them using an industrial grade ASIC design tool, Synopsys Design Compiler and the TSMC 0.18μm CMOS standard cell libraries. The relative performance differences due to the omission of actual interconnections in prelayout simulations will be mediated by using an appropriate and consistent wireload

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

model for all designs being simulated. The scope of the study is targeted at existing architectures for the multiply- accumulate (MAC) units of special moduli set RNS. The purpose is to assist in architectural decision making for the complete RNS systems as well as to provide a modular approach to system-level design.

2. Development of a low overhead configurable multi-modulus modulo architectures

   One of the driving forces behind the evaluation of VLSI metrics for different residue arithmetic operations for varying dynamic ranges is the important concept of RNS system sustainability and flexibility. RNS, by virtue of its characteristic of providing independent channels is well-suited to applications requiring configurability. One goal of this research is to support a larger project of an amorphous RNS processor with multiple morphological modulus operators that can be utilized for designing systems which have variable dynamic ranges and application complexity. Configurable architectures using modulo arithmetic, however, is a field of study that is relatively new. A study of applications for configurable RNS architectures thus contributes a significant amount to existing knowledge of ways in which RNS systems can be applied to real world system design. A formal framework for designing configurable architectures for the triple moduli set will also need to be articulated for this goal.

3. Development of an automatic HDL code generator

   An additional aspect of analyzing architectural tradeoffs for RNS designs is the ability to generate parameterizable RTL code for various types of moduli adders and multipliers. A rigorous analysis of different modulo operations for varying dynamic range is confronted with high development cost and engineer efficiency. To address this issue, an object oriented paradigm for automatic generation of RTL codes for moduli arithmetic units is aimed at, which will capitalize on the modularity of RNS system design to ease hardware description language (HDL) code generation. We would like to automate the code generation by building a code generator in PERL (Practical Extraction and Reporting Language) called BuRNS (Build RNS). The code generator generates parameterized Verilog codes for modulo adders and multipliers that can be plugged into a RNS system. The idea is to support multiple hardware targets, such as the mapping to any available standard cell library. We believe that the successful development of this

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

small scale code generator will facilitate the same for a larger scale system integration with the desired engineer efficiency, sustainability and flexibility.

1.3 Contribution

The salient contributions of this body of work are:

1. A thorough study of modulo adders and multipliers for different moduli sets.

   A comparison of modulo adders for different moduli sets and using various types of design methodologies has been done to identify the bottlenecks accrued in RNS designs. The performance evaluation has been used to draw conclusions on the viability of various modulo arithmetic units in RNS. A clear performance gap between modulo adders and multipliers for modulo $2^n + 1$ and modulo arithmetic units for moduli 2^n and $2^n - 1$ has been demonstrated.

2. A novel modulo multiplier for normal representation of modulo $2^n + 1$ numbers has been proposed. Comparisons with existing modulo $2^n + 1$ multipliers show a slight performance improvement for high operand sizes.

3. Articulation of a shared-moduli structure and its uses in the design of configurable RNS architectures for the special moduli set.

   A configurable architecture is one in which the individual channels can be selectable among multiple moduli sets. Shared-moduli architectures for modulo adders and multipliers ease the design on configurable RNS and the methodology for such moduli sharing is presented in this thesis. Some possible applications for the shared-moduli architectures have also been considered.

4. Development of an application for the automated generation of RTL (Register Transfer Level) codes for modulo arithmetic units.

   An object oriented approach to generation of the fundamental units of RNS has been presented. In this approach, each subunit of a design is considered an object with its own properties and functions to create the object as well as to instantiate it. The developed application has been termed BuRNS (Build RNS). Such an application helps in the generation of synthesizable and parameterized codes in Verilog for the investigation of architectures presented in this thesis.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

1.4 Organization of the thesis

This thesis is organized keeping in mind the concept of RNS building blocks (Fig. 1.2). Chapter 2 starts off with a discussion of binary adders and multipliers. Following this, some of the theoretical foundations behind RNS modulo arithmetic are set out. This chapter also presents the experimental methodology that has been used to evaluate the performance of moduli architectures. Finally, the object oriented philosophy behind the designed code generator is articulated.

The basic arithmetic unit of RNS is the modulo adder. Chapter 3 presents the various architectures for modulo adders. Modulo adders for the general moduli set is first discussed followed by modulo adders for the special moduli of the type 2^n. Next, the commonly known architectures for moduli of the type $2^n - 1$ is presented. Following this, the architectures for moduli of the type $2^n + 1$ is presented. The code generation algorithm for modulo adders used in BuRNS is then explained. Finally, the experimental results of different moduli adders are analyzed and the conclusions are drawn.

Chapter 4 covers architectures for modulo multipliers. This chapter follows a structure similar to the chapter on modulo adders. We start off with a discussion of modulo multiplier architectures for the general moduli set. This is followed by a discussion of the architectures for moduli belonging to the special moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The proposed novel modulo $2^n + 1$ multiplier is then presented and the performance of various modulo multipliers is evaluated by using BuRNS for the generation of synthesizable HDL codes. The results are analyzed and discussed.

Chapter 5 deals with shared-moduli architectures and their applications in building configurable RNS. Firstly, the notion of shared-moduli architectures is introduced. This is followed by the articulation of novel shared-moduli architectures for the special triple moduli set. The applications that have been identified for the usage of the shared-moduli architectures are presented. The chapter is wrapped up by the experimental evaluation of the efficiency of using shared-moduli architectures.

Chapter 6 summarizes the salient contributions of this project and proposes future work to carry forward the work presented in this thesis.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 2

Background

This chapter sets out some of the background information behind the work presented in this thesis. A brief description of binary adders and multipliers is presented first. Mathematical formulation of various RNS properties and operations is presented next. Following this, a brief discussion of the Object Oriented Programming (OOP) paradigm and its use to ease the generation of moduli arithmetic circuit description is presented. Finally, the experimental methodology for performance evaluation of the moduli arithmetic circuits is discussed and the rationale behind it articulated.

2.1 Binary arithmetic units

2.1.1 Dot Notation

A k-bit binary number is represented as $X = \sum_{j=0}^{k-1} 2^j x_j$. The dot notation [17, 28] is used extensively in this thesis. In the dot notation, a dot represents a variable with a binary value of either 0 or 1. Each dot of an integer variable is weighted according to its position in the normal binary representation of the variable. A dot matrix represents a collection of bits from multiple integer variables where all dots in the same column have the same weight. A box around the dots represents an operation on the binary bits represented by the enclosed dots. An example of a dot notation for the addition of two 6-bit binary numbers X and Y to produce a 7-bit number Z is shown in Fig. 2.1.

2.1.2 Binary Adders

Modulo adder architectures usually draw their basic design concepts from binary adders and an understanding of binary adders is thus essential from the viewpoint

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

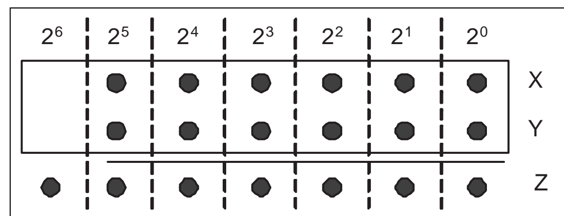

Figure 2.1: Dot notation for computer arithmetic

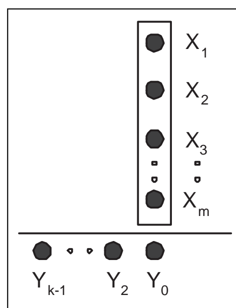

Figure 2.2: A generic (m,k)-counter

of designing modulo adders.

The most basic unit of a binary adder is the 1-bit adder that counts the number of 1's in m 1-bit numbers to produce a k-bit output [17, 28]. This basic unit is also known as a (m,k)-counter. A generic (m,k)-counter is shown in Fig. 2.2. The box around the dots, in this case represents the (m,k)-counter operation. All binary adders can be built up from these basic 1-bit adders. The more important 1-bit adders are the Half Adder (HA) ((2,2)-counter) and the Full Adder (FA) ((3,2)-counter). Fig. 2.3 shows the dot notation for the HA and FA as well as the logic equations and circuit schematic. For both adders, the output component that contributes to the next higher weight is called the carry output C_{out}.

A two-operand binary adder is built using the basic building blocks of FA and HA. Two main approaches towards binary addition can be discerned at this point. Carry-Propagate Adders (CPAs) rely on propagation of carries across the bits of an operand as part of the addition process. The output of a CPA for a k-bit 2-operand addition is a single $(k+1)$-bit number. Carry-Save Adders (CSAs) on the other hand, avoid carry propagation by treating them as an intermediate output and saving them for later propagation. The output of a CSA for a k-bit 2-operand addition is thus a k-bit sum and a k-bit right-shifted carry.

While there are many varieties of CPA [17, 28], the two most commonly known ones are the Ripple Carry Adder (RCA) and the Carry Look Ahead (CLA).

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

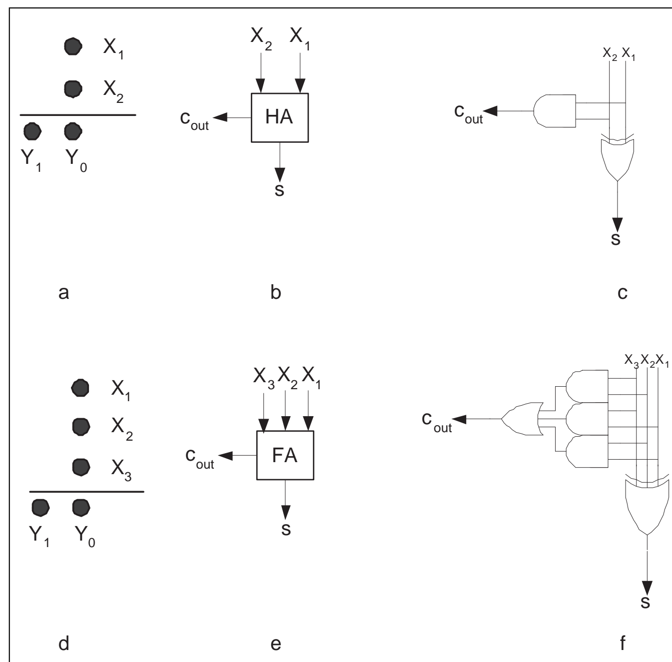

Figure 2.3: Half Adder (a)Dot notation, (b)Logic Symbol , (c) Schematic and Full Adder (d)Dot notation, (e)Logic Symbol and (f) Schematic

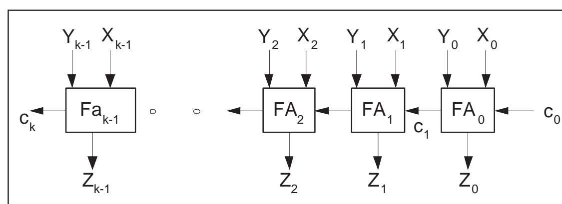

Figure 2.4: A k-bit Ripple Carry Adder (RCA)

2.1.2.1 RCA

Ripple Carry Adder (RCA) [17] ripples the carry serially across bits of the operands and provides a low-complexity, low-area but timing inefficient implementation, whose latency is a linear function of the bit-width, k. Fig. 2.4 shows the RCA structure.

2.1.2.2 CLA

CLA, on the other hand provides a delay efficient implementation, whose latency is proportional to $log_2 k$ [17]. CLA relies on the carry look-ahead principle. This principle answers the question: *Can the carry out at bit position p be calculated with knowledge of carry in at position $p - n$ without having to compute the intermediate carries*? Let the propagate (p_i) and generate (g_i) signals for bit i

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

be defined as in Eq. 2.1 [17, 29], where the operator $\cdot$ denotes a logical AND operation, $\oplus$ denotes an exclusive-OR operation and $+$ denotes an inclusive-OR operation. In words, what it conveys is that if g_i is '1', then a carry out is generated from position i. If p_i is '1', then a carry input to bit i will propagate to the carry out from this bit. The effect of propagate/generate (henceforth, referred to as P/G) signals on carry generation is encapsulated by Eq. 2.2 [17].

$$\begin{aligned} g_i &= x_i \cdot y_i \\ p_i &= x_i \oplus y_i \end{aligned} \tag{2.1}$$

$$c_{i+1} = g_i + p_i \cdot c_i \tag{2.2}$$

The utility of the P/G signal lies in the fact that it lets us compute the carry of bit k in parallel to that of bit $k-1$ for any k. This is done by a simple logic transformation known as the carry unrolling. The logic equations for carry unrolling is shown in Eq. 2.3 [17, 29]. For the sake of brevity the dots representing logical AND operations are omitted. As can be seen from this equation the calculations of all carries can be represented as a SOP (Sum Of Products) depending only on the inputs and not on any previously generated carries. Once these carries have been generated in parallel, the calculation of sum outputs is a matter of using FA's in parallel. An example of a 3-bit adder using CLA is shown in Fig. 2.5.

$$\begin{aligned} c_1 &= g_0 + p_0c_0 \\ c_2 &= g_1 + p_1c_1 \\ &= g_1 + p_1(g_0 + p_0c_0) \\ &= g_1 + p_1g_0 + p_1p_0c_0 \\ c_3 &= g_2 + p_2c_2 \\ &= g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_0) \\ &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \\ &\dots \\ &\dots \\ c_{k+1} &= g_k + p_kg_{k-1} + p_kp_{k-1}g_{k-2} + \dots + p_kp_{k-1}p_{k-2}\dots p_1p_0c_0 \end{aligned} \tag{2.3}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

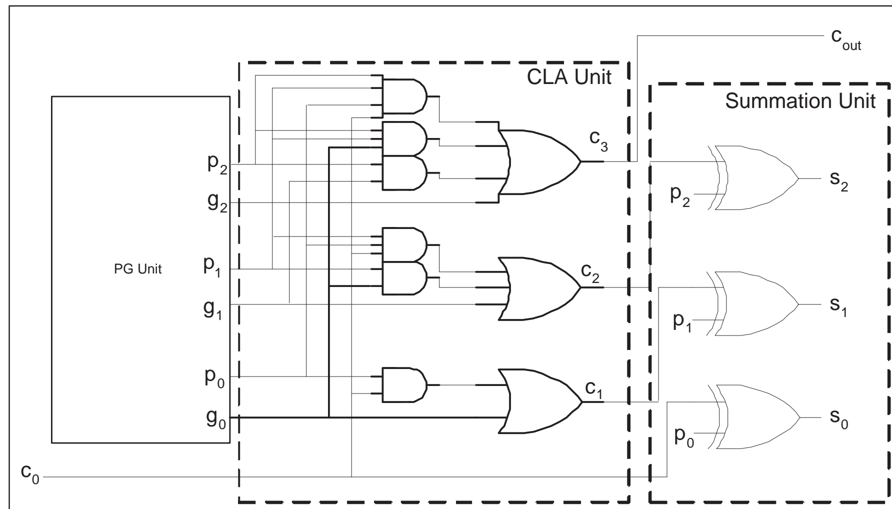

Figure 2.5: A 3-bit Carry Look-ahead Adder (CLA)

The fan-in requirements of the CLA logic circuitry suggests that a CLA scheme as shown in Fig. 2.5, while suitable for low bit widths, is not amenable to addition of large bit widths. **Multilevel Lookahead** is a scheme that is used to overcome this problem [17]. Multilevel lookahead relies on the derivation of a block P/G signal. The signals denoted as $p_{[i,j]}$ and $g_{[i,j]}$ denotes P/G for groups consisting of bits i to j. An example of a block P/G signal for bit groups 0-3 and 4-7 is shown in Eq. 2.4. The carry out from the MSB of the block is also shown. Multilevel lookahead breaks down CLA into multiple levels by using word-level carries at an intermediate level instead of bit-level carries. Word size can be chosen based on allowable fan-ins for the OR gates in the CLA. Typically, word size of 4-bits is chosen. Using multilevel lookahead with word size of 4-bits, a 16-bit adder is shown in Fig. 2.6. The PG Unit generates the P/G signals at the bit levels. Global Propagate Generate Unit (GPG Unit) combines these bit-level P/G signals to give block-level P/G signals according to Eq. 2.4 [17]. The Global P/G signals are used to generate the carry outputs of blocks in the Between Groups Carry Look Ahead Unit (BGCLA Unit). Finally, the Global Carry Look Ahead Unit (GCLA Unit) generates the bit carries.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$\begin{aligned} p_{[3,0]} &= p_3p_2p_1p_0 \\ g_{[3,0]} &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \\ c_4 &= g_{[3,0]} + p_{[3,0]}c_0 \\ p_{[7,4]} &= p_7p_6p_5p_4 \\ g_{[7,4]} &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 \\ c_8 &= g_{[7,4]} + p_{[7,4]}c_4 \end{aligned} \tag{2.4}$$

2.1.2.3 Transfer signal in CLA

A few variations of the CLA scheme exist in literature [17]. Once such variation that occurs commonly involves the usage of the transfer signal (t_i), instead of the propagate signal (p_i) for carry unrolling. The transfer signal is defined as:

$$t_i = x_i + y_i \tag{2.5}$$

The utility of the transfer signal lies in the fact that the delay-inefficient exclusive-OR operation for p_i generation in the critical path is replaced by a inclusive-OR operation that is more delay efficient for standard cell implementations. The downside of using the transfer signal is the increase in gate count due to the need to compute both p_i (needed for summation in the Summation Unit) and t_i.

2.1.2.4 Prefix Adders

A prefix algorithm can be formulated as the calculation of n outputs $(z_{n-1} \cdots z_0)$ from n inputs $(x_{n-1} \cdots x_0)$, where any arbitrary associative operator $\bullet$ defines the calculation as shown in Eq. 2.6 [17, 28, 30, 31, 32, 33, 34, 35]. Thus, the output at any prefix n depends upon all the previous inputs. Due to the associativity of the $\bullet$ operator, the operations can be carried out in any order.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

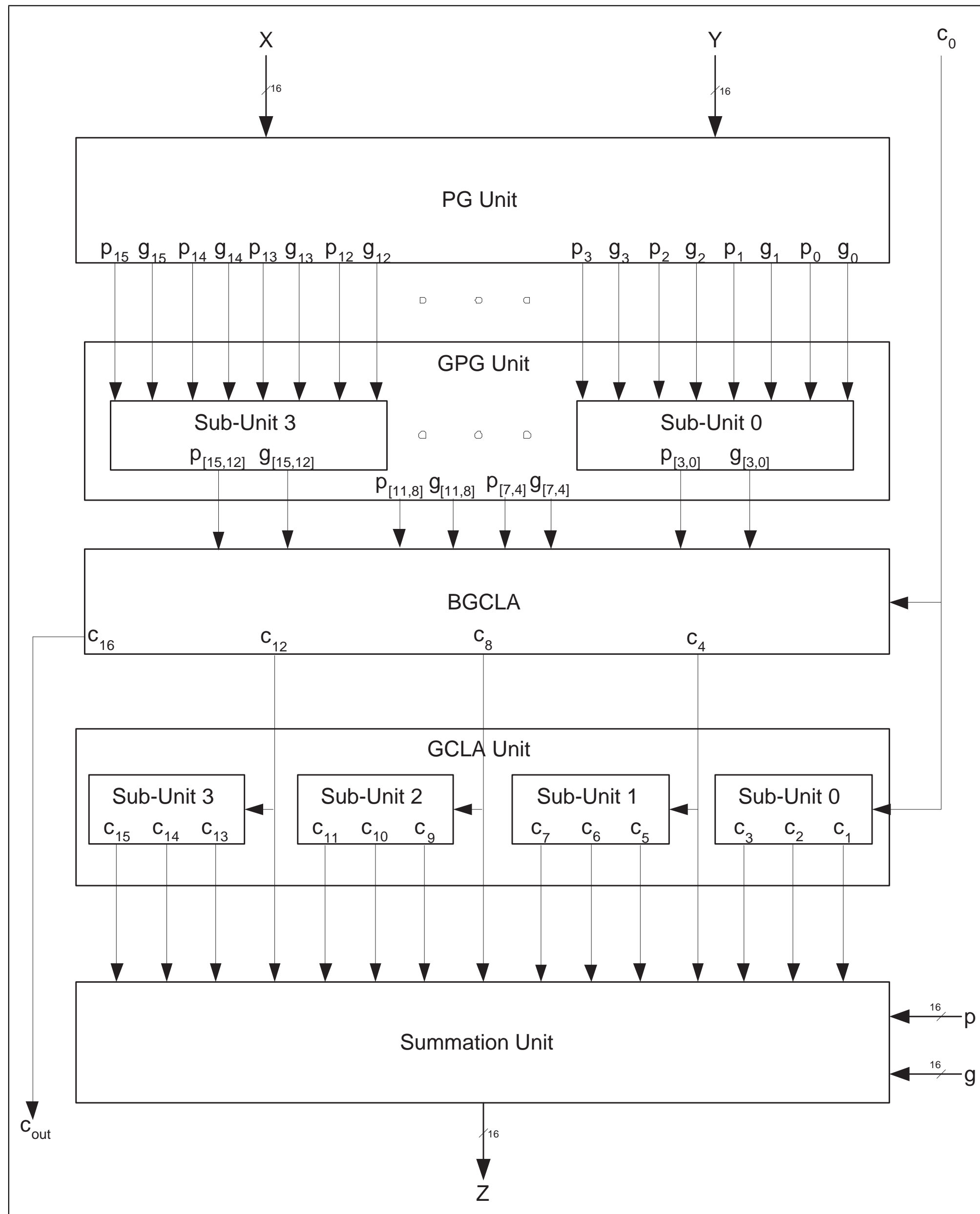

Figure 2.6: A 16-bit Multilevel Carry Look-ahead Adder (CLA)

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

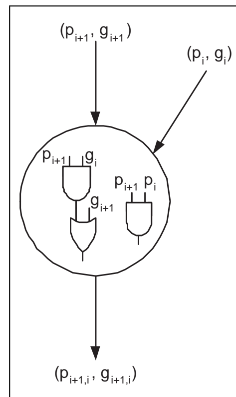

Figure 2.7: Prefix Operator for prefix addition

$$\begin{aligned} z_0 &= x_0 \\ z_1 &= x_1 \bullet x_0 \\ z_2 &= x_2 \bullet x_1 \bullet x_0 \\ &\dots \\ &\dots \\ z_{n-1} &= x_{n-1} \bullet x_{n-2} \cdots x_1 \bullet x_0 \end{aligned} \tag{2.6}$$

Carry determination in binary adders can be viewed as a prefix problem. This view uses the P/G signals defined in Eq. 2.1. Consider a prefix problem defined by inputs which are the bit-level P/G signals((p_{n-1}, g_{n-1}), $\cdots (p_0, g_0)$). The generation of block level P/G signals can then be defined by a prefix operator $\bullet$ as shown in Eq. 2.7. Once the block-level P/G signals are determined, the carry out for a bit position i is the generate signal $g_{[i,0]}$. The prefix operator in this equation can be considered as a prefix node as shown in Fig. 2.7 for VLSI implementation.

$$\begin{aligned} (p_0, g_0) &= (p_0, g_0) \\ (p_{[1,0]}, g_{[1,0]}) &= (p_1, g_1) \bullet (p_0, g_0) = (p_1 \cdot p_0, g_1 + p_1 \cdot g_0) \\ (p_{[2,0]}, g_{[2,0]}) &= (p_2, g_2) \bullet (p_1, g_1) \bullet (p_0, g_0) \\ &= (p_2 \cdot p_1 \cdot p_0, g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0) \\ \dots \end{aligned} \tag{2.7}$$

The utility of considering carry determination as a prefix operation is that

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

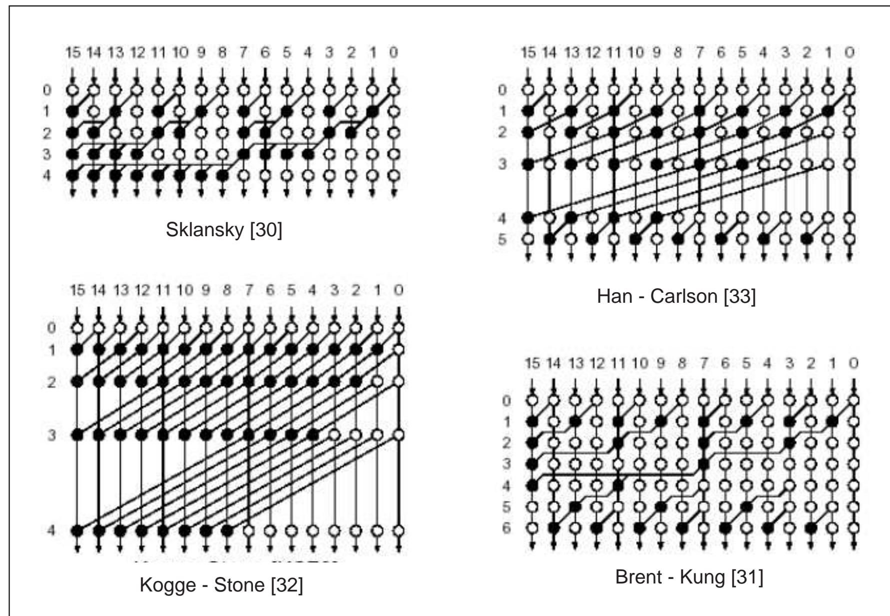

Figure 2.8: Prefix Trees for 16-bit prefix computation

many serial and parallel algorithms with well known properties exist to implement prefix structures [30, 31, 32, 33, 34, 35]. A very good discussion of the prefix structures and their properties from a VLSI implementation point of view can be obtained from [28]. Examples of 16-bit prefix trees using some commonly used structures is shown in Fig. 2.8. The shaded bullets represent the prefix node, while the empty bullet denotes a buffer that directly passes the input to the output without any logical transformation. The general structure of a binary adder that utilizes a prefix tree is shown in Fig. 2.9.

2.1.2.5 Ling Adders

An adder structure that achieves significant hardware savings is the Ling adder [36]. The main concept behind the Ling adder is the definition of a group carry term h_i, such that:

$$h_i = c_i + c_{i-1} \tag{2.8}$$

Ling adders propagate h_i instead of c_i and this leads to reduction in the SOP expression for carry computation, leading to hardware savings [36]. Consider a 4-bit example. The carry term for bit 4 can be expressed as:

$$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \tag{2.9}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

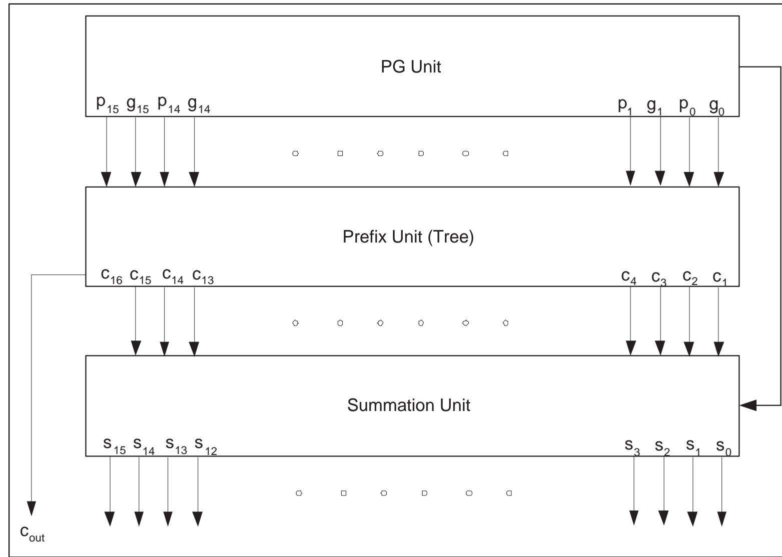

Figure 2.9: Prefix Adder for 16-bit operands

For the same bit width, h_4 can be expressed as:

$$\begin{aligned} h_4 &= c_4 + c_3 \\ &= g_3 + c_3 p_3 + c_3 \\ &= g_3 + c_3 \\ &= g_3 + g_2 + p_2 c_2 \end{aligned} \tag{2.10}$$

Using the properties, $g_i = h_{i+1} g_i$, $g_i p_i = 0$ and $p_i c_i + p_i c_i p_i = p_i c_i$, this equation can be reduced as follows:

$$\begin{aligned} h_4 &= g_3 + h_3 g_2 + p_2 c_2 + g_2 p_2 + p_2 c_2 p_2 \\ &= g_3 + h_3 g_2 + p_2 (c_2 + g_2 + p_2 c_2) \\ &= g_3 + h_3 g_2 + p_2 (c_2 + c_3) \\ &= g_3 + h_3 g_2 + p_2 h_3 \\ &= g_3 + h_3 (g_2 + p_2) \\ &= g_3 + h_3 t_2 \end{aligned} \tag{2.11}$$

The Ling carries h_i can be unrolled in a way similar to the unrolling of carries in a normal CLA.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$\begin{aligned} h_4 &= g_3 + t_2(g_2 + t_1 h_2) \\ &= g_3 + t_2 g_2 + t_2 t_1 h_2 \\ &= g_3 + g_2 + t_2 t_1 h_2 \end{aligned} \tag{2.12}$$

Unrolling all intermediate Ling carries in this way will reduce the above expression to:

$$h_4 = g_3 + g_2 + t_2 g_1 + t_2 t_1 g_0 \tag{2.13}$$

As can be seen from Eq. 2.14, the SOP expression for Ling carry has four terms with the critical path being through a 3-input AND gate. The expression for normal carry as shown in Eq. 2.9 has four terms with the critical path being through a 4-input AND gate. Thus adders based on propagation of Ling carries will compute the carries faster than normal CLA. The downside of Ling adders is that the computation of the sum term s_i is more complex:

$$s_i = (t_i \oplus h_{i+1}) + h_i g_i t_{i-1} \tag{2.14}$$

While the commonly known Ling Adders rely on a CLA approach to propagating the Ling carries, a parallel prefix approach was proposed in [37].

2.1.2.6 Sparse adders

A special category of adders that reduces the wiring complexity and are especially useful for higher bit widths are the sparse adders. Sparse adders reduce the complexity of the carry computation unit by computing the carries only at the boundaries of blocks with predefined sizes. These predefined sizes are sometimes referred to as the sparseness. An example of a 16-bit binary adder with a sparseness of 4 is shown in Fig. 2.10. The summation units for sparse adders are designed as carry select blocks (CS). The structure of a CS block is shown in Fig. 2.11. The carry computation unit for sparse adders can be based on CLA or parallel prefix algorithms [38].

2.1.3 Binary Multipliers

The design of modulo multipliers relies on much of the same concepts that underpin the design of binary multipliers. While many flavors of binary multipliers exist [17, 29], the most commonly used types of fast multipliers are tree multipliers

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

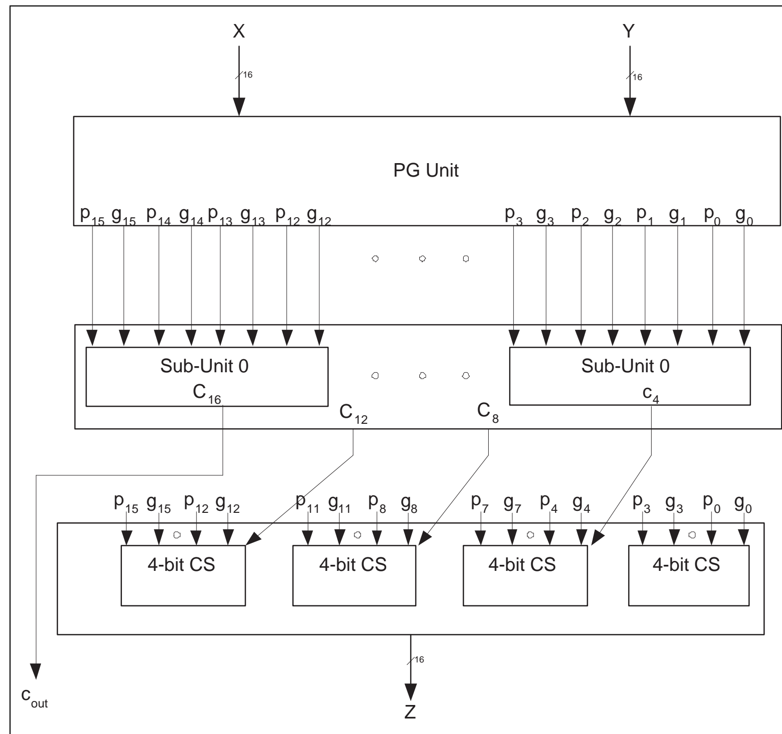

Figure 2.10: A 16-bit sparse adder

that have a common structure as shown in Fig. 2.12. The Partial Product Generator (PPG) generates the partial products from the multiplier and multiplicand. For two inputs of bitwidth n, the number of partial products is in general n with a maximum bitwidth of $2n - 1$. These n partial product operands are reduced to two operands by the usage of a multioperand adder with n input operands. Finally, a carry propagation adder is used to obtain the result in normal binary representation.

2.1.3.1 Multioperand Addition

A generic multioperand adder reduces n input operands to 2 outputs. Common implementations of multioperand addition rely on a tree of CSA to achieve this reduction. Two types of tree structures that have gained popularity are the Wallace tree [39] and the Dadda tree [40] structures. Wallace tree structure tries to reduce the number of operands at the earliest given opportunity, while the Dadda tree reduces the number of operands by combining operands as late as possible while keeping the critical path length of the CSA tree intact [17]. An example of a multioperand adder using a Wallace tree of CSA is shown in Fig. 2.13. In this example the reduction of seven operands to two is shown. The equivalent dot notation for this reduction is shown in Fig. 2.14. While the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

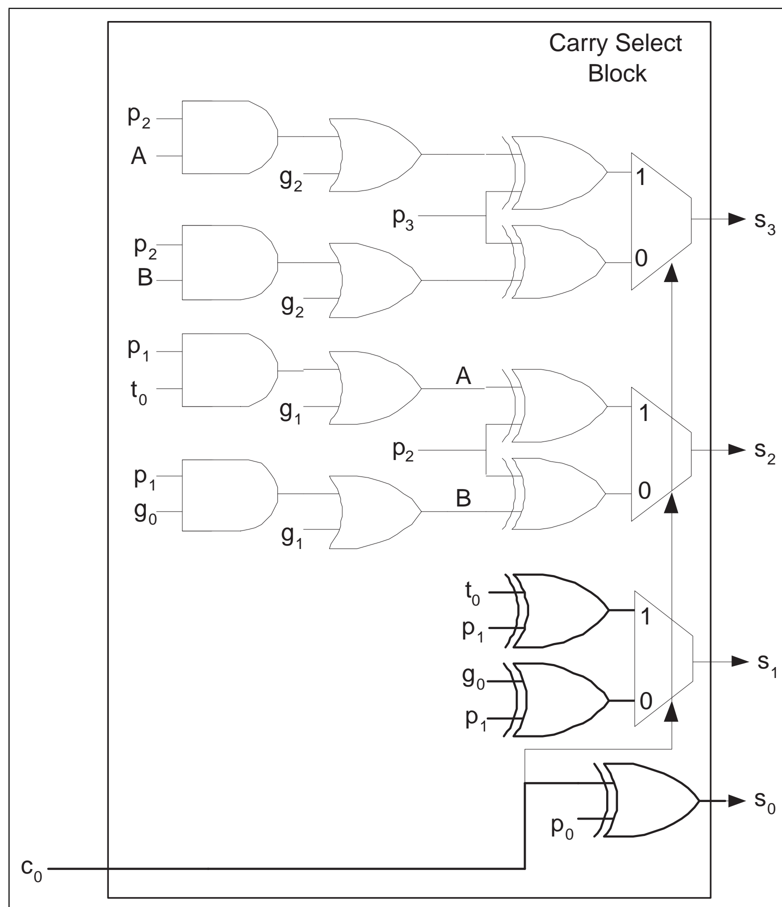

Figure 2.11: A 4-bit Carry Select Block

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

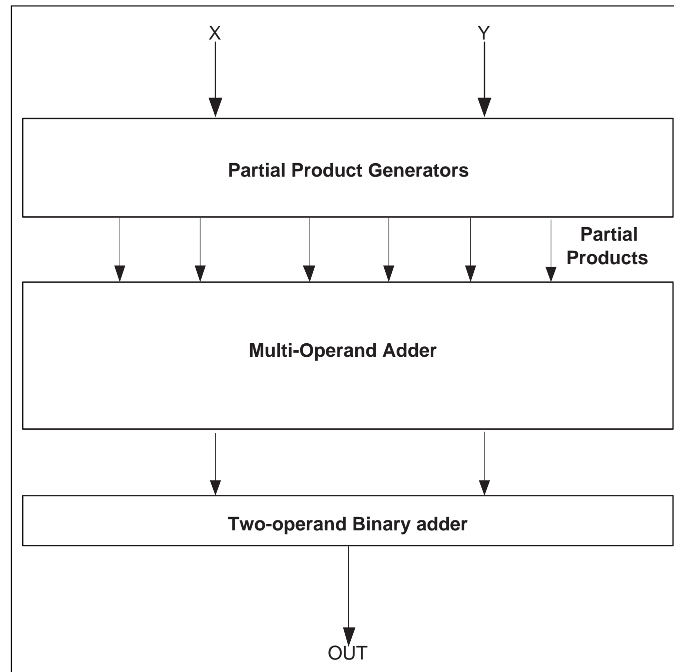

Figure 2.12: General structure of a binary multiplier

reduction tree shown here operates at the word-level, a similar tree structure that operates at bit-level might sometimes lead to more efficient implementations [17].

2.1.3.2 Booth Recoding

Booth recoding is a technique used to speed up multiplication by transforming the digit set to reduce the number of shift and add operations[41]. In a normal binary multiplication, the appearance of the digit 1 in the multiplier means a shift-and-add operation. The digit set transformation of Booth is based on the observation that consecutive ones in a binary number can be reduced to a simple subtraction as shown below:

$$2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} 2^i \tag{2.15}$$

Using this property, a binary number with digit sets [0,1] can be recoded to a digit set [-1,0,1]. An example of such a recoding is shown in Fig. 2.15. As can be seen, the recoded term can have fewer non-zero digits leading to fewer shift-add operations. In twos complement arithmetic, radix-2 Booth recoding does not help to reduce the number of partial products below that of a plain multiplier without any encoding. Higher radix Booth encoders are normally used in high precision multiplication but the hard multiple generation problem has limited its

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

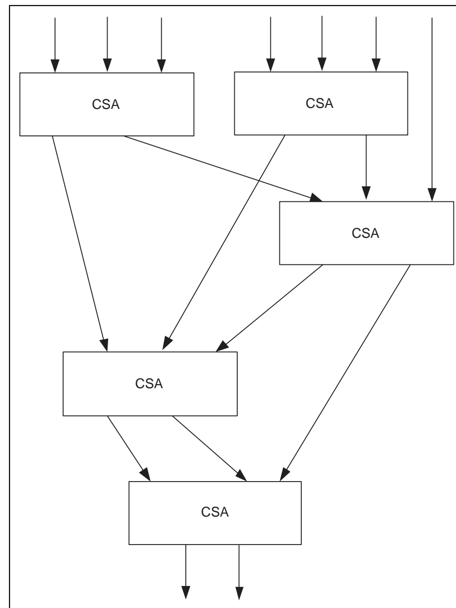

Figure 2.13: An example of a Wallace tree for reduction of 7 operands

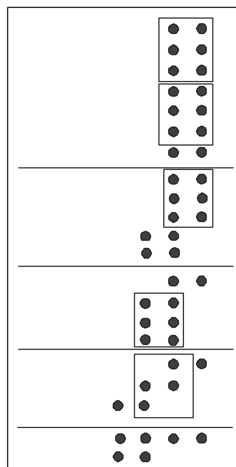

Figure 2.14: Dot notation of a Wallace tree for reduction of 7 operands

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

x_i	x_{i-1}	y_i
0	0	0
0	1	1
1	0	-1
1	1	0

(a)

0	1	1	1	1	1	1	0	0
1	0	0	0	0	0	0	-1	0

(b)

Figure 2.15: Booth recoding (a)Recoding rule (b)Example

advantage gained from the reduction of partial products.

2.2 Residue Number Representation

Residue Number System, in its essence is a system of representation of numbers as a N-tuple of residues with respect to a set of relatively prime numbers called moduli (the plural form of modulus) [16]. The residue representation of an integer X is $\{x_1, x_2..., x_N\}$, where the residues x_i are defined by the modulo operation in Eq. 2.16. The integer X is thus represented as shown in Eq. 2.17 [16]. The integer set $\{m_1, m_2...m_N\}$ is known as the moduli set.

$$x_i = |X|_{m_i} \tag{2.16}$$

$$X = \left\lfloor \frac{X}{m_i} \right\rfloor m_i + x_i \tag{2.17}$$

2.2.1 Properties of RNS

Some important identities involving residue representations and integers are articulated in equations Eq. 2.18 - Eq. 2.26 [16].

$$|Km|_m = 0 \tag{2.18}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$|X + Km|_m = |X|_m \tag{2.19}$$

$$|Km + 1|_m = 1 \tag{2.20}$$

$$|X + Y|_m = \left||X|_m + |Y|_m\right|_m \tag{2.21}$$

$$|X \cdot Y|_m = \left||X|_m \cdot |Y|_m\right|_m \tag{2.22}$$

$$|-X|_m = |(m-1)X|_m = |m - X|_m \tag{2.23}$$

$$|KX|_{Km} = K\,|X|_m \tag{2.24}$$

$$\text{For X and Y prime, if } |KX|_m = |KY|_m\,,\text{ then } |X|_m = |Y|_m \tag{2.25}$$

$$\text{For m prime, } \ |X^m|_m = |X|_m\,|X|_m = |X|_m \tag{2.26}$$

2.2.2 Dynamic Range and Pairwise relatively prime moduli

The dynamic range of a moduli set defines the interval of values in which an integer has a unique RNS representation. Consider a moduli set of {2,3}. The residue representations of integers 5 ({1,2}) and 11 ({1,2}) are the same. Thus, it is impossible for a number system defined on this moduli set to distinguish between these two numbers. The dynamic range thus defines the legally permissible values of the RNS. For example, if a system uses the moduli set {2,3}, the dynamic range will be 6 because any number in the set [0,6-1] will have a unique residue representation. Concomitant to this definition is an understanding of the periodic properties of RNS. For the set {2,3}, the residue representation is periodic about the dynamic range, i.e., a number i and $i + 6$ will have the same residue representation.

The process of reverse conversion from a residue representation to a unique binary/decimal representation requires a special property of the moduli set, namely,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

that the moduli set be pairwise relatively prime. This means that for any moduli set $\{m_1, m_2...m_N\}$, $GCD(m_i, m_j) = 1$, for $i \neq j$. Given a moduli set, $\{m_1, m_2...m_N\}$ which satisfies the requirement of pairwise relatively prime moduli, the dynamic range of the system is defined as in Eq. 2.27. Most RNS architectures exploit the pairwise relative primality of the moduli set.

$$\text{Dynamic Range}, M = m_n m_{n-1} m_{n-2} \cdots m_1 m_0 \tag{2.27}$$

2.2.3 Common RNS Operations

We can define a set of modulo arithmetic operations $\diamond$ such that in RNS, these operations can be articulated as: $X \diamond Y = \{x_1, ..., x_N\} \diamond \{y_1, ..., y_N\} = \{x_1 \diamond y_1, ..., x_N \diamond y_N\}$. Thus, these operations provide the basis for parallelism in RNS that makes it suitable for fast VLSI implementation. The common arithmetic operations in the set $\diamond$ are addition, subtraction and multiplication. Thus, it can be seen that any application that involves operators which are from the set $\diamond$ and invokes these operations frequently in intermediate processing can be considered as suitable for RNS implementation. Some RNS operations such as division and negation are not efficient and thus applications which involve a lot of these operations are usually not considered for implementation using RNS [16].

2.2.4 Binary/RNS Conversion

The interfacing of RNS to external applications working on the binary representation requires conversion between the two representations. Conversions to/from binary systems have a major impact on any design involving RNS.

2.2.4.1 Forward Conversion

Forward conversion involves transforming a binary number into its component residues. The formulation for forward conversion is given in Eq. 2.16. Forward conversion for a residue channel i is independent of other channels and this conversion is thus parallel. ROM (Read Only Memory) and LUT (Look Up Table) approaches to Forward Conversion [42] were popular before the advent of VLSI technology because of the cost advantages of memory. Most modern implementations rely on logic circuit based architectures [43, 44, 45]. Most forward conversions rely on periodic properties of powers of two taken modulo m to reduce the forward conversion process into multioperand addition [46, 47].

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2.2.4.2 Reverse Conversion:

The earliest known algorithm for reverse conversion from a residue representation to a binary number is the Chinese Remainder Theorem (CRT) as shown in Eq. 2.28 [16]. Here M is the dynamic range and $\hat{m}_j = M/m_j$ and the double bar represents multiplicative inverse computation.

$$|X|_M = \left| \sum_{j=1}^{N} \hat{m}_j \left| \frac{x_j}{\hat{m}_j} \right|_{m_j} \right|_M \tag{2.28}$$

Direct implementation of CRT involves multipliers modulo M (dynamic range) and are thus inefficient for VLSI implementations. Many varieties of reverse converters based on the CRT concepts are known in literature [48, 49, 50, 51]. A recent research known as the new CRT algorithms [52] has thrown up more efficient implementations of reverse converters for the general moduli set. The Mixed Radix Conversion (MRC) is another well-known algorithm in which the mixed-radix form of the integer with respect to the moduli set is used for reverse conversion [53, 54]. As opposed to forward conversion and residue arithmetic calculation, reverse conversion works on all the moduli of the system. For this reason, the selection of moduli set has a great role to play in the design of reverse converters [21].

2.2.5 Special Moduli set

The choice of moduli belonging to some particular sets with known good number theoretic properties make RNS architectures more efficient [55, 24, 11, 43, 6, 56, 57, 58]. Architectures for these special moduli sets are thus a main area of research. The increased efficiency comes about due to the following reasons:

1. Residue arithmetic units like modulo adder and modulo multiplier for special moduli sets can use properties such as End Around Carry (EAC) [6, 11] that cannot be used for general moduli sets. The use of these properties leads to more efficient VLSI implementations of these arithmetic units.

2. Reverse converters for special moduli sets are simpler than those for a general moduli set. This holds for both CRT-based as well as the MRC-based reverse converters. Some special properties of special moduli sets allow these simpler implementations [21].

3. With special moduli sets, it is comparatively easier to choose a moduli

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

set for a given dynamic range. For a general moduli set, such a direct relationship between dynamic range and chosen moduli set is missing.

Most proposed modulus in special moduli sets belong to the category $\{2^n - 1, 2^n, 2^n + 1\}$ [23, 55, 24, 25, 59, 21]. Notable exceptions are [60, 61]. While many varieties of the special moduli sets exist, residue arithmetic units that need to be used in them usually belong to the type $\{2^n - 1, 2^n, 2^n + 1\}$. Consider examples of the following special moduli sets in literature.

1. $2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1$ [26] : The residue arithmetic units used for the $2^{n+1} - 1$ modulus also belongs to the $2^n - 1$ category.

2. $2^n - 1, 2^n, 2^n + 1, 2^{n+1} + 1$ [62] : The residue arithmetic units used for the $2^{n+1} + 1$ modulus also belongs to the $2^n + 1$ category.

3. $2^n - 1, 2^n, 2^n + 1, 2^{2n} + 1$ [24] : The residue arithmetic units used for the $2^{2n} + 1$ modulus also belongs to the $2^n + 1$ category.

It can thus be seen that even with a multitude of special moduli sets, moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ have an overwhelming presence in modern RNS implementations.

2.3 RNS Code Generation and the OO paradigm

The design of RNS units and systems is modular. From the basic units of full and half adders through modulo adders and multipliers to complete RNS systems including forward and reverse converters, architecture definitions can be considered as placing reusable building blocks in a manner suiting system implementation. This concept is shown in Fig. 2.16. This modularity leads up to an additional feature that blocks at a higher levels of hierarchy are more or less independent of implementations of the lower levels. For example, a modulo adder for moduli set $2^n - 1$ using an End Around Carry (EAC) approach [19] can be delineated from the implementation of the CLA Unit where the details of the EAC implementation are considered. The modulo adder only needs to instantiate the CLA unit without being concerned about the exact implementation of the EAC approach.

These features of RNS design suggest the use of a Object Oriented Programming (OOP) approach as a simple and elegant way of generating modulo arithmetic circuit descriptions. The basis of an object oriented approach lies in the encapsulation of data and operations to within classes. These classes are then instantiated as objects. A directly analogous situation for moduli arithmetic units

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

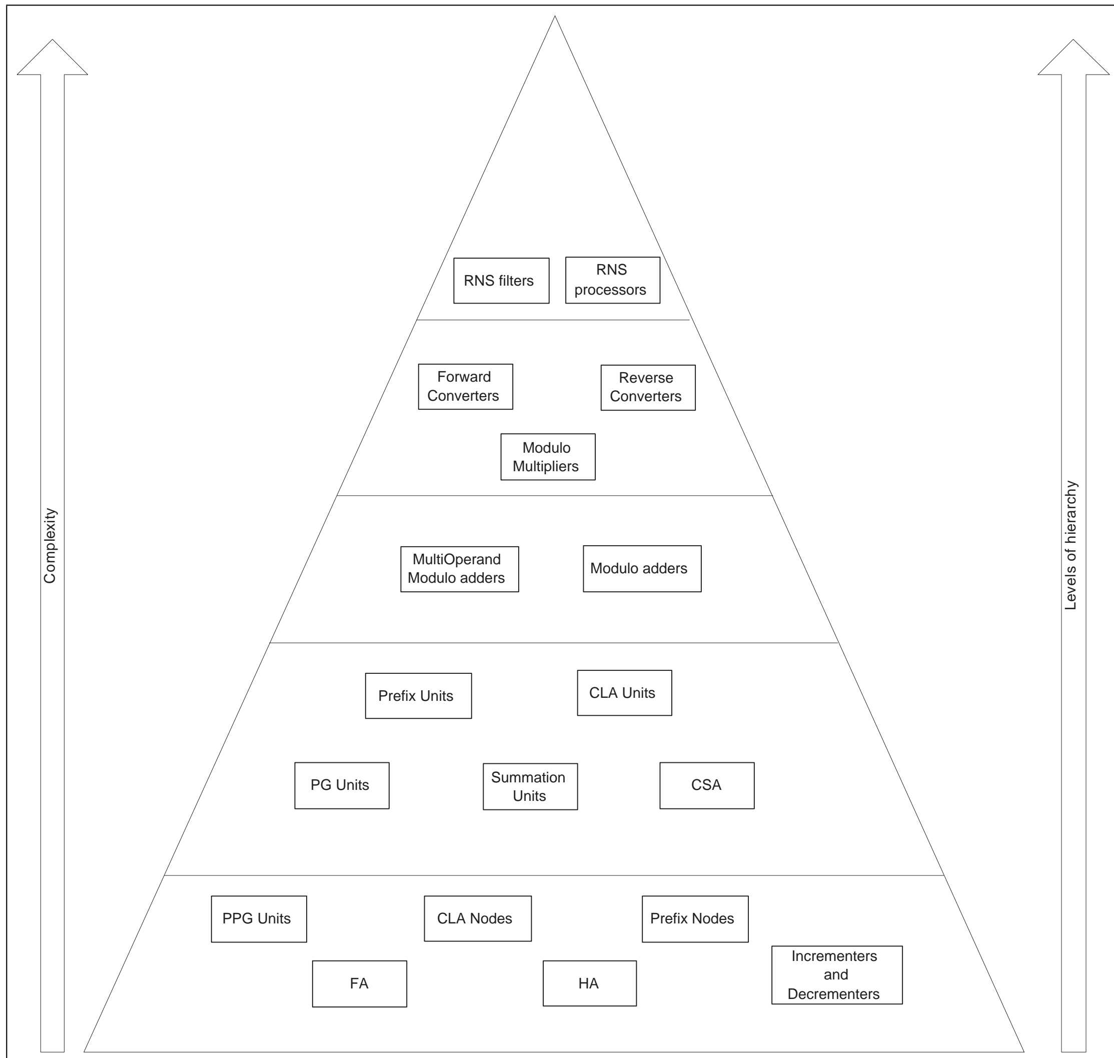

Figure 2.16: Modularity and hierarchy in modulo arithmetic units

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

can be seen, where arithmetic blocks lower down the hierarchy form classes that encapsulate the implementation details. Modulo arithmetic blocks at a higher level instantiate these classes for the creation of larger units. The classes that make up this modular approach for the modulo adders and multipliers are detailed in Chapters 3 and 4

While there are existing code generators for RNS in literature [63, 64, 65], no other code generator that follows the OO approach to generate modulo arithmetic units are known. A code generator (BuRNS) that uses these features has been built using PERL. Following an OOP approach, this code generator generates Verilog HDL (Hardware Description Language) descriptions of modulo arithmetic circuits. The following subsections set out the fundamentals behind the language used to build the code generator (PERL) as well as the language used for circuit descriptions (Verilog).

2.3.1 PERL

PERL, which stands for Practical Extraction and Reporting Language, is a high level interpreted language which has become quite popular as a scripting language [66]. It is a powerful tool for text manipulation and functions associated with text editing. Object oriented programming using PERL relies on making a perl package for individual classes.

2.3.2 Verilog

Verilog is a Hardware Description Language (HDL), which allows the description of digital circuitry at a high level of abstraction [67]. EDA (Electronic Design Automation) tools then take these descriptions and synthesize them to circuit level descriptions.

2.4 Experimental Methodology

While presenting the experimental results, the aim has been to obtain results that will be valid for real world implementation. Thus, the thrust for results has been using industry standard synthesis tools (Synopsys Design compiler [68]) on standard-cell libraries from TSMC (Taiwan Semiconductor Manufacturing Corporation) 0.18μm CMOS process technology. Version 2004.12-SP2 of Synopsys Design Complier was used for reporting the results. Power results were obtained by using switching activity obtained from RTL simulation for a set of 200 random

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

vectors. The switching activity in SAIF (Switching Activity Interchange Format) was annotated for running a power analysis. A switching frequency of 100 MHz (10ns) was used.

The following settings were used for running Design Compiler:

- Library: Library characterizing standard cells for the worst case timing condition (slowest) was used.
- Operating Condition: Operating condition leading to the worst case timing conditions was chosen. The following operating condition represents the worst case condition: Process (1), Voltage (1.62) and Temperature (125°)
- Wire Load Model: A nominal wire load model was used.

The surrounding environment was modeled by using the following settings:

- Input drive strength : Strength of a standard 4X buffer
- Output load : Input of a standard 4X buffer.

The constraints that were used for running synthesis are:

- Maximum delay of 1 ns specified through the combinational logic
- Maximum area of 0 specified
- The option for logic structuring was turned off to prevent the tool from changing the structure of the modulo arithmetic units.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 3

Modulo Adders

Modulo adders are the omnipresent building blocks of RNS. The three basic units as shown in Fig. 1.1 all have modulo adders as their components. As shown in Fig. 1.2, modulo adders stand at the very foundation of RNS architectures. The criticality of modulo adder design thus cannot be overstated. This chapter starts off with a brief discussion on adders for the general moduli set. This is followed by sections that focus on modulo adders belonging to moduli of type $\{2^n-1, 2^n, 2^n+1\}$. The usage of BuRNS for generation of modulo adders is then articulated. Finally, modulo adders for the special moduli set are evaluated using a standard-cell prelayout implementation for the TSMC $0.18\mu m$ process. Some interesting results from this evaluation are then discussed.

3.1 Modulo Addition

Modulo addition adds two modulo m numbers to give a result modulo m. In its most general form, modulo addition can be articulated as in Eq. 3.1 [2, 1]. An alternate representation that is more amenable to hardware implementation is shown in Eq. 3.2 [2, 1]. Architectures for modulo adders can be considered from the perspective of general moduli sets and special moduli sets. Special moduli sets of the type $\{2^n-1, 2^n, 2^n+1\}$ have special properties which can be used for optimizing architectures for modulo adders, while general moduli sets lacking such properties tend to be more involved from the design perspective.

$$|X+Y|_m, X, Y < m = \begin{cases} X+Y & \text{if } X+Y < m \\ X+Y-m & \text{if } X+Y \geq m \end{cases} \tag{3.1}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$|X+Y|_m, X, Y < m = \begin{cases} |X+Y+(2^n-m)|_{2^n} & \text{if } X+Y+(2^n-m) \geq 2^n \\ X+Y & \text{otherwise} \end{cases} \tag{3.2}$$

When VLSI structures for modulo addition are considered, two distinct types of systems can be discerned. Some rely on LUT(Look Up Tables) and ROM(Read Only Memories) that use precomputed values to improve the efficiency of computations. Most papers before the advent of VLSI technology adopted a ROM-based approach [1, 69, 70]. A distinct disadvantage of using a LUT/ROM based approach is the exponential growth in the storage requirement with increasing bit width [4]. The other choice is to use logic circuitry to do all the computations. Improvements in VLSI technology have ensured that logic implementations provide a much more efficient structure than those obtained by using a LUT based approach. Most modern research into modulo adders thus deal exclusively with logic implementations [6, 19, 11, 4, 2, 3].

3.2 Modulo adders for the general moduli set

The most straight forward implementation for modulo addition is defined by Eq. 3.1 and Eq. 3.2. A direct implementation of these equations uses binary adders. Addition of two n-bit numbers X and Y modulo m ($n = \lceil log_2 m \rceil$) can give a result which would fall in one of the following ranges: $[0, m-1], [m, 2^n - 1], [2^n, 2m-2]$. If the result falls in the last two sets, then a $2^n - m$ correction needs to be factored in. This is precisely what the earliest non-ROM based implementations of moduli adders proposed in [1] and [2] accomplished. Fig. 3.1 shows the proposed moduli adders in [1] and [2]. The architecture in [1] does not seem to consider results falling in the range $[2^n, 2m-2]$. This oversight is corrected in [2] by the usage of the OR gate to generate the select signal for the MUX. [2] also suggests multi-cycle adders that use latches for storing intermediate values.

Modulo adders using a CSA approach have been proposed in [3]. The architecture is shown in Fig. 3.2. The input operands are represented in a carry save format. Since a number can have multiple carry save representations, the input operands can thus take a multitude of carry save representation. The multiplexers in the circuit implement conditional correction of $2^n - m$. With the inputs and outputs being in carry save representations, a complete modulo adder will

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

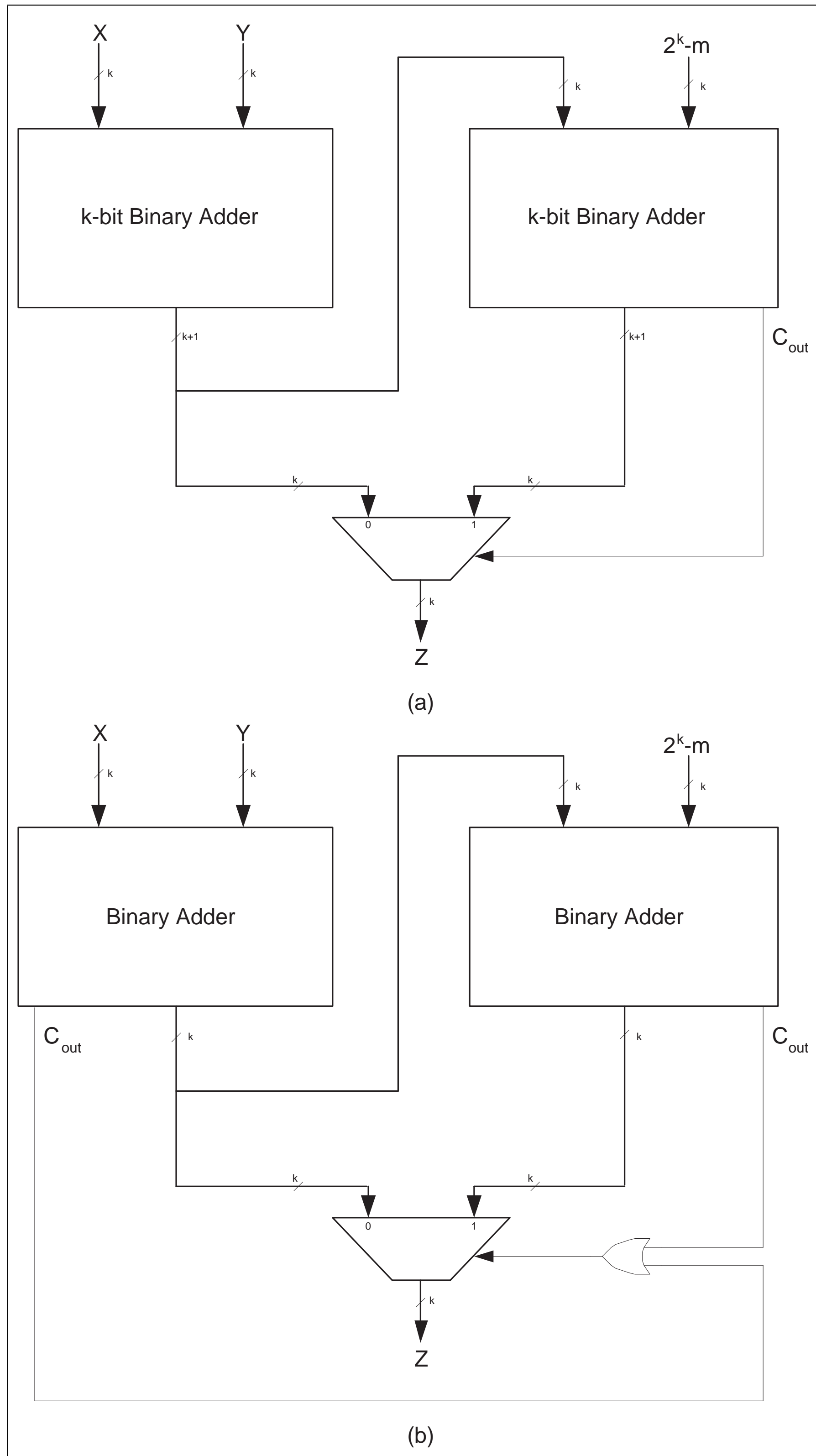

Figure 3.1: Modulo adders using binary adders as proposed in (a)[1] and (b)[2]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

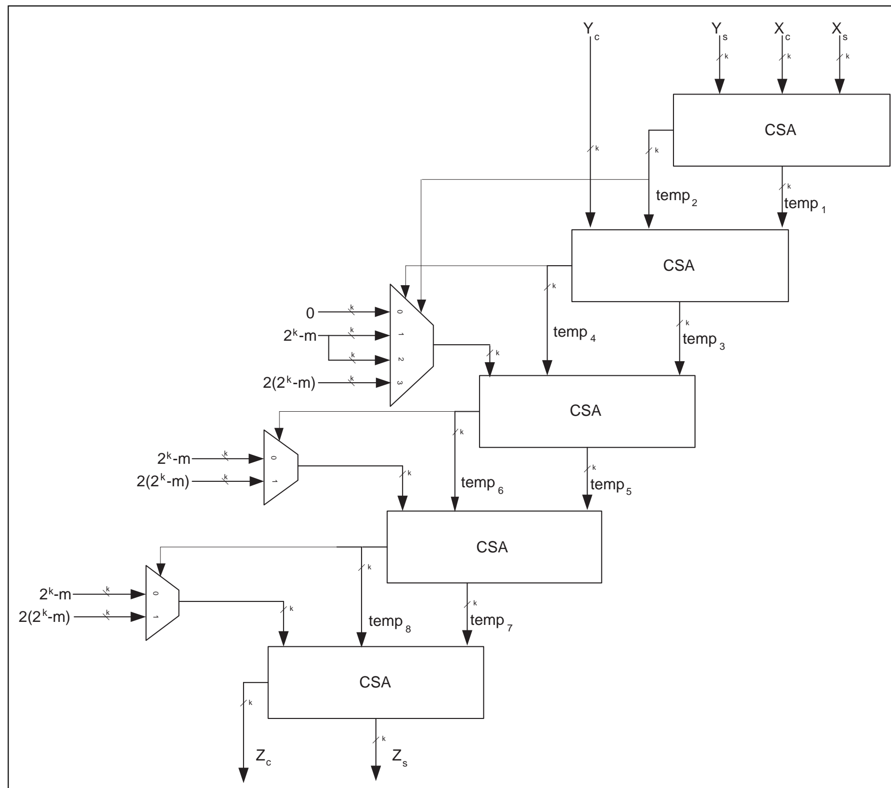

Figure 3.2: Modulo adders using CSA as proposed in [3]

need a final 2-input modulo CPA adder to reduce the output to a single word. This adder has a constant delay that is independent of the moduli.

A hybrid approach that utilizes some CSA concepts and combines them effectively in a CLA scheme has been articulated in [4]. This architecture calculates in parallel only the P/G signal for the sums $X+Y$ and $X+Y+(2^n-m)$. Using these P/G signals the carry output of $X+Y+(2^n-m)$ is computed using a CLA scheme. This carry output then selects the P/G signals which go into a 2-operand CLA adder to accomplish the conditional addition of Eq. 3.2. The effectiveness of the solution is improved by optimizing for particular values for (2^n-m). At the bit positions which are 0's in the binary representation of (2^n-m), $X+Y$ and $X+Y+(2^n-m)$ share common HA structures. For 1's in the binary representation of (2^n-m), an optimized structure called HAL (acronym for Half Adder Like) is used. An example for $m=6$ is shown in Fig. 3.3. The P/G signals in uppercase denote the signals that are calculated for $X+Y+(2^n-m)$ while the lowercase signals denote calculation for $X+Y$. A HAL is used only for bit '1' since the binary representation of 2^n-m in n-bits is (010).

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

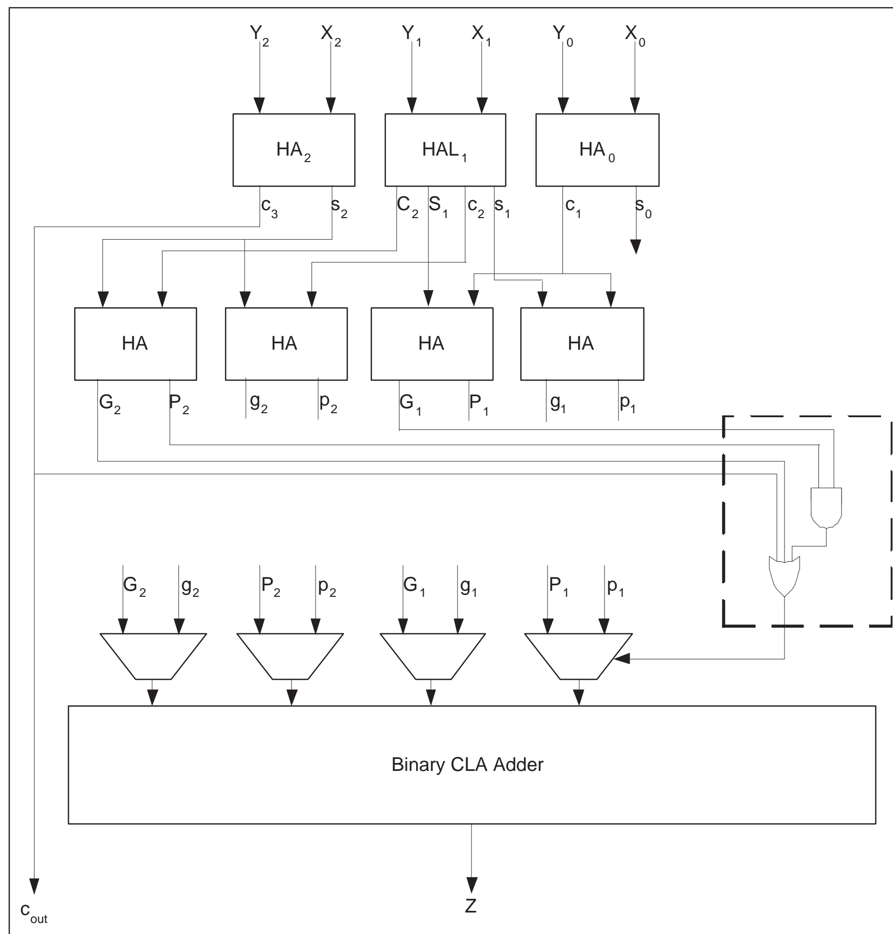

Figure 3.3: Modulo adders as proposed in [4]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.3 Modulo adders for moduli of type $\{2^n-1, 2^n, 2^n+1\}$

Adders for special moduli sets of the type $\{2^n-1, 2^n, 2^n+1\}$ are more efficient that adders for a general moduli set. This is because special moduli sets have properties such as End Around Carry (EAC) [19] and Complemented End Around Carry (CEAC) [6] that can be utilized to efficiently implement modulo adders. For the moduli set 2^n, adder implementation is rather obvious and straightforward in that it involves a normal binary adder with the carry out discarded.

3.4 Modulo adders for moduli of type $2^n - 1$

Modulo adders for moduli sets of type 2^n-1 have been proposed in [19, 11, 5, 71]. The basic improvement for a moduli set of this type comes from the concept of End Around Carry, whereby a modulo 2^n-1 adder can be expressed as a binary addition followed by the addition of the carry out of this operation to the sum in a second cycle of addition. Modulo adders for this type can be divided into implementations relying on CLA [19], parallel prefix [5], Ling [71] or sparse adder [71] schemes. Some implementations also rely on reducing this two-level EAC operation to a single level operation to improve latency [19, 71].

Modulo 2^n-1 addition is expressed as

$$|X+Y|_{2^n-1}, X, Y < 2^n-1 = \begin{cases} X+Y-(2^n-1) & \text{if } X+Y > 2^n-1 \\ X+Y & \text{otherwise} \end{cases} \tag{3.3}$$

The condition $X+Y > 2^n-1$ is detected for an n-bit addition if $c_{out} = 1$. Also, $X+Y-(2^n-1) = |X+Y|_{2^n} + 1$. Thus, modulo 2^n-1 addition can be expressed as

$$|X+Y|_{2^n-1} = |X+Y+c_{out}|_{2^n} \tag{3.4}$$

This equation expresses the EAC concept that forms the basis for efficient modulo 2^n-1 adder implementations. An example of modulo 2^n-1 addition is shown in Fig. 3.4 for $n=3$.

3.4.1 Direct EAC implementations

Direct EAC implementations can take a few forms :

- A combinational feedback of c_{out} as c_{in} on the same binary adders can be

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

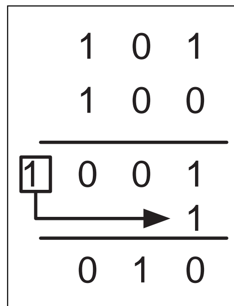

Figure 3.4: Example of EAC modulo $2^n - 1$ adder for $n = 3$

used. This, however is not a very common practice as it causes unwanted race conditions [19].

- A two-cycle addition can be used where the first cycle accomplishes an n-bit binary addition with $c_{in} = 0$. The c_{out} from this addition is then used as c_{in} in the second cycle. This implementation is inefficient in terms of speed owing to the need for two clock cycles to accomplish the addition.

- A binary adder for the first cycle followed by a simple incrementer can be used. However, the additional delay of an incrementer has to be incurred.

- Two adders can be used to compute $X + Y$ and $X + Y + 1$ in parallel. A multiplexer can then be used to select the correct result. This would be a special case of an earlier architecture presented for a general moduli set (Fig. 3.1) as proposed in [1, 2]. This implementation is area inefficient since it requires two binary adders as well as incurs the delay of a MUX.

3.4.1.1 CLA-based implementations

A direct implementation of the EAC concept using CLA adders involves first computing the carry out by reduction of the P/G signals. The carry out is used as a carry input to the other CLA level. An example of a multilevel direct CLA implementation of EAC for n=16 is shown in Fig. 3.5. This implementation needs an extra CLA level (termed the CLAforCout) for computation of the end around carry before the carry computation for other bits. The CLAforCout block constitutes an overhead compared to a binary CLA adder.

3.4.1.2 Prefix adder based implementation

Using prefix adders for a direct implementation of the EAC concept avoids many of the pitfalls seen in the other implementations. This is because the prefix tree

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

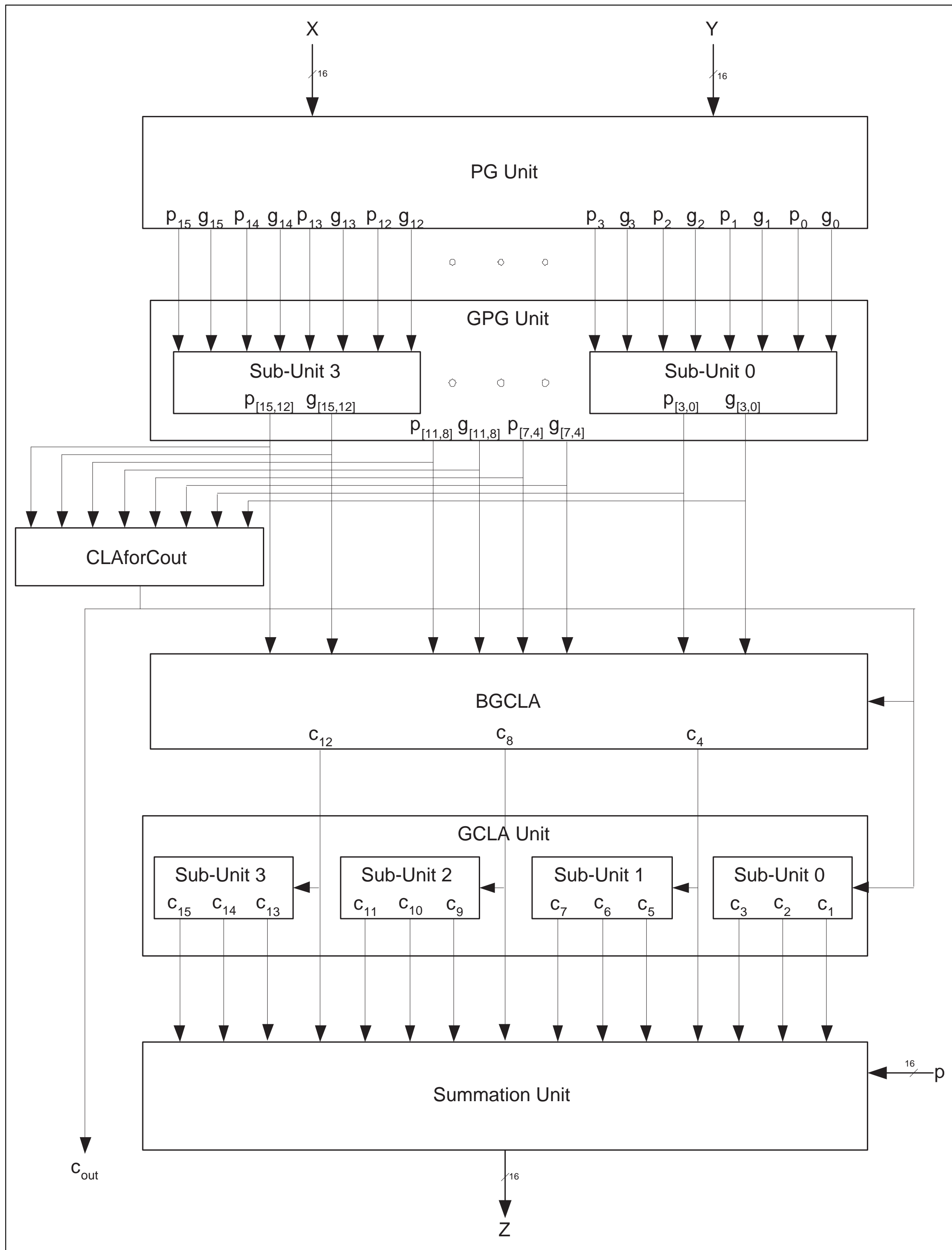

Figure 3.5: Example of direct EAC implementation for a modulo $2^n - 1$ adder for $n = 16$ using multilevel CLA

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

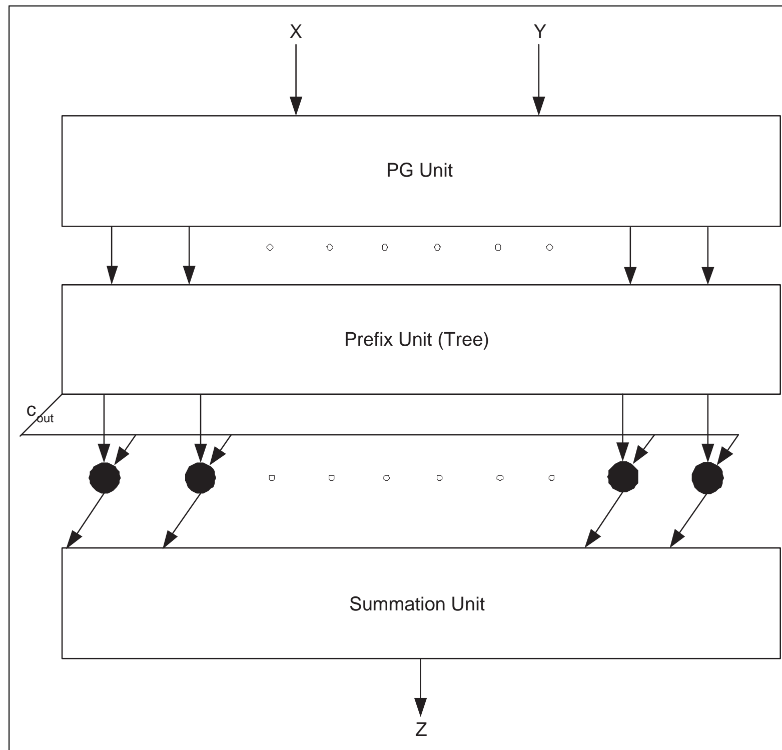

Figure 3.6: Direct EAC implementation using Prefix adders

outputs are already of the form $g_{[k,0]}$. Thus to account for a carry input c_{in} requires only an extra prefix operation for all bits such that the carries for the second cycle $c_{k+1} = (p_{[k,0]}, g_{[k,0]}) \bullet (0, c_0)$, where c_0 is the carry output from the first cycle. The architecture of a parallel prefix adder accomplishing the EAC addition is shown in Fig. 3.6 [11, 28]. The prefix nodes at the incrementing level do not need to compute propagate signals, since only the generate signal outputs from this level are used in the Summation Unit. This suggests that the prefix nodes at the final increment level can be a reduced version of the normal prefix nodes in Fig. 2.7.

3.4.1.3 Sparse adder based implementation

A sparse adder based implementation of modulo $2^n - 1$ addition is easily visualized and has been proposed in [71]. The carry computation unit for the sparse adder needs to compute only the block carries and can implement either a CLA or a prefix scheme. The Carry Select blocks that replace the Summation unit have the same structure as that of a binary sparse adder (Fig. 2.11). The theoretical proof that the same Carry Select blocks can be used is provided in [71] and is not repeated here. The overall structure of a sparse-4 parallel prefix modulo $2^n - 1$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

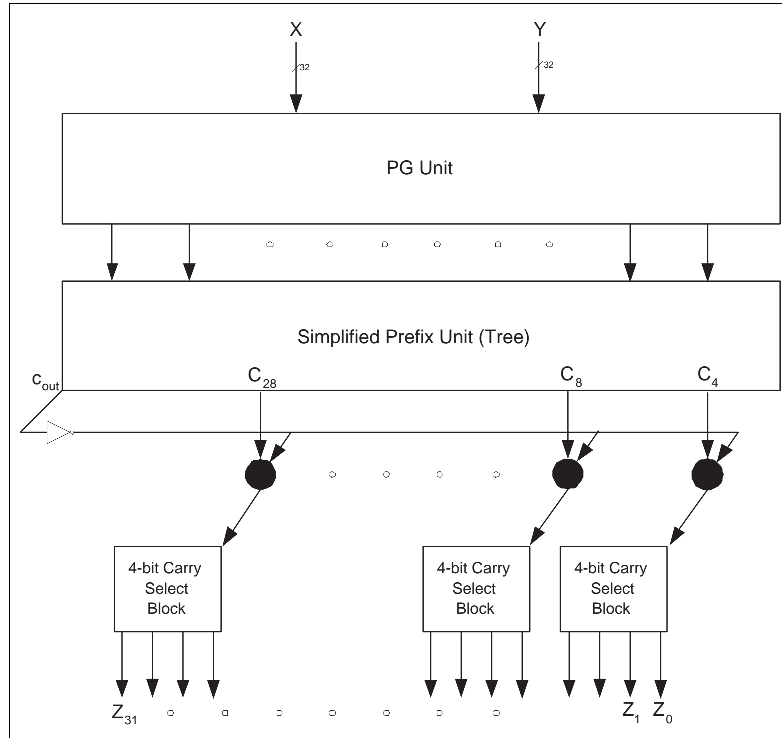

Figure 3.7: Direct EAC implementation using sparse prefix adder

adder is shown in Fig. 3.7.

3.4.2 Implementations based on unwrapping EAC

A class of modulo $2^n - 1$ adders that rely on logic transformations to unwrap the EAC have been proposed in [19, 5]. [19] proposes a scheme for CLA based implementations while [5] proposes a scheme for prefix adders. [71] extends this concept to propose modulo $2^n - 1$ adders based on the sparse adder and Ling adder schemes.

3.4.2.1 CLA-based implementation

For a one-level CLA based implementation, the carry out of the first addition can be represented as : $c_{out} = c_n = g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_0p_1p_2\cdots p_{n-1}$ This carry output is considered the carry in for the second cycle, i.e., for the second cycle, $c_0 = c_n = g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_0p_1p_2\cdots p_{n-1}$ The unwrapping of EAC relies on the usage of the above representation of c_0 in the computation of carries. Conceptually, what it means is that since the carry out of the first cycle depends on the P/G signals which are stable during both the first and second cycles, carry computation in the second cycle can

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

be computed from the P/G signals available without relying on the computation of the carry out signals from the first cycle. With EAC, the carry output for the first bit can be expressed as :

$$\begin{aligned} c_1 &= g_0 + c_0 p_0 \\ &= g_0 + p_0(g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_0 p_1 p_2 \cdots p_{n-1}) \end{aligned} \quad (3.5)$$

Using the logic property $a + ba = a$, this carry bit can be expressed as:

$$c_1 = g_0 + p_0(g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_1 p_2 p_3 \cdots p_{n-1}) \quad (3.6)$$

This process of carry unwrapping is extended to higher bits:

$$\begin{aligned} c_2 &= g_1 + g_0 p_1 + c_0 p_0 p_1 \\ &= g_1 + g_0 p_1 + p_0 p_1 (g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_0 p_1 p_2 \cdots p_{n-1}) \\ &= g_1 + g_0 p_1 + p_0 p_1 (g_{n-1} + g_{n-2}p_{n-1} + g_{n-3}p_{n-2}p_{n-1} + \cdots + g_2 p_3 p_4 \cdots p_{n-1}) \\ \cdots \\ \cdots \end{aligned} \quad (3.7)$$

The difference from the usual CLA equations is immediately obvious. While c_n for normal CLA depends only on the P/G bits of weights lower than n, for a CLA with unwrapped EAC, c_n depends on P/G of all higher weights too. This adds gate complexity to the CLA unit. However, the advantage of such an architecture is also immediately obvious. The computation of the final carries directly from the P/G signals means that no extra gate delay is incurred on the critical path.

Carry unwrapping for a multi-level CLA adder follows a similar structure. Consider a modulo $2^n - 1$ adder using multi-level CLA for n=8 with block size of 4 using EAC unwrapping as shown in Fig. 3.8. The PGUnit and GPGUnit structures are the same as for a normal binary adder. BGCLA is the unit where the EAC unwrapping takes place. Carry out for the first cycle is represented as:

$$c_{out} = c_8 = g_{[7,4]} + g_{[3,0]}p_{[7,4]} = c_0* \quad (3.8)$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

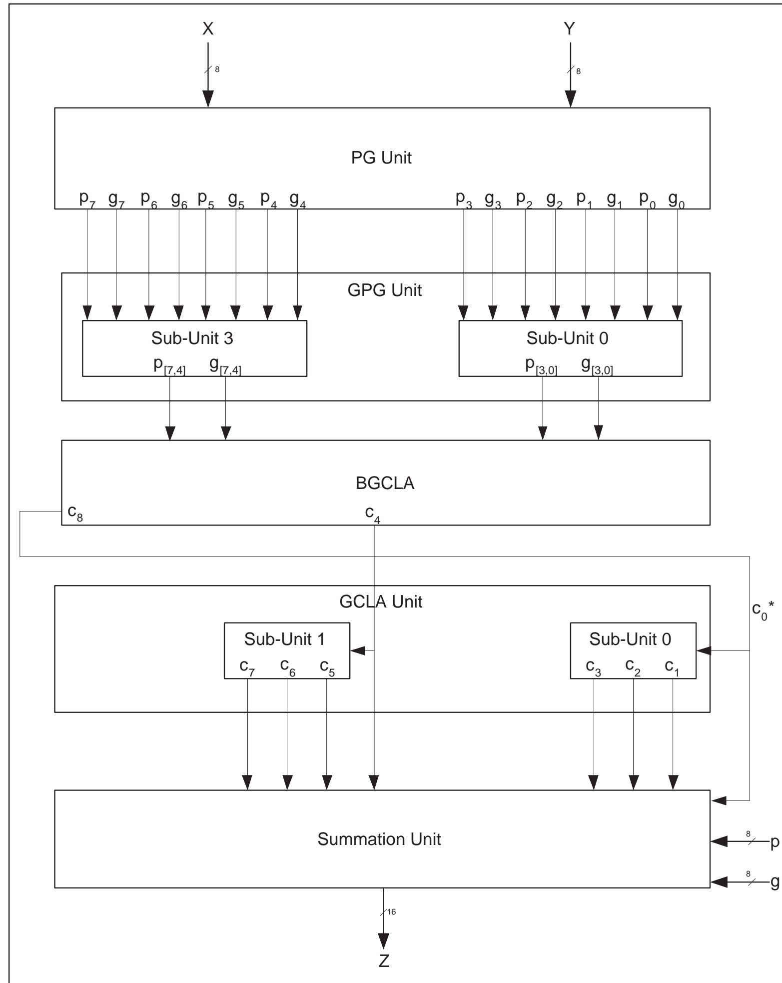

Figure 3.8: Two-level CLA implementation of modulo $2^n - 1$ adder with EAC unwrapping

Unwrapping of EAC in the BGCLA can then be accomplished as follows:

$$c_4 = g_{[3,0]} + p_{[3,0]}c_0^* = g_{[3,0]} + p_{[3,0]}(g_{[7,4]} + g_{[3,0]}p_{[7,4]}) = g_{[3,0]} + p_{[3,0]}g_{[7,4]} \quad (3.9)$$

Once the unwrapping in the BGCLA is accomplished the Group carries can be used for the following GCLA unit. The carry out from BGCLA is used as carry input to the GCLA unit. The GCLA units and the Summation Units work in the same way as for a binary adder.

3.4.2.2 Prefix adder based implementation

A scheme similar to the ones discussed previously for the CLA based adders has been proposed for a parallel-prefix based adder in [5]. Consider a prefix adder for $n = 8$ with EAC. The P/G signals generated by the prefix tree during the first

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

cycle are:

$$\begin{aligned}
(p_{[0,0]}, g_{[0,0]}) &= (p_0, g_0) \\
(p_{[1,0]}, g_{[1,0]}) &= (p_1 p_0, g_1 + g_0 p_1) \\
&\cdots \\
(p_{[7,0]}, g_{[7,0]}) &= (p_7 p_6 \cdots p_0, g_7 + g_6 p_7 + \cdots + g_0 p_1 \cdots p_0)
\end{aligned} \tag{3.10}$$

Expressed as a prefix operation,

$$\begin{aligned}
(p_{[0,0]}, g_{[0,0]}) &= (p_0, g_0) \\
(p_{[1,0]}, g_{[1,0]}) &= (p_1, g_1) \bullet (p_{[0,0]}, g_{[0,0]}) \\
&\cdots \\
(p_{[7,0]}, g_{[7,0]}) &= (p_7, g_7) \bullet (p_{[6,0]}, g_{[6,0]})
\end{aligned} \tag{3.11}$$

During the second cycle, the carry out from the first cycle is considered the carry input, i.e. $(p_0^*, g_0^*) = (p_{[7,0]}, g_{[7,0]})$. The P/G computation for the second cycle can be unwrapped as shown in the following equation:

$$(p_{[0,0]}, g_{[0,0]}) = (p_0, g_0) \bullet (p_0^*, g_0^*) = (p_0, g_0) \bullet (p_7, g_7) \bullet (p_6, g_6) \cdots (p_0, g_0) \tag{3.12}$$

Using the property $(p_i, g_i) \bullet (p_k, g_k) \bullet (p_i, g_i) = (p_i, g_i) \bullet (p_k, g_k)$, Eq. 3.12 can be expressed as:

$$(p_{[0,0]}, g_{[0,0]}) = (p_0, g_0) \bullet (p_7, g_7) \bullet (p_6, g_6) \cdots (p_1, g_1) \tag{3.13}$$

A similar unwrapping is done for the rest of the P/G signals:

$$(p_{[1,0]}, g_{[1,0]}) = (p_1, g_1) \bullet (p_0, g_0) \bullet (p_7, g_7) \bullet (p_6, g_6) \cdots (p_2, g_2) \tag{3.14}$$

It can be seen that the prefix tree implementation with unwrapped EAC now has the condition where the P/G signals of weight i depend on all the other P/G signals instead of only P/G signals of lower weights as in a binary adder. A typical example of a prefix tree that can be used to implement the unwrapped EAC equations is shown in Fig. 3.9. It can be seen that unwrapping the EAC removes an extra level of prefix nodes from the critical timing path of the design while adding gate and wiring complexity to the prefix tree itself.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

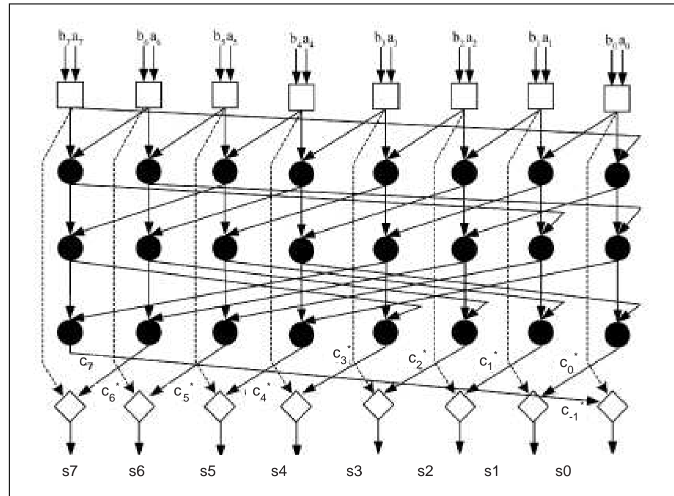

Figure 3.9: Prefix tree for a modulo $2^n - 1$ adder using unwrapped EAC [5]

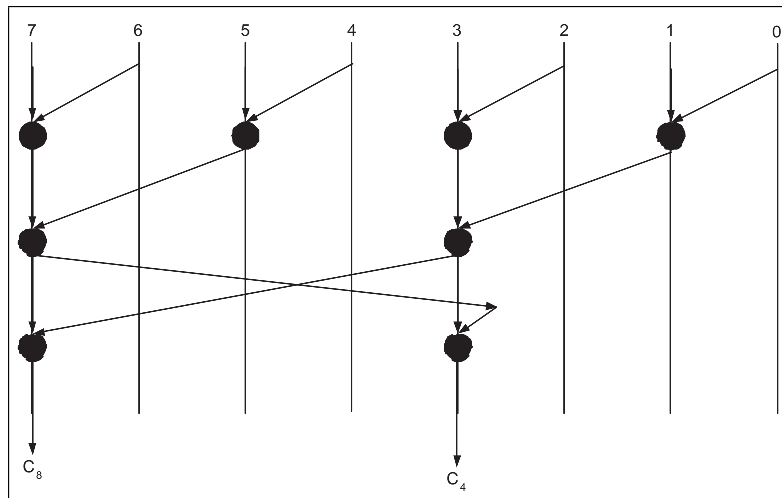

Figure 3.10: Reduced Prefix Tree for a sparse adder

3.4.2.3 Sparse adder based implementation

A sparse adder based implementation for EAC unwrapped modulo $2^n - 1$ adder was proposed in [71]. The structure of such a adder is very similar to its binary counterpart shown in Fig. 2.10. The same Carry Select block as its binary counterpart can be used as was proven in [71]. An example of a reduced prefix tree for the sparse modulo $2^n - 1$ adder, which computes carries at the boundaries of blocks using unwrapped EAC is shown in Fig. 3.10.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.4.2.4 Ling adder based implementation

A Ling adder based implementation for EAC unwrapped modulo $2^n - 1$ adder was proposed in [71]. The structure for this architecture is based on the binary adders proposed in [37]. The derivation of the prefix structure for this architecture relies on the definition of the terms G_i^* and T_i^*:

$$\begin{aligned} G_i^* &= g_i + g_{i-1} \\ T_i^* &= t_i t_{i-1} \end{aligned} \tag{3.15}$$

where $g_{-1} = g_{n-1}$ and $t_{-1} = t_{n-1}$. By using these terms, [71] has derived the equivalent of Ling carries for modulo $2^n - 1$ adders as :

$$H_i = (G_i^*, T_{i-1}^*) \bullet (G_{i-2}^*, T_{i-3}^*) \bullet \ldots (G_{i+2}^*, T_{i+1}^*) \tag{3.16}$$

Compared to the propagation of the normal carries, the propagation of Ling carry terms H_i in this case leads to one level less in the prefix unit. The PGUnit has to be modified to generate the terms of Eq. 3.15. Additionally, the Summation unit is more complex as the summation term to be computed based on the propagated Ling carries is:

$$s_i = \bar{H}_{i-1} h_i + H_{i-1}(h_i \oplus p_{i-1}) \tag{3.17}$$

The structure for the adder thus proposed in [37] is shown in Fig. 3.11.

3.4.3 Dual-representation of zero

For moduli of type $2^n - 1$, there are two ways to represent the value 0.

- Dual representation : In this representation, the value 0 is represented in two forms, one of which is the standard all zero's representation. Additionally, an all 1's representation (which denotes the $2^n - 1$ number) is also considered to represent zero. Some applications involving error detection and coding [72] prefer the dual representation for modulo $2^n - 1$ numbers

- Standard Representation : This is the standard representation where 0 can have only an all zero's representation.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

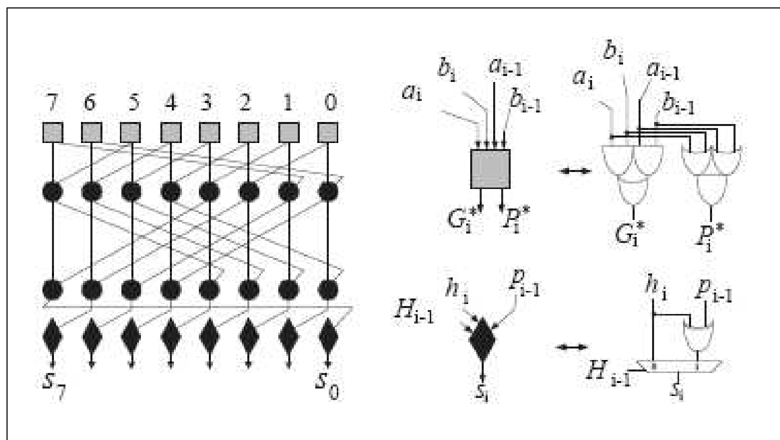

Figure 3.11: Ling modulo $2^n - 1$ adder

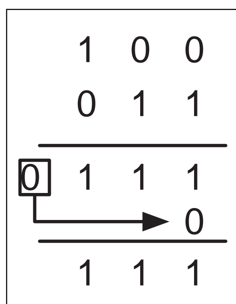

Figure 3.12: EAC leading to dual representation in modulo $2^n - 1$ addition : An example

Any direct implementation of the EAC gives a dual representation of zero. An example of a modulo $2^n - 1$ addition that can give an all 1's representation of the result is shown in Fig. 3.12. If the dual representation is to be avoided, then some minor modifications to the adder need to be effected. These modifications rely on detecting inputs which will give a all 1's representation at the output. Such inputs are detected by 'AND'ing the propagate signals together. Once such a combination is detected, an all 0's value is output. The scheme for detecting and avoiding dual representation of zero is shown in Fig. 3.13.

3.5 Modulo Adders for type $2^n + 1$

Modulo adders for moduli of type $2^n + 1$ can be discerned as belonging to a few distinct types. Firstly, there are the modulo adders that deal with the normal representation of modulo $2^n + 1$ numbers [4, 11, 73]. Alternately, there are modulo adders that deal with the diminished-one representation of modulo $2^n + 1$ numbers

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

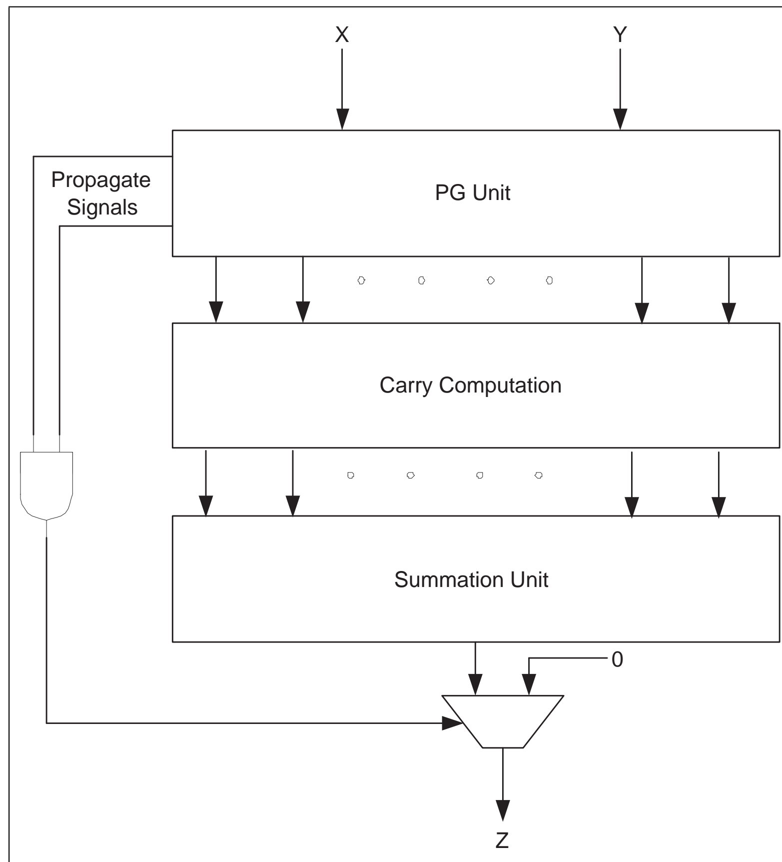

Figure 3.13: Solving the problem of dual representation of zero for modulo $2^n - 1$ adders

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[6, 11]. Some modulo $2^n + 1$ adders rely on a slightly modified diminished-one representation [7]

Diminished-one system for the representation of Fermat numbers was first proposed in [74] and is now widely used in architectures for moduli of type $2^n + 1$ [6, 11]. In the diminished-one system, the numbers to be operated upon are decremented before being input to the modulo $2^n + 1$ system. The bitwidth for the inputs thus reduce from the original $(n + 1)$-bits for a number in the range $[0,\ 2^n]$ to n-bits for a number in the range $[0,\ 2^n - 1]$. However, the conversion between a modulo $2^n + 1$ number and a diminished-one representation requires incrementing and decrementing operations, which might prove to be unacceptable overheads in some systems.

The basic formulation of a modulo $2^n + 1$ addition is:

$$|(X + Y + 1)|_{2^n+1}\,, X, Y < 2^n + 1 = \begin{cases} X + Y + 1 - (2^n + 1) & \text{if } X + Y > 2^n \\ X + Y + 1 & \text{otherwise} \end{cases} \tag{3.18}$$

With the observation that $X + Y + 1 - (2^n + 1) = X + Y - 2^n = |X + Y|_{2^n}$, Eq. 3.18 can be transformed to:

$$|(X + Y + 1)|_{2^n+1}\,, X, Y < 2^n + 1 = \begin{cases} |X + Y|_{2^n} & \text{if } X + Y > 2^n \\ X + Y + 1 & \text{otherwise} \end{cases} \tag{3.19}$$

Eq. 3.19 specifies that if the sum $X+Y$ exceeds 2^n, then the modulo sum is the modulo 2^n addition $|X + Y|_{2^n}$. Otherwise, an incrementing operation will give the correct result. This can be alternatively expressed as : $|X + Y + 1|_{2^n+1} = |X + Y + \bar{c}_{out}|_{2^n}$. The addition of the inverted carry out is similar to the EAC operation for the modulo $2^n - 1$ adders and is known as Complemented End Around Carry (CEAC). An example of a calculation of modulo addition using the CEAC concept is shown in Fig. 3.14.

An important observation here is that all modulo 2^n+1 adders using CEAC to compute the outputs according to Eq. 3.19 actually compute the incremented sum $X + Y + 1$. For the diminished-one representation, a final incrementing operation is required to obtain a modulo $2^n + 1$ number. There are some applications, however, which can work with the result from the modulo adder directly [11].

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

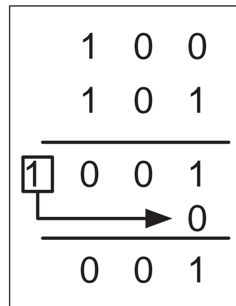

Figure 3.14: Modulo $2^n + 1$ addition using CEAC : An example

3.6 Modulo $2^n + 1$ Adders for diminished-one representation

Modulo $2^n + 1$ addition with a diminished-one representation can be articulated as:

$$\begin{aligned} \text{Diminished-one representation: } & X' = X - 1; Y' = Y - 1 \\ \text{Modulo } 2^n + 1 \text{ addition: } & Z' = |X' + Y' + 1|_{2^n+1} \\ & Z = Z' + 1 \end{aligned} \tag{3.20}$$

The general structure of a complete modulo 2^n+1 adder using the diminished-one representation is shown in Fig. 3.15.

3.6.1 Direct CEAC implementations

Direct implementation of CEAC structures can be based on prefix adders [11] or CLA. A prefix structure with an extra level of prefix operators for the CEAC operations has been proposed in [11] and is shown in Fig. 3.16. It can be seen that this architecture is very similar to one used for modulo $2^n - 1$ addition, the main difference being that that the end-around carry is inverted for the case of a modulo $2^n + 1$ adder. Overhead of an extra level of prefix nodes is incurred over the binary adders.

CLA based implementation of CEAC is quite straightforward and involves an extra unit called the CLAforCout that computes the CEAC which is input to the rest of the carry computation blocks as a carry input. An example of a 16-bit modulo 2^n+1 adder using a multilevel CLA with direct implementation of CEAC is shown in Fig. 3.17.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

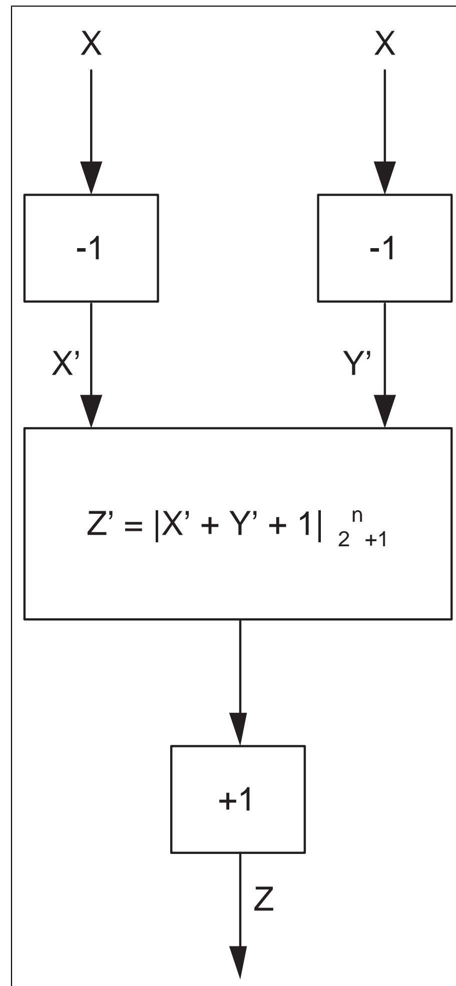

Figure 3.15: General structure of a modulo $2^n + 1$ adder using the diminished-one representation

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

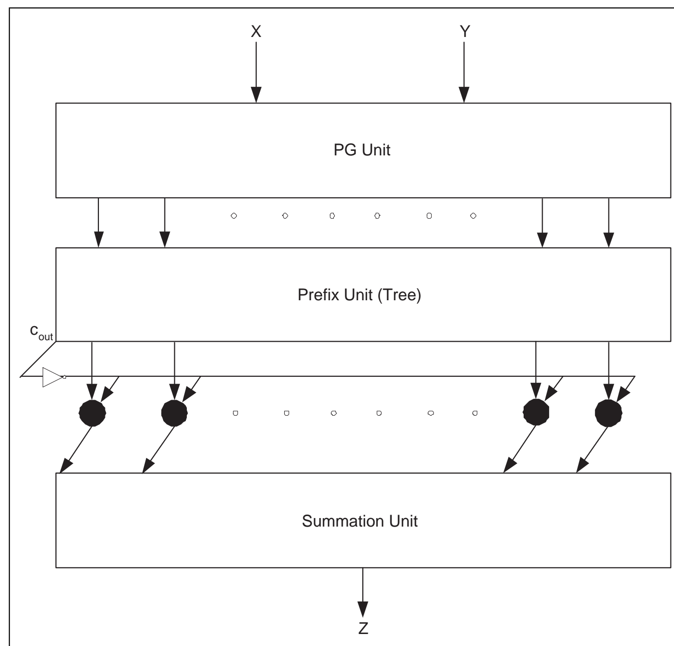

Figure 3.16: Prefix based modulo $2^n + 1$ adder with direct CEAC implementation

3.6.2 Implementations based on unwrapping CEAC

An improved architecture for modulo $2^n + 1$ addition has been proposed where the CEAC is unwrapped [6]. The designs proposed in [6] encompass single-level CLA, multi-level CLA as well as parallel prefix based adders.

3.6.2.1 CLA based implementation

The CEAC operation can be represented as adding the inverted carry output of the first cycle c_{out} as the carry input of the second cycle, c_0^*. Since the carry out of the first cycle depends solely on the P/G signals, it signifies that the carry input for the second cycle can be computed from the stable P/G signals available in the first cycle. The usage of c_0^* in a form consisting of P/G terms leads to the unwrapping of the carry terms of the second cycle.

Using the lemma $\bar{c}_{k+1} = \bar{p}_k + \bar{g}_k\bar{c}_k$, the proof of which can be obtained in [6], an example of the equations for a unwrapped CEAC implementation for a single-level CLA with $n = 3$ is shown in Eq. 3.21. The unwrapping equations use the property $g_0 + p_0\bar{g}_{n-1}\cdots\bar{g}_1\bar{g}_0 = g_0 + p_0\bar{g}_{n-1}\cdots\bar{g}_1$. As can be seen from these equations, with the CEAC unwrapped, the carry outs depend on P/G from all the bits and not just the bits lower in weight. While this increases the complexity of the CLA unit, it does not add any extra gate delay on the carry path.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

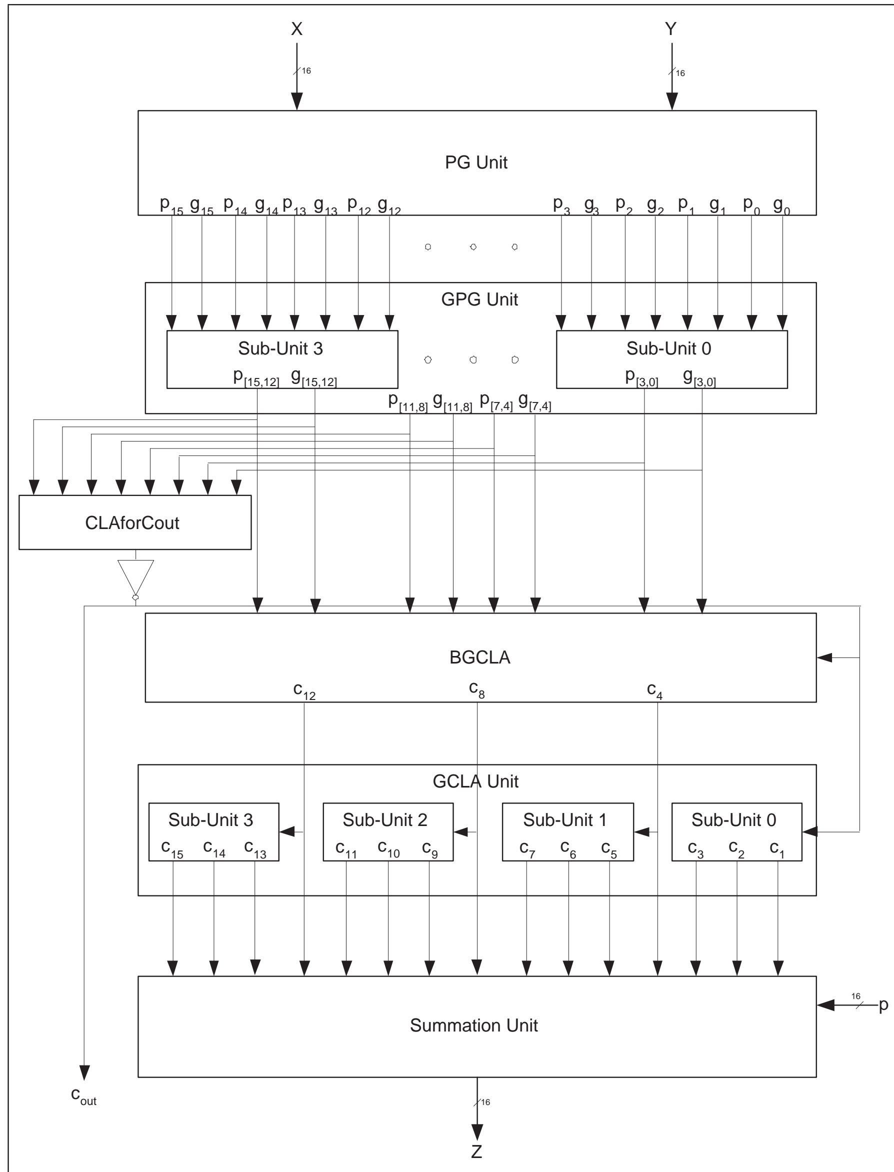

Figure 3.17: CLA based modulo $2^n + 1$ adder with direct CEAC implementation

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$
\begin{aligned}
\bar{c}_{out} &= \bar{c}_3 = \bar{p}_2 + \bar{g}_2\bar{c}_2 \\
&= \bar{p}_2 + \bar{g}_2(\bar{p}_1 + \bar{g}_1\bar{c}_1) \\
&= \bar{p}_2 + \bar{g}_2\bar{p}_1 + \bar{g}_2\bar{g}_1(\bar{p}_0 + \bar{g}_0) \\
&= \bar{p}_2 + \bar{g}_2\bar{p}_1 + \bar{g}_2\bar{g}_1\bar{p}_0 + \bar{g}_2\bar{g}_1\bar{g}_0 \\
c_0^* &= \bar{c}_{out} \\
c_1^* &= g_0 + p_0c_0^* = g_0 + p_0(\bar{p}_2 + \bar{g}_2\bar{p}_1 + \bar{g}_2\bar{g}_1\bar{p}_0 + \bar{g}_2\bar{g}_1\bar{g}_0) \\
c_1^* &= g_0 + p_0\bar{p}_2 + p_0\bar{g}_2\bar{p}_1 + p_0\bar{g}_2\bar{g}_1\bar{p}_0 + p_0\bar{g}_2\bar{g}_1\bar{g}_0 \\
&= g_0 + p_0\bar{p}_2 + p_0\bar{g}_2\bar{p}_1 + p_0\bar{g}_2\bar{g}_1\bar{g}_0 \\
&= g_0 + p_0\bar{p}_2 + p_0\bar{g}_2\bar{p}_1 + p_0\bar{g}_2\bar{g}_1 \\
c_2^* &= g_1 + p_1c_1^* = g_1 + p_1g_0 + p_1p_0\bar{p}_2 + p_1p_0\bar{g}_2\bar{p}_1 + p_1p_0\bar{g}_2\bar{g}_1 \\
&= g_1 + p_1g_0 + p_1p_0\bar{p}_2 + p_1p_0\bar{g}_2
\end{aligned}
\tag{3.21}
$$

For a multi-level CLA, CEAC unwrapping basically boils down to modifying the BGCLA unit to generate the Between Groups carry signals that are unwrapped using the CEAC property. The following units require no modifications. The carry out computed from the BGCLA is fed in at the carry input for the GCLA unit. An example of a two-level CLA with CEAC for $n = 8$ is shown in Fig. 3.18. The BGCLA equation that implements the unwrapping is shown in Eq. 3.22.

$$
\begin{aligned}
c_4 &= g_{[3,0]} + p_{[3,0]}c_0^* \\
&= g_{[3,0]} + p_{[3,0]}(p_{[\bar{7},4]} + p_{[\bar{3},0]}g_{[\bar{7},4]} + g_{[\bar{3},0]}) \\
&= g_{[3,0]} + p_{[3,0]}p_{[\bar{7},4]} + p_{[3,0]}p_{[\bar{3},0]}g_{[\bar{7},4]} + p_{[3,0]}g_{[\bar{3},0]} \\
&= g_{[3,0]} + p_{[3,0]}p_{[\bar{7},4]} + p_{[3,0]} \\
c_8 &= c_0^* = p_{[\bar{7},4]} + p_{[\bar{3},0]}g_{[\bar{7},4]} + g_{[\bar{3},0]}
\end{aligned}
\tag{3.22}
$$

3.6.3 Prefix Adders with CEAC Unwrapped

For the derivation of equations pertaining to parallel prefix operation with the CEAC for modulo $2^n + 1$ addition, the following theorems need to be set out. The proof for these theorems can be found in [6] and are not repeated here.

Theorem 3.1 *Let* $\overline{(p_{[i,0]}, g_{[i,0]})} = (p_{[i,0]}, \overline{g_{[i,0]}})$. *Then the carries generated during the second cycle are given by* $(p^*_{[i,0]}, g^*_{[i,0]}) = (p_{[i,0]}, g_{[i,0]}) \bullet \overline{(p^*_{[n-1,i+1]}, g^*_{[n-1,i+1]})} =$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

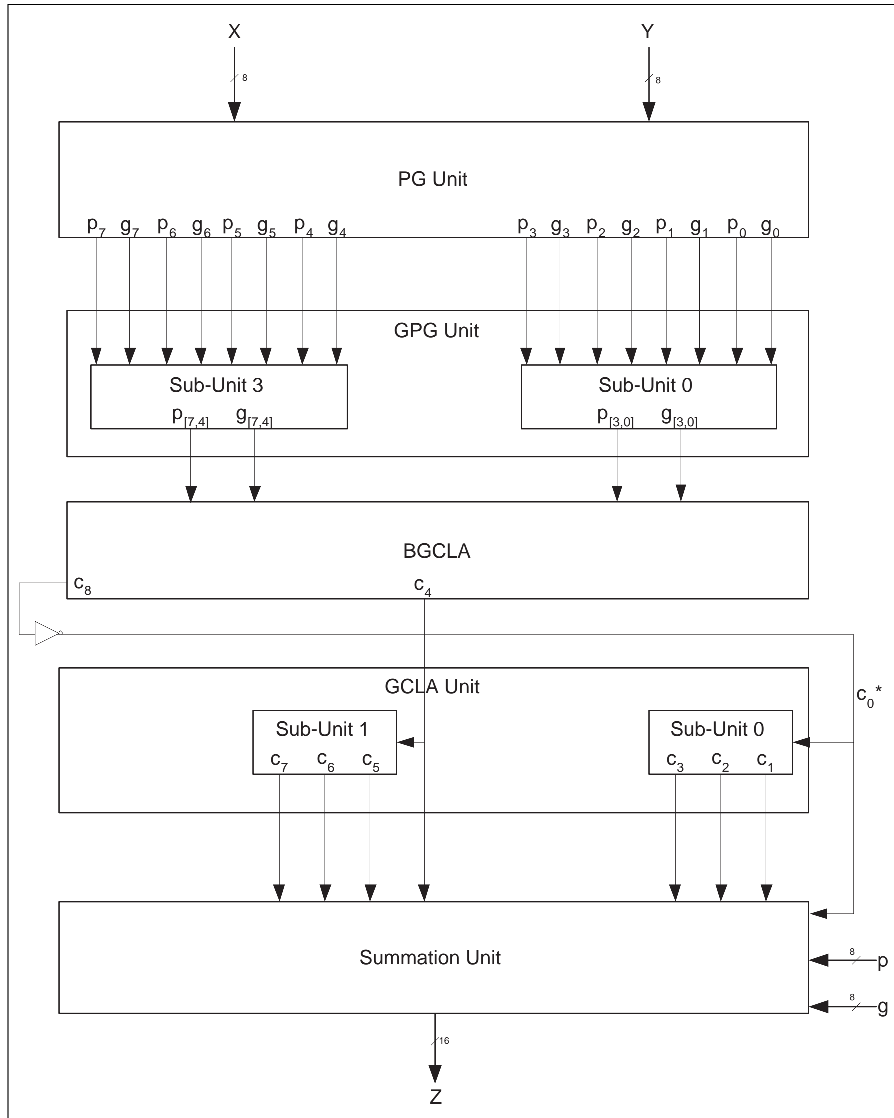

Figure 3.18: A modulo $2^n + 1$ adder using two-level CLA with CEAC unwrapped for n=8

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$(p_{[i,0]}, g_{[i,0]}) \bullet (p^*_{[n-1,i+1]}, \overline{g^*_{[n-1,i+1]}})$

Theorem 3.2 *If* $(p_x, g_x) = (p_i, g_i) \bullet \overline{(p_{[a,b]}, g_{[a,b]})}$, *and* $(p_y, g_y) = \overline{(\bar{g}_i, \bar{p}_i) \bullet (p_{[a,b]}, g_{[a,b]})}$, *then* $g_x = g_y$.

Using these two theorems, the equations in Eq. 3.23 have been derived for the prefix tree structure for an adder with $n = 8$ in [6]. The articulation of unwrapped prefix operation as in Eq. 3.23 ensures that the prefix tree for these equations can be implemented with $log_2 n$ levels with some modification to the prefix structures. The prefix tree for $n = 8$ is shown in Fig. 3.19.

$$
\begin{aligned}
c_0^* &= \overline{(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)\bullet(p_1,g_1)\bullet(p_0,g_0)}\\
c_1^* &= (p_0,g_0)\bullet\overline{(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)\bullet(p_1,g_1)}\\
&= \overline{(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)\bullet(p_1,g_1)}\\
c_2^* &= (p_1,g_1)\bullet(p_0,g_0)\bullet\overline{(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)}\\
&= \overline{(\bar{g}_1,\bar{p}_1)\bullet(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)}\\
c_3^* &= (p_2,g_2)\bullet(p_1,g_1)\bullet(p_0,g_0)\bullet\overline{(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)}\\
&= \overline{(\bar{g}_2,\bar{p}_2)\bullet(\bar{g}_1,\bar{p}_1)\bullet(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)}\\
c_4^* &= (p_3,g_3)\bullet(p_2,g_2)\bullet(p_1,g_1)\bullet(p_0,g_0)\bullet\overline{(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)}\\
c_5^* &= (p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)\bullet(p_1,g_1)\bullet\overline{(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)\bullet(p_6,g_6)\bullet(p_5,g_5)}\\
c_6^* &= (p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet(p_2,g_2)\bullet\overline{(\bar{g}_1,\bar{p}_1)\bullet(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)\bullet(p_6,g_6)}\\
c_7^* &= (p_6,g_6)\bullet(p_5,g_5)\bullet(p_4,g_4)\bullet(p_3,g_3)\bullet\overline{(\bar{g}_2,\bar{p}_2)\bullet(\bar{g}_1,\bar{p}_1)\bullet(\bar{g}_0,\bar{p}_0)\bullet(p_7,g_7)}
\end{aligned}
\tag{3.23}
$$

3.7 Modulo $2^n + 1$ Adders for normal representation

The main advantage of using a normal representation for a standalone modulo $2^n + 1$ addition is that the usage of incrementers and decrementers of the diminished-one representation can be avoided. Modulo adders for the normal representation are proposed in [4, 11, 75, 73]. The approach in [11] involves the usage of a CEAC adder and correction units to detect the values 2^n which is very similar to the approach used for a diminished-one modulo adder.

The modulo adder cited in [4] relies on a articulation of modulo 2^n+1 addition slightly different from that of Eq. 3.18. Modulo 2^n+1 addition in [4] is represented

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

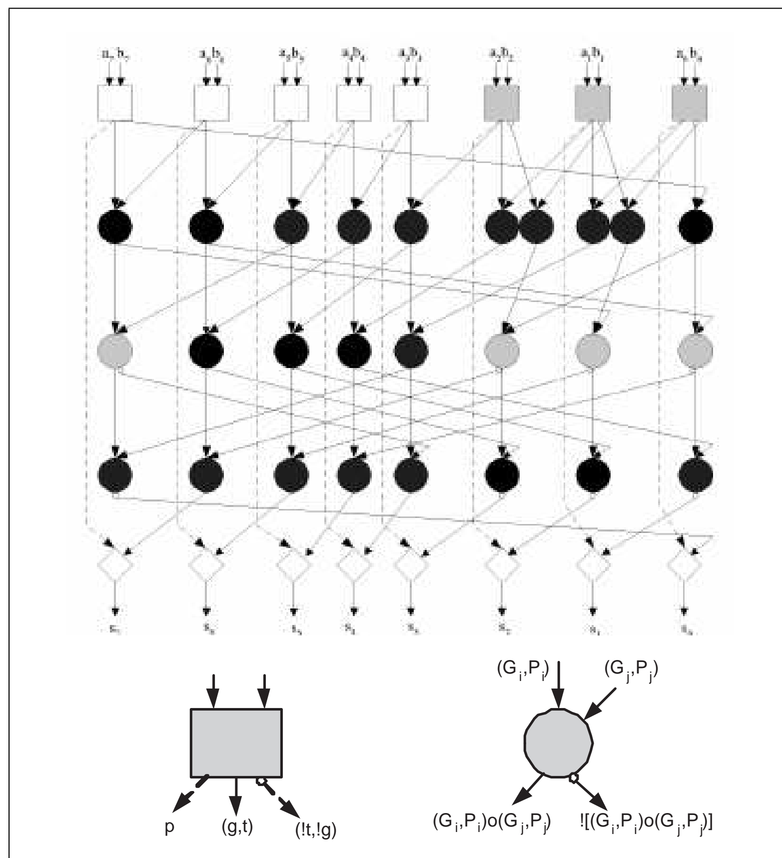

Figure 3.19: Prefix tree using modified prefix operators for a $2^n + 1$ adder with CEAC unwrapped for n=8 [6]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

as in Eq. 3.24, where $Z = 2^n - 1$. It can be seen that the implementation of modulo $2^n + 1$ adder in [4] is a special case of the adder for the general moduli proposed by the same author [4].

$$|(X+Y)|_{2^n+1} = \begin{cases} |X+Y+Z|_{2^n} & \text{if } X+Y+Z \geq 2^{n+1} \\ |X+Y+Z|_{2^n+1} & \text{otherwise} \end{cases} \tag{3.24}$$

A modulo 2^n+1 adder as proposed in [4] has a structure as shown in Fig. 3.20. The Sum And Carry Unit (SAC Unit) generates the carry-save representation of $X + Y + (2^n - 1)$. These two operands are then fed to a $n + 2$-operand CEAC adder. The MSB of the output Z_n is obtained from the propagate signals directly. The width of the adder used is $n + 2$. The CEAC adder in this case can be of any of the types discussed in the previous sections:

- CLA adder with direct implementation of CEAC
- Prefix adder with direct implementation of CEAC
- CLA adder with unwrapped CEAC implementation
- Prefix adder with unwrapped CEAC implementation

Two flavours of modulo $2^n + 1$ adders for the normal representation using parallel-prefix schemes have been proposed in [73]. The first one relies on usage of a CSA and a n-bit parallel adder to reduce the latency of the carry incrementing stage. This architecture has been termed the Parallel-Prefix with Fast Carry Increment (PPFCI). The second unwraps the reentering carry to achieve better latency and has been termed the Totally Parallel-Prefix (TPP) adder. The following subsections discuss these adders.

3.7.1 PPFCI

The formulation of this architecture depends upon the expression of a modulo $2^n + 1$ addition in the form:

$$|(X+Y)|_{2^n+1} = \begin{cases} ||X+Y+2^n-1|_{2^n+1} + 2^n + 1|_{2^n+1} & \text{if } X+Y+2^n-1 < 2^{n+1} \\ |X+Y+2^n-1|_{2^n+1} & \text{otherwise} \end{cases} \tag{3.25}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

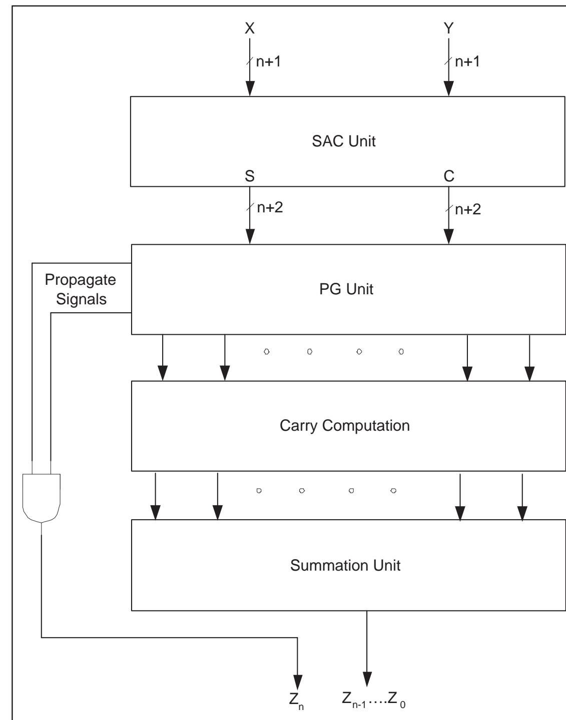

Figure 3.20: Modulo $2^n + 1$ addition for the normal representation

This formulation implies a modulo adder architecture with a stage for computation of $|X + Y + 2^n - 1|_{2^n+1}$ and a second stage for a conditional addition of the term $2^n + 1$. The PPFCI architecture proposes the usage of a CSA and a parallel-prefix two operand adder for the first stage. Four main modifications from this basic architecture leads to a fast carry increment stage:

1. The usage of a inclusive-NOR gate at the MSB of the first stage to generate the CEAC.

2. The direct usage of the LSB sum generated from the CSA in the Summation unit of the parallel-prefix adder

3. The usage of an AND gate in the carry incrementing stage for the correct generation of the modulo $2^n + 1$ carries.

4. The second stage that implements the conditional addition of the term 2^n+1 can be considered as affecting only the MSB of the result and can thus be computed in parallel with the other bits.

The overall architecture for the PPFCI for the case of a $2^8 + 1$ modulo adder is shown in Fig. 3.21.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

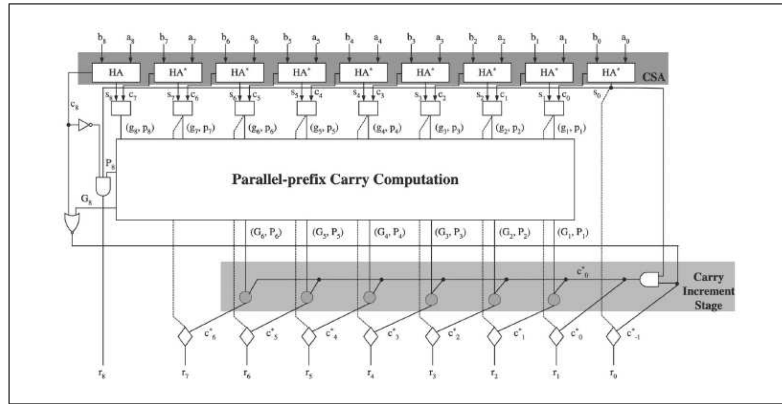

Figure 3.21: PPFCI Modulo $2^n + 1$ addition for the normal representation

3.7.2 TPP

[73] takes the concept of PPFCI further by unwrapping the reentering carry. As a first step, the modulo 2^n+1 carries for the summation unit, c_i^* can be considered equal to G_i^*, where

$$(G_i^*, P_i^*) = \begin{cases} \overline{(g_n + c_n, p_n) \bullet (g_{[n-1,1]}, p_{[n-1,1]})} & \text{if } i = -1 \\ s_0(G_{-1}^*, P_{-1}^*) & \text{if } i = 0 \\ (g_{[i,1]}, p_{[i,1]}) \bullet s_0 \overline{g_{[n-1,i+1]}, p_{[n-1,i+1]}} & \text{if } 1 \leq i \leq (n-2) \end{cases} \tag{3.26}$$

In this equation, $\overline{(g,p)} = (\bar{g}, p)$ and $s(g,p) = (sg,p)$. Based on this equation, modulo 2^n+1 addition carries for $n=8$ can be expressed as:

$$\begin{aligned} c_{-1}^* &= (g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \cdots \bullet (g_1, p_1) \\ c_0^* &= s_0 c_1^* \\ c_1^* &= (g_1, p_1) \bullet (s_0((g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \cdots \bullet (g_2, p_2))) \\ c_2^* &= (g_2, p_2) \bullet (g_1, p_1) \bullet (s_0((g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \cdots \bullet (g_3, p_3))) \\ c_3^* &= (g_3, p_3) \cdots \bullet (g_1, p_1) \bullet (s_0((g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \cdots \bullet (g_4, p_4))) \\ c_4^* &= (g_4, p_4) \cdots \bullet (g_1, p_1) \bullet (s_0((g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \cdots \bullet (g_5, p_5))) \\ c_5^* &= (g_5, p_5) \cdots \bullet (g_1, p_1) \bullet (s_0((g_8 + c_8, p_8) \bullet (\bar{g}_7, p_7) \bullet (g_6, p_6))) \\ c_6^* &= (g_6, p_6) \cdots \bullet (g_1, p_1) \bullet (s_0((g_8 + c_8, \bar{p}_8) \bullet (g_7, p_7))) \end{aligned} \tag{3.27}$$

Carry computation of this form will need more than three prefix level for the carries, c_1^*, c_2^*, c_3^*, c_5^* and c_6^*. [73] tackles this problem by developing two

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

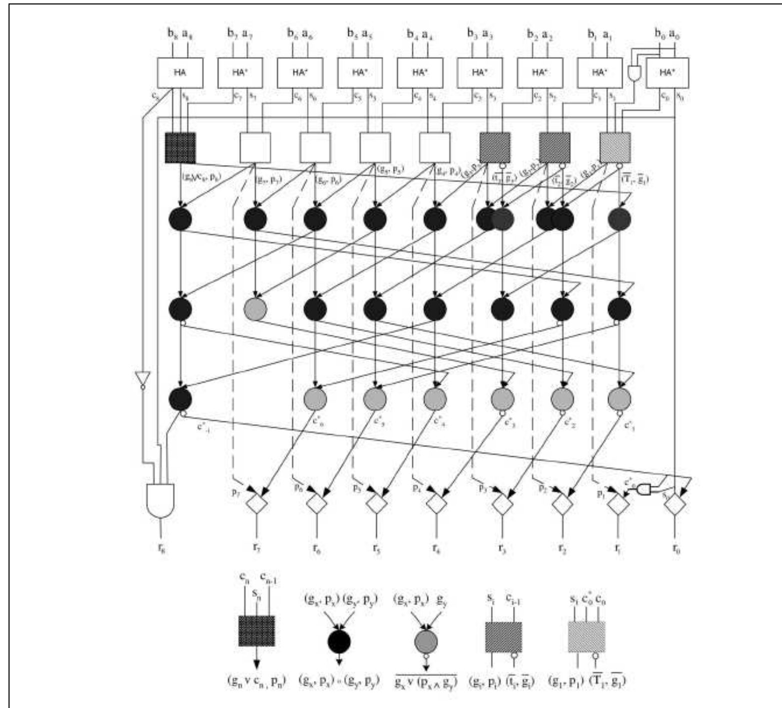

Figure 3.22: TPP Modulo $2^n + 1$ addition for the normal representation

theorems that can be used to transform these carry computation equations to give minimum prefix levels. The theorems are shown below. The proof for these theorems can be found in [73].

Theorem 3.3 *If* $(G^x, P^x) = (g, p) \bullet \overline{(G, P)}$ *and* $(G^y, P^y) = \overline{((\bar{t}, \bar{g}) \bullet (G, P))}$, *then* $G^x = G^y$

Theorem 3.4 *If* $(G^z, P^z) = (g_1, p_1) \bullet s_0(G, \bar{P})$ *and* $(G^w, P^w) = \overline{(\bar{T}_1, \bar{g}_1) \bullet (G, P)}$, *where* $T_1 = s_1 + (a_0 b_0)$, *then* $G^z = G_w$.

Using these two theorems, carry computation for modulo $2^8 + 1$ addition can be reduced to three prefix levels:

$$\begin{aligned}
c_1^* &= \overline{((\bar{T}_1, \bar{g}_1) \bullet (g_8 + c_8, p_8) \bullet (g_7, p_7) \cdots \bullet (g_2, p_2))} \\
c_2^* &= \overline{((\bar{t}_2, \bar{g}_2) \bullet (\bar{T}_1, \bar{g}_1) \bullet (g_8 + c_8, p_8) \bullet (g_7, p_7) \cdots \bullet (g_3, p_3))} \\
c_3^* &= \overline{((\bar{t}_3, \bar{g}_3) \bullet (\bar{t}_2, \bar{g}_2) \bullet (\bar{T}_1, \bar{g}_1) \bullet (g_8 + c_8, p_8) \bullet (g_7, p_7) \cdots \bullet (g_4, p_4))} \\
c_5^* &= (g_5, p_5) \cdots \bullet (g_2, p_2) \bullet \overline{((\bar{T}_1, \bar{g}_1) \bullet (g_8 + c_8, p_8) \bullet (g_7, p_7) \bullet (g_6, p_6))} \\
c_6^* &= (g_6, p_6) \cdots \bullet (g_3, p_3) \bullet \overline{((\bar{t}_2, \bar{g}_2) \bullet (\bar{T}_1, \bar{g}_1) \bullet (g_8 + c_8, p_8) \bullet (g_7, p_7))}
\end{aligned} \tag{3.28}$$

An implementation of such an architecture is shown in Fig. 3.22.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

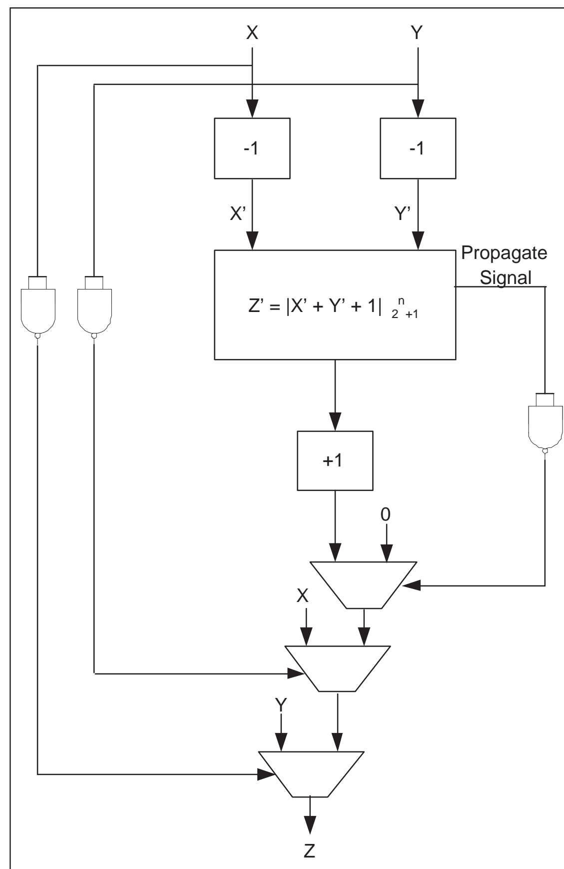

Figure 3.23: Diminished-one modulo $2^n + 1$ addition with correction

3.8 Correction for diminished-one addition

Diminished-one adders as discussed in Section. 3.6 need correction for two different scenarios:

- When any of the inputs is zero, the other input should be considered as the result.
- Inputs that add up to the value of $2^n + 1$ have to be detected and a zero output. This detection uses the fact that such inputs after diminishing give an all 1's representation of the Propagate (P) signal. This detection is in fact similar to the dual representation avoidance circuitry for the $2^n - 1$ modulo adder

A diminished-one modulo $2^n + 1$ adder with these corrections accounted for thus has the structure shown in Fig. 3.23.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.9 Modified diminished-one representation

Correction of the zero values for a diminished-one representation by treating the operands separately increases the latency of a modulo $2^n + 1$ adder. If the multiplexers at the last stage of Fig. 3.23 can be done away with, the performace of the modulo 2^n+1 adder will correspondingly increase. [7] proposes some novel ways for zero handling in diminished-one representation. This solution relies on expressing a diminished-one number with an additional zero indication bit such that for a modulo $2^n + 1$ numbers, $X = x_n x_{n-1} \cdots x_0$, the modified diminished representation is $x_z X$, where X is the diminished-one representation and

$$x_z = \begin{cases} 0 & \text{if } X \neq 0 \\ 1 & \text{if } X = 0 \end{cases} \tag{3.29}$$

Based on this representation, the CEAC can be represented as $F = x_z + \bar{y_z} + c_{out}$. By using this CEAC expression, [7] derives modulo $2^n + 1$ adders based on the CLA and parallel-prefix architectures. These architectures are discussed in the following sections.

3.9.1 CLA Adders with zero handling

The inverted carry at each bit position can be expressed as $\bar{c}_{k+1} = \bar{t}_k + \bar{g}_k \bar{c}_k$. Using this expression, the CEAC can be derived as follows:

$$\begin{aligned} \bar{c}_{out} &= \overline{x_z + y_z + c_n} \\ &= (\overline{x_z + y_z})\bar{c}_n \\ &= (\overline{x_z + y_z})(\bar{t}_{n-1} + \bar{g}_{n-1}\bar{c}_{n-1}) \\ &= \cdots \\ &= (\overline{x_z + y_z + t_{n-1}}) + (\overline{x_z + y_z + g_{n-1}})\bar{t}_{n-2} \\ &+ (\overline{x_z + y_z + g_{n-1}})\bar{g}_{n-2}\bar{t}_{n-3} + \cdots \\ &+ (\overline{x_z + y_z + g_{n-1}})\bar{g}_{n-2} \cdots \bar{g}_0 \end{aligned} \tag{3.30}$$

Using this CEAC recursively, the modulo $2^n + 1$ carries can be easily derived as:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

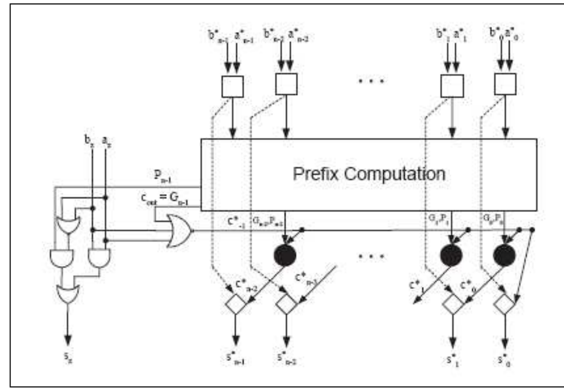

Figure 3.24: Parallel-prefix modulo $2^n + 1$ adder with correct zero handling [7]

$$
\begin{aligned}
c_0^* &= \overline{c_{out}} \\
c_1^* &= g_0 + p_0 c_0^* \\
&= g_0 + p_0(x_z + \overline{y_z} + t_{n-1}) + (x_z + \overline{y_z} + g_{n-1})\overline{t_{n-2}} \\
&+ \cdots + (x_z + \overline{y_z} + g_{n-1})\overline{g_{n-2}} \cdots \bar{g}_0) \\
&= g_0 + p_0(x_z + \overline{y_z} + t_{n-1}) + p_0(x_z + \overline{y_z} + g_{n-1})\overline{t_{n-2}} \\
&+ \cdots p_0(x_z + \overline{y_z} + g_{n-1})\overline{g_{n-2}} \cdots \bar{g}_2 \bar{g}_1
\end{aligned}
\tag{3.31}
$$

An implementation of these carry equations give a modulo $2^n + 1$ adder that can handle zero operands correctly.

3.9.2 Parallel-Prefix adders with zero handling

Two flavours of parallel-prefix adders that have built-in capability for zero handling can be discerned. By feeding the modified CEAC, F to a carry increment stage, the architecture of Fig. 3.24 can be easily visualized.

The unwrapping of the CEAC, F can be done in a way similar to that of modulo $2^n + 1$ adders for the normal and diminished-one representations. Accomplishing this leads to the articulation of unwrapped CEAC modulo $2^n + 1$ adders with correct zero handling. The equations for modulo carries for this adder is the same as in Eq. 3.23 with the only modification to be done being that the term g_7 is expressed as $g_7 = x_z + y_z + (x_7 y_7)$.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.10 Modulo adder Generation using BuRNS

The modulo adders presented in this chapter were generated using BuRNS. The OOP paradigm that is the foundation of the BuRNS eases the architecture of the code generator. A class diagram of BuRNS relevant to modulo adder generation is shown in Fig. 3.25. Each sub-unit of the modulo adder can be considered a class with its own attributes and methods. Common attributes of almost all classes involved in BuRNS are:

- Language : Attribute of type string which chooses language to output the circuit schematic. Possible options are Verilog and VHDL.
- BitWidth : Attribute of type integer which sets the word size of the units

Common methods associated with almost all classes are:

- create() : This method is called on the class to create that particular class. For example, when the method create() is called on a class PGUnit, a PGUnit module is created.
- Instantiate() : This method is called on a created object to instantiate it within the calling module. For example, the class ModuloAdder first creates an object of class PrefixUnit and then calls the method PrefixUnit::Instantiate() to instantiate a prefix unit in the modulo adder.

The following classes are used for modulo adder generation in BuRNS:

- CLANode : This class models a CLA node of selectable bit width. The generated CLA node can be customized to generate several varieties of CLA nodes including :
  - CLA with EAC unwrapped
  - CLA with CEAC unwrapped
  - CLA with input carries
  - CLA with no group signals
- PrefixNode : This class models a prefix node. The prefix node generated can be customized to a reduced prefix node that can be used at the final level of incrementers for EAC/CEAC adders.
- PGNode : This class models a 1-bit Propagate/Generate signal generator.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- SumNode : This class models a 1-bit Sum generator.
- HALNode : This class models a Half Adder Like node as was articulated in [Ahm02].
- func_common : This class provides commonly used methods such as computing the log to base 2, printing the header and module declaration in Verilog, conversion to a binary number etc.
- EmptyNode : This class models a feedthrough.
- CLAUnit : This class models a CLA Unit which can have multiple instances of CLA nodes. The levels of CLA are computed based on the maximum allowable size of a CLA node.
- PrefixUnit : This class models a prefix unit which instantiates multiple prefix nodes to buils a parallel prefix tree. The type of tree built is selectable.
- PGUnit : This class models a PG unit that generates the P/G signals.
- SumUnit : This class models a Summation Unit and instantiates multiple summation nodes.
- SACUnit : This class models a Sum And Carry Unit as was articulated in [4].
- IncUnit : This class models a binary incrementer.
- DimUnit : This class models a binary decrementer.
- ModuloAdder : This class models a modulo adder that instantiates all the common structure of modulo adders.

3.11 Experimental evaluation of modulo adders

Some of the well-known modulo adder architecture presented in this chapter were implemented and analyzed for performance using area, power and delay as metrics. The approach followed for evaluation is standardized across all the modulo adders. The salient features of this approach are:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

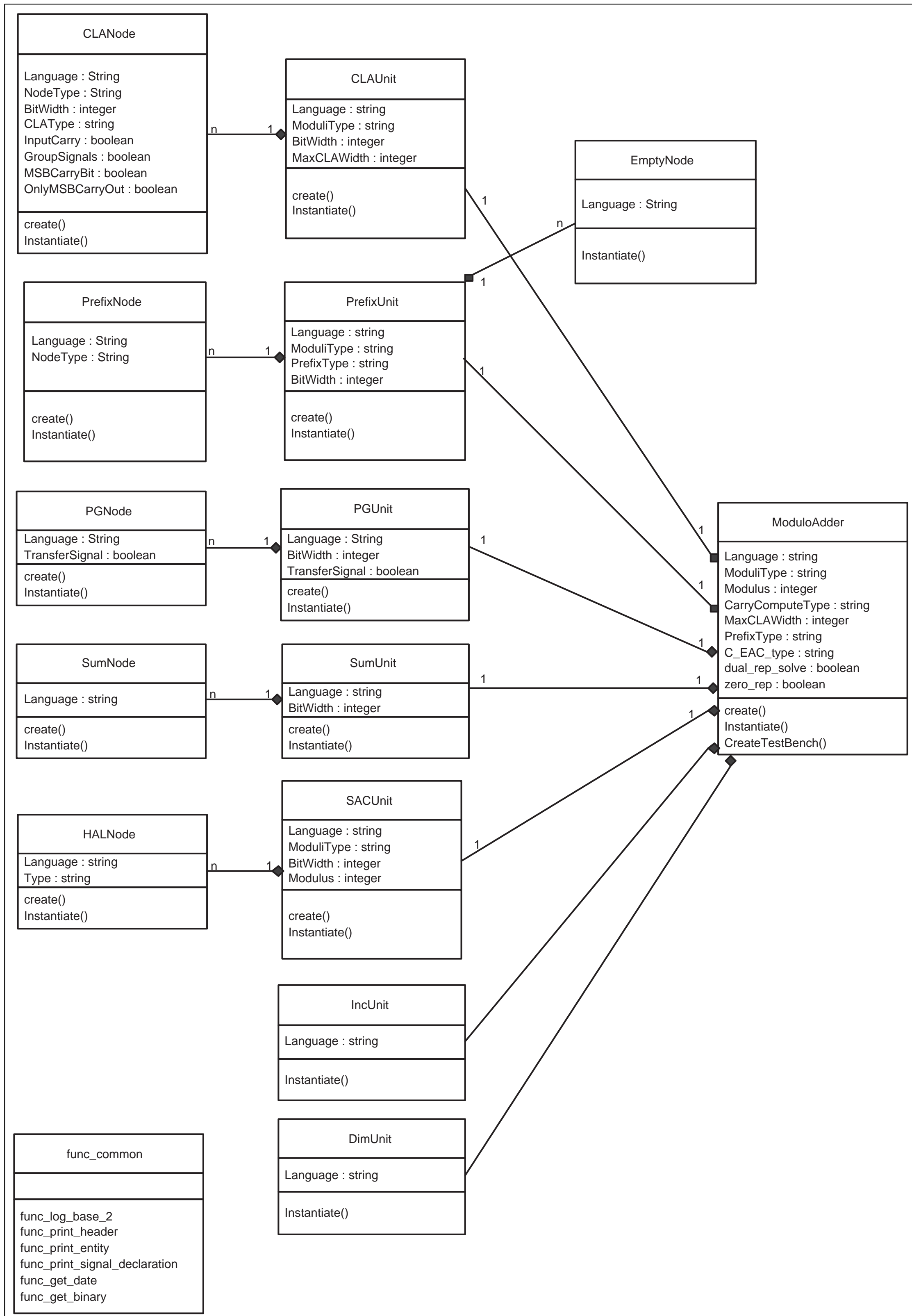

Figure 3.25: Class diagram for modulo adder generation in BuRNS

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

1. The modulo adders have been built as standalone units that accomplish a complete modulo addition of two input operands. Thus, diminished-one adders for modulo $2^n + 1$ representation include the incrementing and decremeting units.

2. The modulo adders have been built to be able to give exact values for all the values within the dynamic range. This means that the modulo adders for modulo $2^n - 1$ solve the dual representation problem and modulo adders for modulo $2^n + 1$ include the correction units as discussed in Section. 3.8.

3. The modulo adders have been built for bit widths of 4, 8, 16 and 32. This is expected to provide broad trends of the behaviour of modulo adders.

4. The block factor of a CLA has been taken as 4 for all the adders based on CLA implementation. Thus, when the bit width of the operands is 8, a 2-level CLA is implemented.

5. Both unwrapped as well as direct implementations of EAC/CEAC using CLA and prefix adders have been considered. For direct implementation of prefix adders, the prefix structure of [30] was used.

Performance evaluation of modulo adders has been done using the metrics of cell area, interconnect area, critical path delay and average dynamic power dissipation.

3.11.1 Cell Area

Cell area is a measure of the gate complexity of a design. The more the cell area, the larger the design. The implication of a large cell area is the increased die size required to fit in the design. Increase in die size means increase in cost per die. Thus cell area is a direct measure of the silicon cost of the modulo adders evaluated.

Table. 3.11.1 shows the cell area results for the modulo adders evaluated. Some trends can be observed from these results :

- Fig. 3.26(a) shows the cell area for modulo $2^n - 1$ adders for the case of n=16. Ling carry based adders are the largest modulo $2^n - 1$ adders. This holds true across all the n evaluated. The prefix with EAC and sparse adder based implementations of the modulo $2^n - 1$ are the most efficient ones. An interesting observation is that while for lower bit widths the prefix with EAC implementations are smaller, as the modulo adders get wider, the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

sparse adder based implementation prove to be smaller. This is line with known properties of the sparse adder being area efficient for large bit widths.

- Fig. 3.26(b) shows the cell area for modulo $2^n - 1$ diminished one adders for the case of n=16. The most area efficient implementations are the CLA and Prefix structures without unwrapping of the CEAC.

- The area overhead of unwrapping EAC/CEAC for a prefix implementation is quite high as can be seen in Fig. 3.26(a) and (b). For unwrapping the EAC/CEAC of a prefix structure, many extra nodes have to be added in the prefix tree.

- The area overhead of unwrapping EAC/CEAC is much lower with a CLA implementation than for a prefix implementation as can be seen in Fig. 3.26(a) and (b). This can be explained by considering that EAC/CEAC unwrapping for CLA only involves adding extra logic to the BGCLA (Fig. 3.8 and Fig. 3.18). Additionally, an extra level of CLA for Cout calculation is removed. Thus, the difference in area for an normal CLA and unwrapped CLA implementation is very trivial.

- The effect on area of unwrapping CEAC for a modulo $2^n + 1$ adder using the normal representation suggests that unwrapping the CEAC actually reduces the area (Fig. 3.26(c)). This incongruity with the other modulo adders can be explained by the fact that the modulo $2^n + 1$ adder actually needs a CLA with bit width of $n + 2$. This means that while the overhead of the modified BGCLA remains around the same, the reduction of area by the removal of the extra CLAForCout unit is much more pronounced. The area difference between the unwrapped CEAC implementation and the normal CLA implementation is more pronounced at higher bit widths.

- Modulo 2^n+1 adder with the diminished-one representation has the largest area. Thus a standalone diminished-one modulo adder is area inefficient and system architects would be well advised to consider modulo $2^n + 1$ adders using the normal representation as presented in [4]. However, there are cases where the incrementing and decrementing for the diminished-one addition may be unnecessary [11]. For such cases, the diminished-one adders will approach the performance of the modulo $2^n + 1$ adders for the normal representation.

- Modulo 2^n adders represent the lowest area that can be aimed at for modulo

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

		n = 4	n = 8	n = 16	n = 32
2^n					
	CLA	901.45	2338.46	5661.53	11685.6
	Prefix Adder	858.21	2025.78	5332.22	11449.4
$2^n - 1$					
	CLA with EAC	1613.30	3353.01	7461.12	14942.1
	CLA with EAC unwrapped	2005.82	3432.84	8133.05	14995.4
	Prefix with EAC	1337.21	3146.77	6865.69	14380.0
	Prefix with EAC unwrapped	1623.28	3851.97	9174.21	19888.2
	Sparse adder based	1809.5	3672.3	7441.1	14030.7
	Ling carry based	2498.1	5096.0	11649.0	25367.1
$2^n + 1$ Dim one					
	CLA with CEAC	3712.26	7181.70	15238.24	32841.5
	CLA with CEAC unwrapped	4138.04	7354.67	16618.70	35036.9
	Prefix with CEAC	3010.39	6832.43	15530.96	33393.7
	Prefix with CEAC unwrapped	3832.01	8834.92	20929.71	48259.4
$2^n + 1$					
	CLA with CEAC	2428.27	4507.27	9264.02	16941.3
	CLA with CEAC unwrapped	2391.68	4444.07	8462.36	16568.7
	Prefix with CEAC	2095.63	4267.77	8449.06	17087.7
	TPP	2245.3	5382.1	11991.6	26977.1
$2^n + 1$ Mod Dim One					
	Prefix Unwrapped	3442.8	8003.3	17802.8	41779.5

Table 3.1: Cell area (in μm^2) for modulo adders

adders. As Fig. 3.26(d) shows, there is still a large scope of improvement for modulo $2^n - 1$ and $2^n + 1$ adders.

3.11.2 Timing

Critical path delay of a design is a measure of how fast the circuit works. For high-speed systems that rely on fast computation of modulo additions, this metric would be the most critical. Critical path delay denotes the worst combinational path delay in the design. This value thus denotes the maximum clock frequency at which any system that encapsulates the design can operate.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

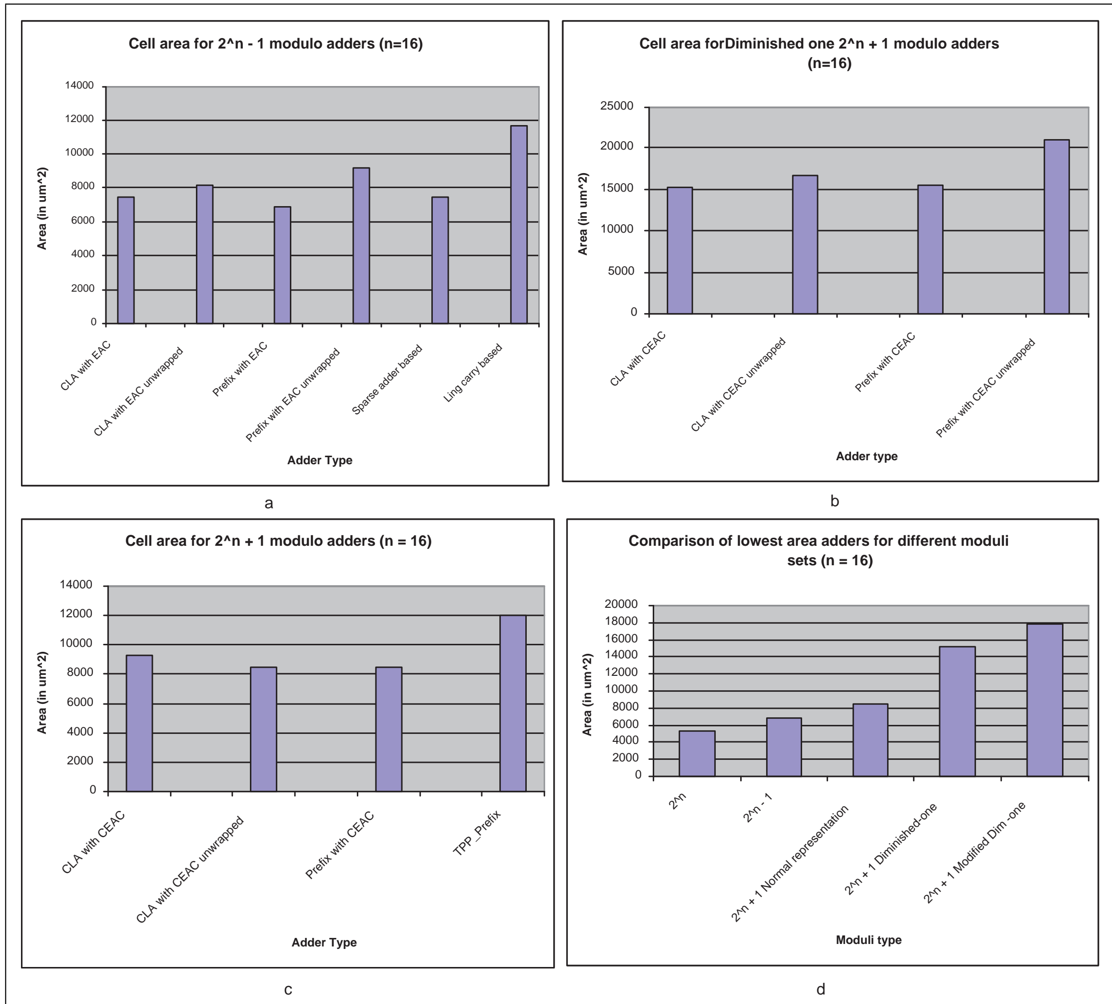

Figure 3.26: Area evaluation of modulo adders for the special moduli set

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table. 3.2 tabulates the synthesis results for the obtained critical path delay. Some trends that can be observed with these results are captured in Fig. 3.27. These trends are:

- Fig. 3.27(a) shows the delay for modulo $2^n - 1$ adders. A clear conclusion that the Ling carry based modulo adders are the best performing can be drawn from this.
- Fig. 3.27(b) shows the delay for diminished one modulo $2^n + 1$ adders. The best performing adder for diminished-one modulo $2^n + 1$ is the prefix with CEAC unwrapped.
- Unwrapping of the EAC/CEAC shows a consistent improvement in critical path delays for modulo $2^n - 1$ adder as well as modulo $2^n + 1$ adder with the diminished one number representation (Fig. 3.27(a) and (b)). This is easily explained by considering that unwrapping of EAC/CEAC removes an extra level for both prefix implementations as well as CLA implementations. For high performance modulo adders for modulo $2^n - 1$ and diminished-one representation of modulo $2^n + 1$, it is thus recommended to go for EAC/CEAC unwrapped implementations.
- For modulo $2^n + 1$ adders with normal representation, no clear trend as to the effectiveness of unwrapping CEAC suggests itself as can be seen in Fig. 3.27(c). The delay differences are trivial enough to prevent drawing any conclusions. Thus, in all probability unwrapping of CEAC does not have much affect on the critical path delay of a modulo $2^n + 1$ adder with the normal representation.
- The best performing adder for the normal representation of modulo $2^n + 1$ adder is the TPP architecture of [73] .
- In almost all types of modulo adders for the special moduli set, prefix implementations show better performance than CLA implementations (Fig. 3.27(d)).
- Modulo 2^n adder benchmarks the best performance that can be aimed at from modulo adders for the special moduli set. As can be seen from Fig. 3.27(e), there is much scope for improving the speed performance of modulo adders for the special moduli set.
- Modulo adders for moduli of type $2^n + 1$ with normal representation outperforms the adder with diminished-one representation. This is not surprising

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

		n = 4	n = 8	n = 16	n = 32
2^n					
	CLA	1.49	2.25	2.76	3.31
	Prefix Adder	1.6	2.06	2.33	2.8
$2^n - 1$					
	CLA with EAC	2.28	2.96	3.43	4.41
	CLA with EAC unwrapped	1.86	2.79	3.09	3.94
	Prefix with EAC	2.23	2.65	3.02	3.39
	Prefix with EAC unwrapped	1.89	2.22	2.61	2.92
	Sparse adder based	2.2	2.63	2.96	2.41
	Ling carry based	1.77	2.14	2.46	2.73
$2^n + 1$ Dim One					
	CLA with CEAC	3.5	5.11	6.33	7.52
	CLA with CEAC unwrapped	3.38	4.84	5.9	7.11
	Prefix with CEAC	3.65	4.94	6	7.04
	Prefix with CEAC unwrapped	3.54	4.54	5.67	6.34
$2^n + 1$					
	CLA with CEAC	2.81	3.43	3.92	4.59
	CLA with CEAC unwrapped	2.94	3.07	4	4.27
	Prefix with CEAC	2.71	3.08	3.44	3.85
	TPP	2.49	2.81	3.11	3.41
$2^n + 1$ Mod Dim One					
	Prefix Unwrapped	3.81	5.03	6.54	7.09

Table 3.2: Critical path delay (in ns) for modulo adders

as the overheads of extra increment and decrement for the diminished-one representation affects the critical path delay. However, there are cases where the incrementing and decrementing for the diminished-one addition may be unnecessary [11]. For such cases, the diminished-one adders will approach the performance of the modulo $2^n + 1$ adders for the normal representation.

3.11.3 Dynamic Power

Power consumption has grown as an important metric for performance evaluation of integrated circuits due to the popularity of portable systems and the need to conserve battery life for such systems. Power consumption of a digital circuit

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

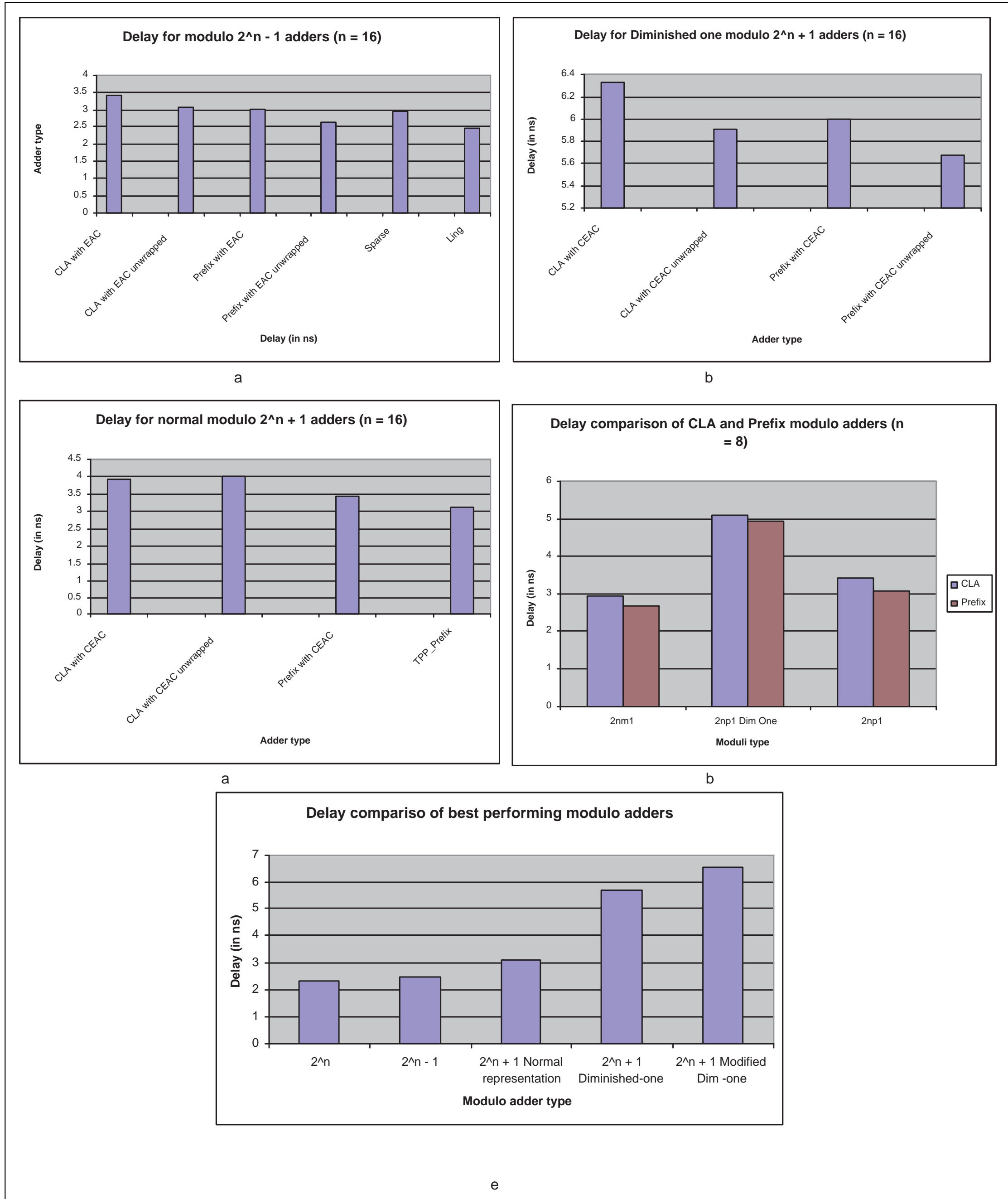

Figure 3.27: Delay evaluation of modulo adders for the special moduli set

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

has two main components : Dynamic power and leakage power. For the 0.18μm CMOS technology being used here for performance evaluation, dynamic power is still the major contributor and thus more critical.

Table. 3.3 shows the dynamic power consumption measured for the modulo adders. A few observations can be made:

- For modulo $2^n - 1$ as well as modulo $2^n + 1$ adder using the diminished-one system, unwrapping of EAC/CEAC causes slightly more power to be dissipated. This can be explained by considering that unwrapped implementations have a larger gate count and thus more gates which switch to consume dynamic power.
- For modulo $2^n + 1$ adders with normal representation, unwrapping does not affect the power performance in a discernible trend.
- As would be expected, the power consumption gap between modulo adders for modulo 2^n and the other types of special moduli is a fair bit and this suggests that there is further scope for optimization.
- Keeping in trend with the area/timing results, the power consumption of a standalone modulo $2^n + 1$ adder with diminished-one representation is higher than one with the normal representation.

3.11.4 Interconnect Area

Interconnect area during synthesis stage is a measure of wiring complexity. The higher the interconnect area, the more complex the routing for the modulo adder is likely to be during Place and Route. Table. 3.4 shows the interconnect area for different modulo adders. Some conclusions can be immediately drawn from the results.

- The unwrapping of carries for both EAC and CEAC increasing wiring complexity of the design. This is because the carries after unwrapping depend also upon carries of more significant bits. Thus in a system where routing resources are limited, unwrapping carries can lead to high congestion.
- The interconnect area of $2^n + 1$ adders are more than the area for $2^n - 1$ modulo adders. This can be expected since the modulo adders for $2^n + 1$ are more complex than the ones for modulo $2^n - 1$.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

		n = 4	n = 8	n = 16	n = 32
2^n					
	CLA	50.79	129.73	328.06	754.5
	Prefix Adder	46.54	106.59	285.13	615,7
$2^n - 1$					
	CLA with EAC	101.99	201.45	441.12	869.0
	CLA with EAC unwrapped	120.90	200.51	481.74	1156.1
	Prefix with EAC	75.29	177.66	362.25	746.24
	Prefix with EAC unwrapped	94.40	219.30	519.75	1195.4
	Sparse adder based	126.4	235.6	484.5	1095.0
	Ling carry based	166.6	320.7	700.4	1561.9
$2^n + 1$ Dim one					
	CLA with CEAC	250.89	389.54	781.39	1828.4
	CLA with CEAC unwrapped	265.43	405.09	900.11	2088.2
	Prefix with CEAC	187.95	375.05	795.73	1909.7
	Prefix with CEAC unwrapped	247.05	487.63	1089.0	2174.1
$2^n + 1$					
	CLA with CEAC	110.97	222.83	522.09	991.9
	CLA with CEAC unwrapped	103.47	233.36	453.74	1052.6
	Prefix with CEAC	96.75	210.71	418.96	879.8
	TPP	121.9	299.8	650.6	1366.3
$2^n + 1$ Mod Dim One					
	Prefix Unwrapped	199.6	456.5	970.3	1843.5

Table 3.3: Dynamic power (in uW) for modulo adders

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

		n = 4	n = 8	n = 16	n = 32
2^n					
	CLA	8400.04	24080.12	55580.27	119140
	Prefix Adder	8120.04	20440.10	53060.26	120120
$2^n - 1$					
	CLA with EAC	15260.07	33180.16	72380.36	145320
	CLA with EAC unwrapped	16520.08	33180.16	74900.37	149380
	Prefix with EAC	13860.06	32480.16	74060.37	159180
	Prefix with EAC unwrapped	16240.08	39340.19	93520.46	213781
	Sparse adder based	16100	35700	73220	137200
	Ling carry based	21140	50120	115360	256901
$2^n + 1$ DimOne					
	CLA with CEAC	31500.15	69020.345	150500.75	324661
	CLA with CEAC unwrapped	34720.17	68460.34	159600.79	335441
	Prefix with CEAC	29120.14	67340.33	153440.76	334881
	Prefix with CEAC unwrapped	36120.17	88760.44	213921.07	499242
$2^n + 1$					
	CLA with CEAC	23240.11	44800.22	83860.42	163520
	CLA with CEAC unwrapped	23240.11	43680.21	82600.41	164640
	Prefix with CEAC	21980.10	42980.21	87080.43	180180
	TPP	19880	47740	111860	259141
$2^n + 1$ Mod Dim One					
	Prefix Unwrapped	31080	77000	180600	434982

Table 3.4: Interconnect area (in μm^2) for modulo adders

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

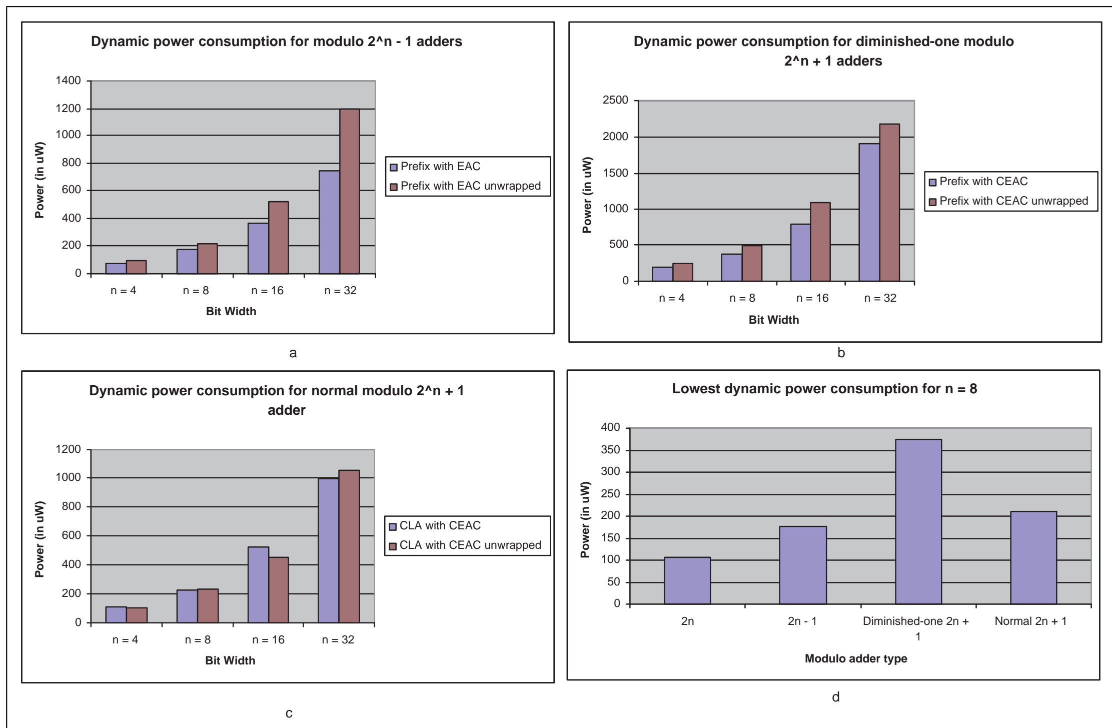

Figure 3.28: Dynamic Power evaluation of modulo adders for the special moduli set

3.12 Summary

This chapter covers modulo adder architectures with special emphasis on moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$. Standard cell implementation of commonly known modulo adder architectures was used to evaluate modulo adders. Usage of BuRNS for generation of modulo adder structures was detailed.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 4

Modulo Multipliers

Modulo multipliers are important components of RNS systems. This chapter deals with modulo multipliers for moduli set belonging to the type $\{2^n-1, 2^n, 2^n+1\}$. Firstly, some well known modulo multipliers for the general moduli set is discussed. Following this, modulo multipliers for moduli of type 2^n-1 are discussed. This is followed by a presentation of modulo multipliers for moduli of type 2^n+1. A novel modulo multiplier for the 2^n+1 moduli set is then articulated. Finally, the usage of BuRNS for generation of modulo multipliers and the experimental evaluation of various modulo multipliers are presented.

4.1 Modulo Multiplication

Modulo multiplication accomplishes the multiplication of two modulo m numbers to give a result modulo m. In its most basic form, modulo multiplication can be articulated as :

$$|XY|_m, X, Y < m = \begin{cases} XY & \text{if } X+Y < m \\ |XY|_m & \text{if } X+Y \geq m \end{cases} \tag{4.1}$$

By its very nature multiplication is a more complex operation than addition [28] and this holds true for modulo multiplication as well. A multitude of modulo multipliers using many varied techniques [16, 18, 76, 77, 78, 12, 13, 79, 80, 81] are known in the literature.

Modulo multipliers can be discerned as operating on either a general modulus [16, 18, 76, 77, 78] or on special moduli belonging to the set $\{2^n-1, 2^n, 2^n+1\}$ [12, 13, 79, 80, 81, 82, 11]. Modulo multipliers for special moduli are more efficient because of the usage of special properties possessed by these moduli [12, 79].

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

A multitude of techniques exist for modulo multiplication.

1. Index calculus reduces a multiplication mod m to an addition mod $m-1$ of some index numbers [16]. The index numbers are usually generated using the primitive root of the modulus[16]. These index numbers can be stored in Look Up Tables (LUTs) and reduce the complexity of modulo multiplication by replacing it with addition operation.

2. Techniques have been proposed [78] that split a prime modulus into smaller factors and thus break up the index addition into smaller modulo adders.

3. Modulo multiplication for non-prime numbers which rely on decomposing the modulus into smaller prime moduli have been proposed in [76]. The idea is to break a large non-prime modulus on which index calculus cannot be used into smaller sub moduli that are relatively prime and accomplish the multiplications on these sub moduli.

4. Quarter square multiplication considers multiplication as an addition between two quarter square numbers:

$$XY = \frac{(X+Y)^2}{4} - \frac{(X-Y)^2}{4} \tag{4.2}$$

   Modulo multipliers using quarter square techniques have been proposed in [77, 83].

5. Modulo multipliers using bit-pair recoding have been proposed in [81, 82, 84, 85]. Using a modified booth recoding, these multipliers have been shown to have better performance than other modulo adders.

6. Modulo multipliers using an array of CSA are also known for moduli of type 2^n+1 [82].

The most common architectures for modulo multiplication however, are variations of the full tree structures seen for binary multipliers [12, 13, 79, 80, 82]. Such multipliers have three main sub blocks as shown in Fig. 4.1. The Partial Product Generator (PPG) generates the partial products from the two input operands. MultiOperand Modulo Adder (MOMA) accomplishes a multioperand modulo addition to reduce the partial products to a sum and carry representation. The final two operand modulo adder adds these two operands to obtain a final sum. Owing to the profusion of tree architectures as well as their common design principles that cut across different moduli, this chapter focuses almost exclusively on modulo multipliers of this type.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

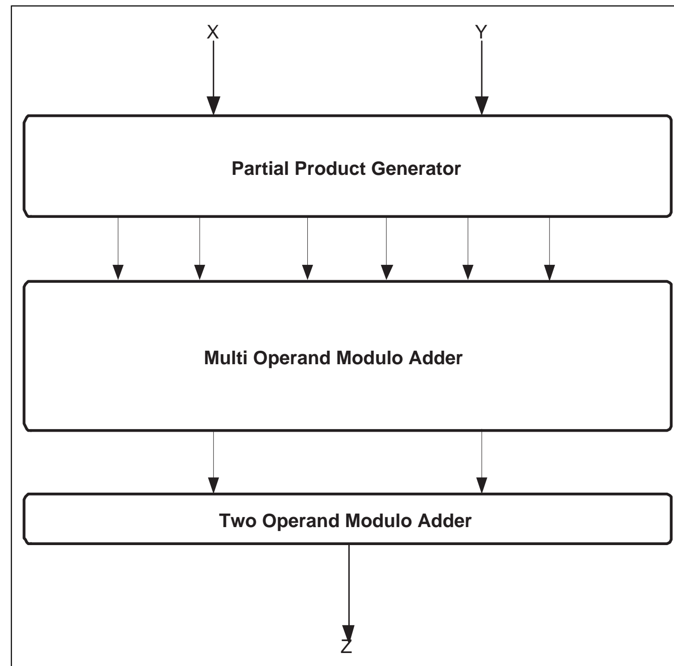

Figure 4.1: General structure of a modulo multiplier

4.2 Modulo multiplier for the general moduli set

Modulo multipliers for a general moduli set do not use any property of special types of moduli for accomplishing multiplication. Many varieties of modulo multipliers for the general moduli set are known in the literature [16, 18, 76, 83, 77, 78, 9, 10]. Designs for modulo multipliers for the pre-VLSI era relied on memory in the form of ROM or LUT's to store precomputed values to ease efficiency of multiplications [76, 83, 77, 78]. The disadvantage of using a ROM/LUT based approach is that as the moduli size grows, the memory requirements grow exponentially and this proves very costly to implement. Most modern proposals for modulo multipliers, however, prefer a logic implementation based approach [9, 10, 8].

Modulo multiplication of two modulo m numbers gives a result that will fall in the range $[0, m-1]$ or $[m, (m-1)^2]$. If the multiplication result falls in the second range a correction factor of $\lfloor \frac{XY}{m} \rfloor$ needs to be subtracted from the result. Mathematically, this operation is represented as:

$$|XY|_m \, , X, Y < m = XY - km, \tag{4.3}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

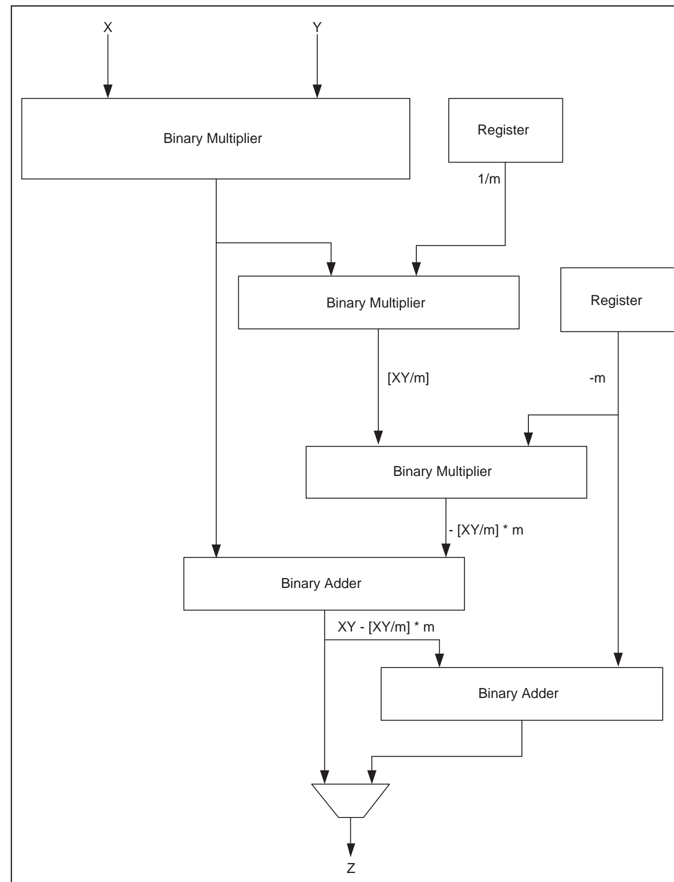

Figure 4.2: Modulo multiplier for general moduli set as proposed in [8]

where the $k = \lfloor \frac{XY}{m} \rfloor$. Thus modulo multiplication is reduced to the computation of the values of k. A direct implementation of this equation was proposed in [8]. In this implementation the values $1/m$ is represented truncated over a particular number of bits so as not to cause any erroneous values. The architecture of such a multiplier is shown in Fig. 4.2. As can be seen, three binary multipliers and two binary adders are required to accomplish modulo multiplication in this scheme and is thus very inefficient.

Residue representations modulo m can be considered as belonging to an integer ring formed of a set of integers $0 \cdots m-1$. A modulo multiplier that relies on recursive decomposition of partial product bits that are larger than or equal to 2^n has been proposed in [9] and extended in [45]. These multipliers rely on the periodicity of the series of powers of 2 taken modulo m [46]. Using an array of Full Adders (FA) to recursively decompose the partial products, a modulo multiplier can be built. An array of FA for modulo 255 multiplication of two 8-bit numbers is shown in Fig. 4.3.

A modulo multiplier that partitions the binary product of the inputs into

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

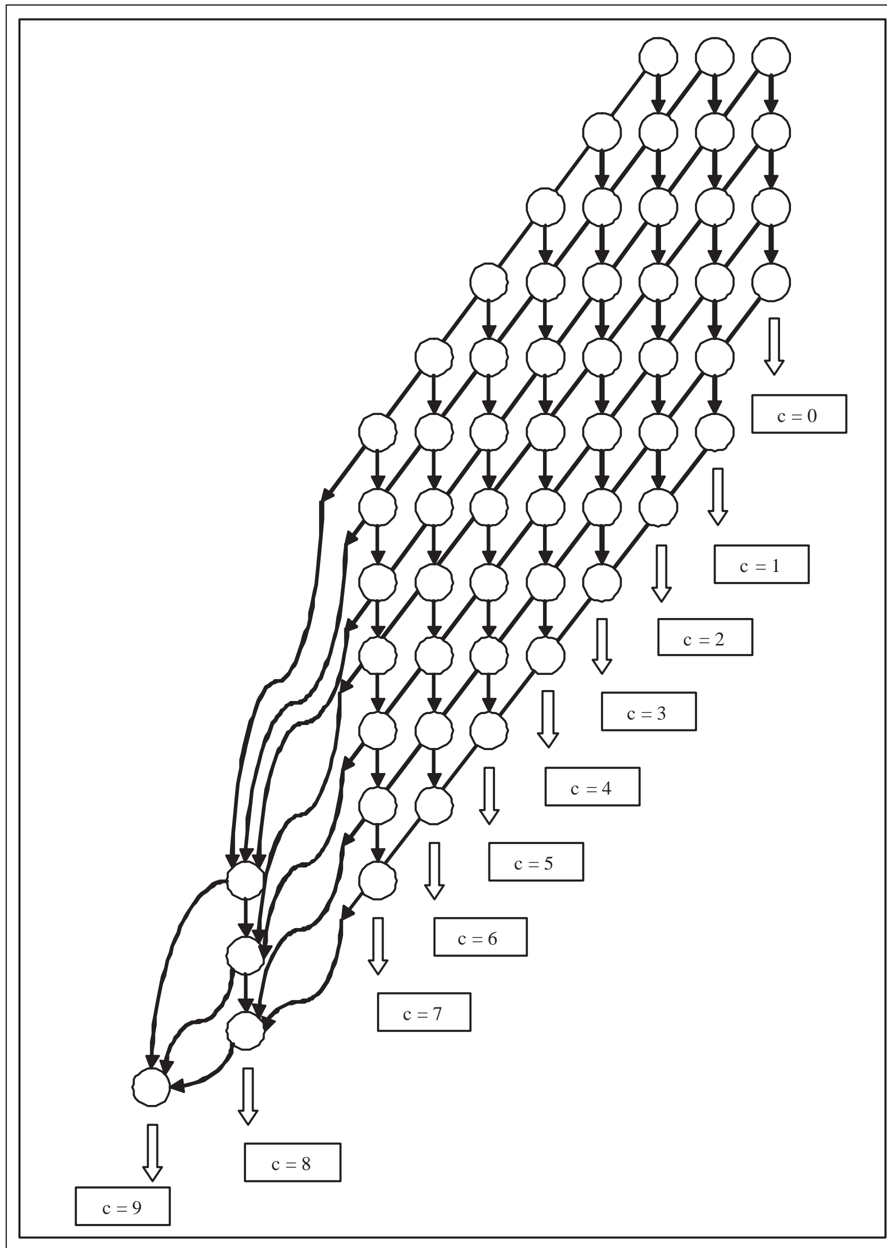

Figure 4.3: Modulo multiplier for a general moduli set as proposed in [9]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

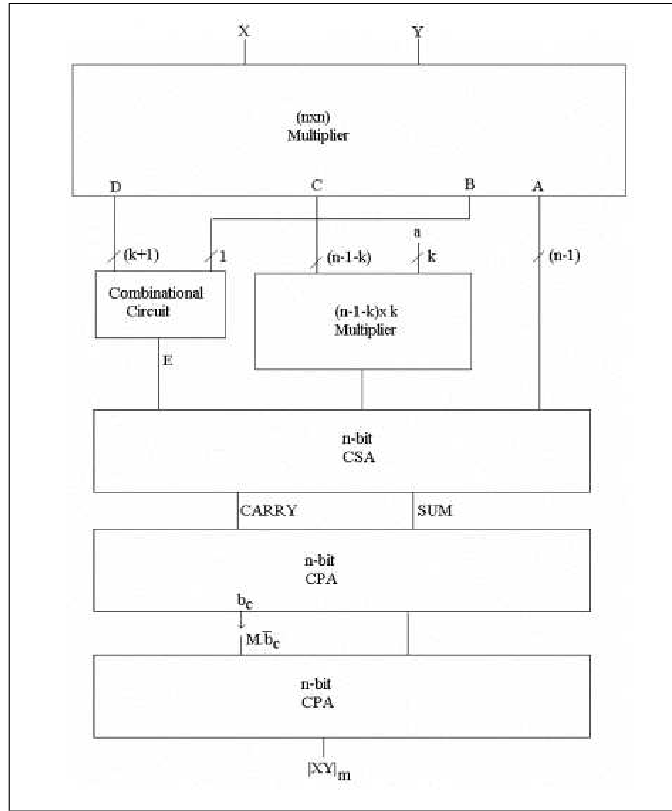

Figure 4.4: Modulo multiplier for a general moduli set as proposed in [10]

four subsets and then relies on an unique reduction method to achieve modulo multiplication was presented in [10]. In this architecture, the binary product $Z = XY$ is partitioned into the four subsets: $A = \sum_{i=0}^{n-2} z_i 2^i$, $B = z_{n-1}$, $C = \sum_{i=n}^{2n-2-k} z_i 2^{i-}$ and $D = \sum_{i=2n-1-k}^{2n-1} z_i 2^{i-(2n-1-k)}$, where k is the width of the value $2^n - m$. These partitioned values are then reduced using the architecture shown in Fig. 4.4.

4.3 Modulo multiplier for moduli set belonging to type $\{2^n - 1, 2^n, 2^n + 1\}$

Modulo multipliers for the general moduli set are expected to work for a wide range of moduli and thus are unable to take advantage of the special properties of some types of moduli. Modulo multipliers for moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ can take advantage of special properties of these moduli and are thus more efficient than modulo multipliers for a general moduli set.

Modulo 2^n multiplier is straightforward to implement in that it involves a binary multiplication with all partial product bits with weight greater than 2^n

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

discarded. This leads to a very simple structure of MOMA with constant word sizes and a final CPA with no carry out. Modulo 2^n multiplier is the most efficient type of modulo multipliers and thus represents the best performance achievable for modulo multipliers.

4.4 Modulo multiplier for moduli set of type $2^n - 1$

ROM/LUT based implementation for modulo $2^n - 1$ multiplication has been proposed in [86]. Such memory based approaches suffer from exponentially increasing memory size with bit width and thus are not very popular in modern implementations [87, 11].

Modulo $2^n - 1$ multiplication can be expressed as :

$$|XY|_{2^n-1} = \left| \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j 2^{i+j} \right|_{2^n-1} \tag{4.4}$$

The term on the RHS can be split up to express the multiplication as:

$$|XY|_{2^n-1} = \left| \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} x_i y_j 2^{i+j} + \sum_{i=1}^{n-1} \sum_{j=n-i}^{n-1} x_i y_j 2^{i+j} \right|_{2^n-1} \tag{4.5}$$

What this expression does is to split up modulo multiplication of two n-bit numbers into two terms: A summation of terms whose weights are less than n and a summation of terms whose weights are greater than or equal to n.

A special case of periodicity of powers of 2 occurs for the modulus $2^n - 1$. For a modulus of type $2^n - 1$ it can be seen that:

$$\left|2^k\right|_{2^n-1}, k \geq n = 2^{k-n} \tag{4.6}$$

Using this special property of modulo $2^n - 1$ numbers, the second term of multiplication modulo $2^n - 1$ in Eq. 4.5 can be reduced as,

$$\left| \sum_{i=1}^{n-1} \sum_{j=n-i}^{n-1} x_i y_j 2^{i+j} \right|_{2^n-1} = \left| \sum_{j=n-i}^{n-1} x_i y_j 2^{i+j-n} \right|_{2^n-1} \tag{4.7}$$

The distinct advantage of this representation is that now there is no partial product term whose weight exceeds 2^n. This means that all the input operands

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

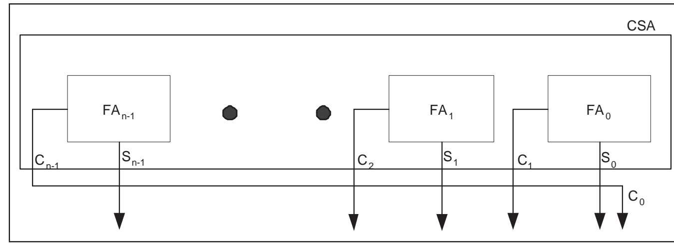

Figure 4.5: CSA with EAC for moduli of type $2^n - 1$

to the MOMA have equal bit width of n-bits. This leads to a very regular implementation of the MOMA. It can be seen that the EAC discussed in Chapter 3 is a special case of Eq. 4.6, where $k = n$.

4.4.1 MOMA for moduli of type $2^n - 1$

A carry save addition of two modulo $2^n - 1$ numbers can be represented as:

$$|X + Y|_{2^n-1} = \left| \sum_{i=0}^{n-1} s_i 2^i + \sum_{i=1}^{n} c_i 2^i \right|_{2^n-1} \tag{4.8}$$

The carry term has a MSB weighted 2^n. Using the property of modulo $2^n - 1$ numbers shown in Eq. 4.6, the carry term can be expressed as :

$$\left| \sum_{i=1}^{n} c_i 2^i \right|_{2^n-1} = \left| \sum_{i=1}^{n-1} c_i 2^i + c_n 2^0 \right|_{2^n-1} \tag{4.9}$$

This representation of the carry save addition for modulo 2^n-1 numbers is similar to the EAC discussed in the context of modulo addition and can be implemented by a modified CSA with EAC unit. A n-bits wide CSA with EAC is shown in Fig. 4.5.

A tree of CSA with EAC can be built to accomplish a multioperand addition for modulo $2^n - 1$ numbers. This tree can be implemented using any of the well known adder tree structures such as Wallace [39] and Dadda trees [40]. An example of a MOMA for modulo $2^n - 1$ numbers for eight operands is shown in Fig. 4.6. Modulo $2^n - 1$ has a very regular structure because all the input operands as well as the intermediate signals are n-bits wide. This is unlike a normal binary multiplier where the carry out from the CSA are left shifted and thus CSA units keep getting wider the lower we go down the tree.

Modulo $2^n - 1$ multipliers that puts together all the concepts of the partial product generation and MOMA presented in this section so far were proposed in

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

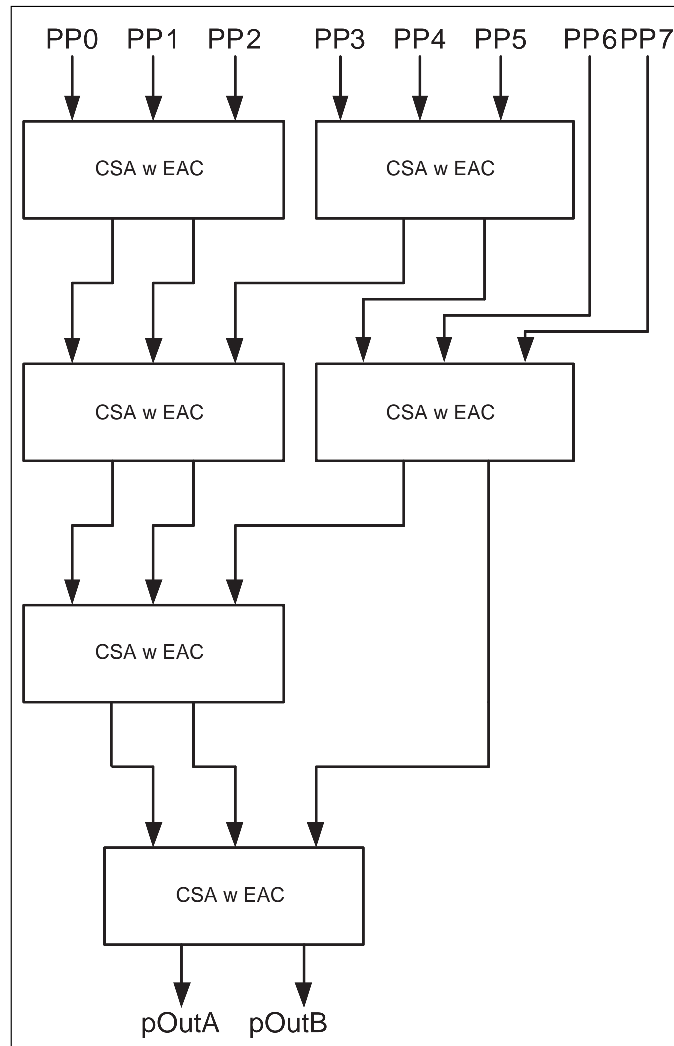

Figure 4.6: MOMA for 8 operands for moduli of type $2^n - 1$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[11, 87]. The articulation of the theory is different in [11] and [87] from what was presented here, but the basic concepts are the same. A modulo $2^n - 1$ multiplier using all these concepts thus looks as shown in Fig. 4.7.

4.5 Modulo multiplier for moduli set of type $2^n + 1$

Modulo multipliers for moduli of type $2^n + 1$ are the slowest of the three special moduli and are thus critical from the standpoint of VLSI performance of RNS systems. In addition to having applications in RNS, standalone modulo multiplier for moduli of type $2^n + 1$ also have applications in cryptography [11].

Multiplication modulo $2^n + 1$ of two n-bit numbers $X = \sum_{i=0}^{n} x_i$ and $Y = \sum_{j=0}^{n} y_i$ is articulated as:

$$|XY|_{2^n+1} = \left| \sum_{i=0}^{n} \sum_{j=0}^{n} x_i y_j 2^{i+j} \right|_{2^n+1} \tag{4.10}$$

It can be noticed from this equation that modulo 2^n+1 multiplication is normally a $(n + 1)$-bit multiplication. The use of diminished-one system [74] can reduce the multiplication to n-bit, albeit with the overhead of additional incrementer and decrementer.

A direct implementation of Eq. 4.10 for modulo $2^n + 1$ multiplication needs to take into account the weights of the partial products having an upper limit of $2n$. Such an implementation is thus very inefficient. To overcome this problem, all implementations [11, 13, 12, 88] rely on special periodic properties of powers of 2 that are greater than or equal to n for modulo 2^n+1 numbers. This property is expressed as:

$$\left|x_k 2^k\right|_{2^n+1} = \left|\bar{x}_k 2^{n-k} + 2^k\right|_{2^n+1} \tag{4.11}$$

The significance of using this property is that the operand term $x_k 2^k$ for all $k > n$ can now be considered as a combination of an operand term $x_k 2^{n-k}$ and a constant term 2^k. The operand term has a weight that is less than or equal to 2^n. The multiplier can now work on $(n + 1)$-bit partial products. The constant terms also need to be summed and this constant summation usually reduces to a final correction term for the multiplier.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

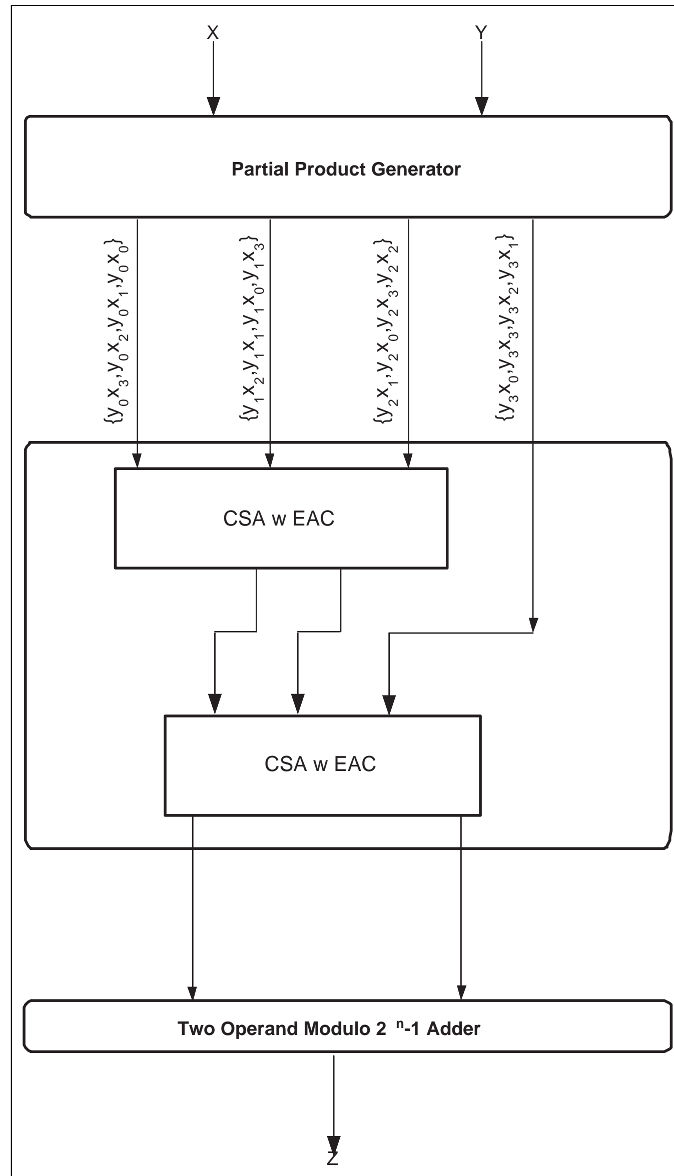

Figure 4.7: Modulo $2^n - 1$ multiplier for n=4

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

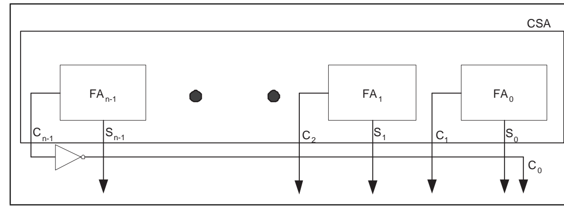

Figure 4.8: CSA with CEAC

4.5.1 MOMA for moduli of type $2^n + 1$

The property of periodicity of powers of 2 modulo $2^n + 1$ as seen in Eq. 4.11 can be used for articulating a variation of CSA structure called the CSA with CEAC (Complemented End Around Carry). Consider the carry save addition of two modulo $2^n + 1$ numbers with n-bit representation:

$$\left| \sum_{i=0}^{n-1} x_i 2^i + \sum_{j=0}^{n-1} y_j 2^j \right|_{2^n+1} = \left| \sum_{i=0}^{n-1} s_i 2^i + \sum_{j=1}^{n} c_j 2^j \right|_{2^n+1} \tag{4.12}$$

The carry bit of weight 2^n can be reduced by considering a special case of Eq. 4.11, where $k = n$. For such a case,$|c_n 2^n|_{2^n+1} = |\bar{c}_n + 2^n|_{2^n+1}$. Thus the carry save addition can now be expressed as:

$$\left| \sum_{i=0}^{n-1} x_i 2^i + \sum_{j=0}^{n-1} y_j 2^j \right|_{2^n+1} = \left| \sum_{i=0}^{n-1} s_i 2^i + \sum_{j=1}^{n-1} c_j 2^j + \bar{c}_n + 2^n \right|_{2^n+1} \tag{4.13}$$

In words, the result of a carry save addition is thus an n-bit sum and an n-bit Complemented End Around carry output. The CSA with CEAC looks as shown in Fig. 4.8. It has to be observed that the there is an additional constant term of 2^n that is accrued for every CSA with CEAC used. The summation of this constant term has to be factored in the correction terms.

Based on the preceding discussions, a complete modulo 2^n+1 multiplier looks as shown in Fig. 4.9. The variations come about in the articulation of the partial product terms and the correction unit needed. In the following subsection, two such well known multipliers proposed in [11, 12] are reviewed.

4.5.2 Review of modulo $2^n + 1$ multipliers

In [11], modulo multiplier for the normal representation of modulo 2^n+1 numbers has been presented. The architecture proposed in [11] is shown in Fig. 4.10. For two $(n + 1)$-bit inputs, the n partial products generated are: $PP_i = x_i \cdot$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

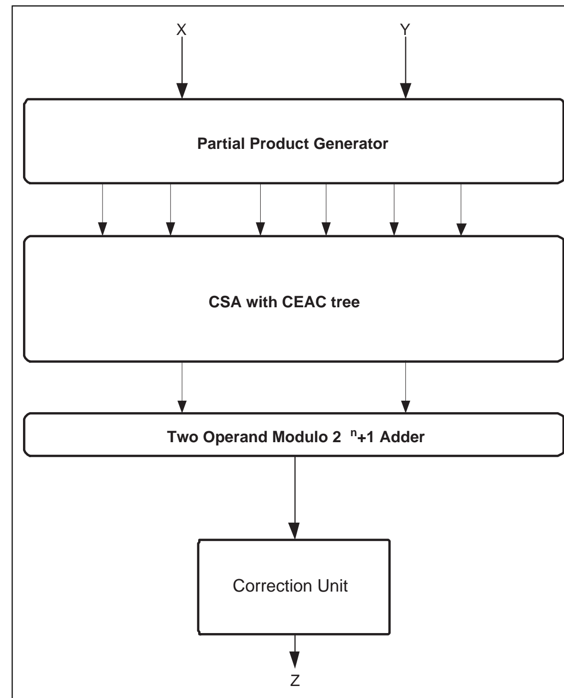

Figure 4.9: General structure of a modulo $2^n + 1$ multiplier

$y_{n-i-1} \ldots y_0 \bar{y}_{n-1} \ldots \bar{y}_{n-i} + \bar{x}_i \cdot 0 \ldots 01 \ldots 1$. Additionally, a constant term of 2 acts to take care of the constant addition terms. The CSA with CEAC tree acts as a MOMA reducing $n+1$ operands to two operands. A final two input modulo 2^n+1 adder adds these two operands. A correction unit for 2^n values at the input implements multiplexer logic to select 2^n-corrected values to be output.

In [12], modulo multiplier for the diminished-one representation of modulo $2^n + 1$ numbers has been presented. The architecture proposed in [12] is shown in Fig. 4.11. The decrementer at the input is required to obtain a diminished-one representation. The $n + 2$ partial products produced for an example of $n = 8$ is shown in Fig. 4.12. Since the inputs are in the diminished-one format, there is no need for a special correction unit for detecting the value 2^n.

In [13] modulo multiplier for the normal representation of modulo $2^n + 1$ numbers has been presented. The architecture proposed in [13] looks as shown in Fig. 4.13. n partial products of n-bits each are reduced to a single modulo $2^n + 1$ number. A constant addition unit then adds a constant to this number to account for the constant correction. After the constant correction, the number can exceed the $2^n + 1$ value. Hence, the need for the final converter.

During the thesis revision, it was noted that two efficient modulo multipliers with slightly different architectures [89, 90] were independently developed and published around the same time.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

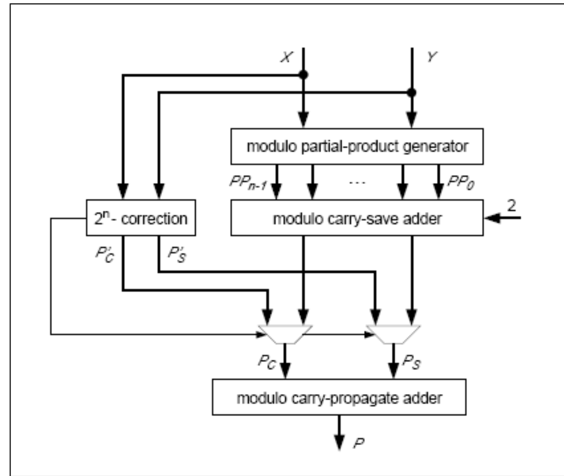

Figure 4.10: Modulo $2^n + 1$ multiplier proposed in [11]

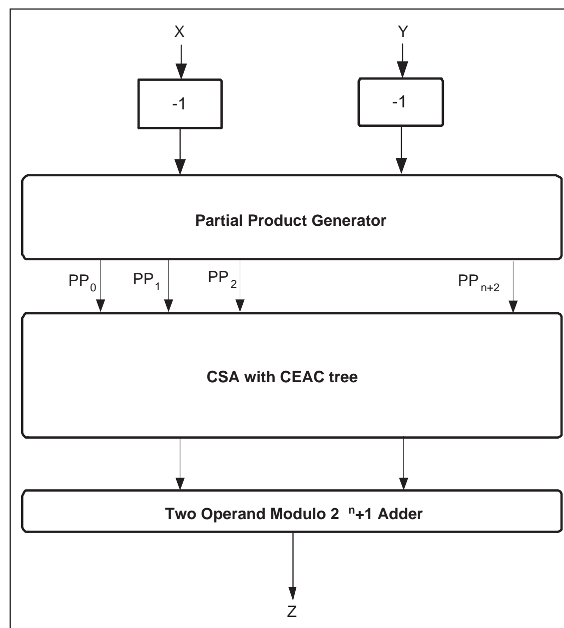

Figure 4.11: Modulo $2^n + 1$ multiplier proposed in [12]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

PP_0 =	a_0b_7	a_0b_6	a_0b_5	a_0b_4	a_0b_3	a_0b_2	a_0b_1	$\underline{a_0b_0}$
PP_1 =	a_1b_6	a_1b_5	a_1b_4	a_1b_3	a_1b_2	a_1b_1	$\underline{a_1b_0}$	$\underline{a_1b_7}$
PP_2 =	a_2b_5	a_2b_4	a_2b_3	a_2b_2	a_2b_1	$\underline{a_2b_0}$	$\underline{a_2b_7}$	$\underline{a_2b_6}$
PP_3 =	a_3b_4	a_3b_3	a_3b_2	a_3b_1	$\underline{a_3b_0}$	$\underline{a_3b_7}$	$\underline{a_3b_6}$	$\underline{a_3b_5}$
PP_4 =	a_4b_3	a_4b_2	a_4b_1	$\underline{a_4b_0}$	$\underline{a_4b_7}$	$\underline{a_4b_6}$	$\underline{a_4b_5}$	$\underline{a_4b_4}$
PP_5 =	a_5b_2	a_5b_1	$\underline{a_5b_0}$	$\underline{a_5b_7}$	$\underline{a_5b_6}$	$\underline{a_5b_5}$	$\underline{a_5b_4}$	$\underline{a_5b_3}$
PP_6 =	a_6b_1	$\underline{a_6b_0}$	$\underline{a_6b_7}$	$\underline{a_6b_6}$	$\underline{a_6b_5}$	$\underline{a_6b_4}$	$\underline{a_6b_3}$	$\underline{a_6b_2}$
PP_7 =	a_7b_0	a_7b_7	a_7b_6	a_7b_5	a_7b_4	a_7b_3	a_7b_2	a_7b_1
PP_8 =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
PP9 =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
PP10 =	0	0	0	0	0	0	0	0

Figure 4.12: Partial product terms for the modulo $2^n + 1$ multiplier proposed in [12]

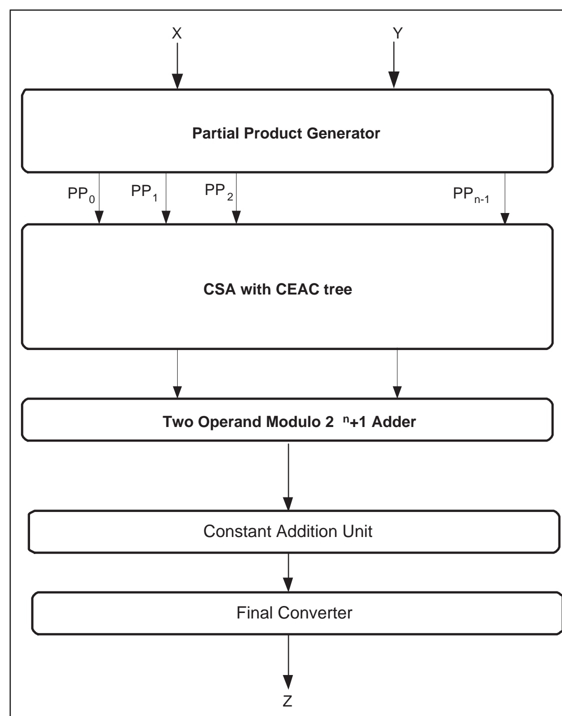

Figure 4.13: Modulo $2^n + 1$ multiplier proposed in [13]

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.6 Proposed modulo $2^n + 1$ multiplier

In this section, our proposal for a novel modulo $2^n + 1$ multiplier for the normal number representation is articulated.

Consider a modulo $2^n + 1$ multiplication.

$$\begin{aligned} |XY|_{2^n+1} &= \left| \sum_{i=0}^{n} \sum_{j=0}^{n} x_i y_j 2^{i+j} \right|_{2^n+1} \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j + x_n \sum_{j=0}^{n-1} y_j \left| 2^{n+j} \right|_{2^n+1} \\ &+ y_n \sum_{i=0}^{n-1} x_i \left| 2^{n+i} \right|_{2^n+1} + \left| x_n y_n 2^2 n \right|_{2^n+1} \\ &= A + B + C + D \end{aligned} \tag{4.14}$$

The term A considers partial products for input bits of weights 2^0 to 2^{n-1}, while terms B, C and D consider the contribution of the partial products of weight 2^n. The term $B + C$ can be combined as:

$$\left| \sum_{k=0}^{n-1} (x_n y_k + x_k y_n) 2^{n+k} \right|_{2^n+1} \tag{4.15}$$

Using the periodic properties of powers of 2 modulo $2^n + 1$ as shown in Eq. 4.11, this term can be reduced as shown:

$$\begin{aligned} &\left| \sum_{k=0}^{n-1} (x_n y_k + x_k y_n) 2^{n+k} \right|_{2^n+1} \\ &= \left| \sum_{k=0}^{n-1} [x_n \oplus y_n][x_k + y_k] 2^{n+k} \right|_{2^n+1} \\ &= \left| \sum_{k=0}^{n-1} s q_k 2^{n+k} \right|_{2^n+1} \\ &= \left| s \sum_{k=0}^{n-1} \bar{q}_k 2^k + 2^{n+k} \right|_{2^n+1} \\ &= \left| \sum_{k=0}^{n-1} s \bar{q}_k 2^k + s \sum_{m=n}^{2n-1} 2^m \right|_{2^n+1} \\ &= \left| \sum_{k=0}^{n-1} s \bar{q}_k 2^k + 2s \right|_{2^n+1} \end{aligned} \tag{4.16}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

The term D of Eq. 4.14 can also be reduced using the property of periodicity:

$$\left|a_n b_n 2^{2n}\right|_{2^n+1} = a_n b_n 2^0 = a_n b_n \tag{4.17}$$

The term A of Eq. 4.14 can be further divided into two components, one with partial product weights lesser than n and the other with partial product weights greater than or equal to 2^n.

$$\left|\sum_{i=0}^{n-1}\sum_{j=0}^{n-1-i} x_i y_j 2^{i+j} + \sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1} x_i y_j 2^{i+j}\right|_{2^n+1} \tag{4.18}$$

The properties of periodicity of powers of 2 modulo 2^n+1 can be used to further reduce the component with partial product weights greater than or equal to 2^n.

$$\begin{aligned}
&\left|\sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1} x_i y_j 2^{i+j}\right|_{2^n+1} \\
&= \left|\sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1} (\overline{x_i b_j} 2^{i+j-n} + 2^{i+j})\right|_{2^n+1} \\
&= \left|\sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1} (\overline{x_i b_j} 2^{i+j-n}) + 2^n(2^n-1-n)\right|_{2^n+1}
\end{aligned} \tag{4.19}$$

Using all the reduced terms, modulo 2^n+1 multiplication can be articulated as:

$$\begin{aligned}
|XY|_{2^n+1} = |&\sum_{i=0}^{n-1}\sum_{j=0}^{n-1-i} x_i y_j 2^{i+j} + \sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1} (\overline{x_i b_j} 2^{i+j-n}) \\
&+ \sum_{k=0}^{n-1} s\bar{q}_k 2^k + a_n b_n + 2s + 2^n(2^n-1-n)|_{2^n+1}
\end{aligned} \tag{4.20}$$

Eq. 4.20 sets out the partial products to be summed to obtain the modulo 2^n+1 product. However, the use of a CSA with CEAC tree as the MOMA requires additional correction factors. This is because every CSA with CEAC adds a constant term 2^n to the result. For a Wallace tree consisting of n input operands, the number of CSA units required is $n-2$ [17]. The number of partial products from Eq. 4.20 is $n+4$. However, if we can reduce the number of partial products to $n+3$, then the constant term to be added through the MOMA is expressed as $(n+1)2^n$. Summing this constant term with the final constant term of Eq. 4.20:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$\begin{aligned}&|2^n(2^n-1-n)+(n+1)2^n|_{2^n+1}\\&=\left|2^{2n}\right|_{2^n+1}\\&=1\end{aligned} \tag{4.21}$$

Thus modulo multiplication can now be expressed as:

$$\begin{aligned}|XY|_{2^n+1}=|&\sum_{i=0}^{n-1}\sum_{j=0}^{n-1-i}x_iy_j2^{i+j}+\sum_{i=1}^{n-1}\sum_{j=n-i}^{n-1}(\overline{x_ib_j}2^{i+j-n})\\&+\sum_{k=0}^{n-1}s\bar{q}_k2^k+a_nb_n+2s+1|_{2^n+1}\end{aligned} \tag{4.22}$$

A modulo 2^n+1 CPA with CEAC implements an incremented modulo addition $|X+Y+1|_{2^n+1}$. Thus the usage of such a CPA takes into account the constant term '1' in the partial products of Eq. 4.23. This leads us to articulation of the modulo 2^n+1 multiplier as:

$$|XY|_{2^n+1}=\left|\sum_{\oplus}PP+s\bar{q}_k+a_nb_n+2s\right|_{2^n+1} \tag{4.23}$$

As can be seen, the number of partial products is now $n+3$ and this validates our reduction of the constant term. The partial products obtained for this multiplier is similar to the one proposed in [13]. However, where the proposed multiplier differs is in the articulation of some extra terms for the partial products so as to avoid additional correction units that reduce the overall latency of the multiplier.

The proposed modulo adder has a structure as shown in Fig. 4.14. The partial products for the proposed modulo adder for $n=4$ is shown in Eq. 4.24.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

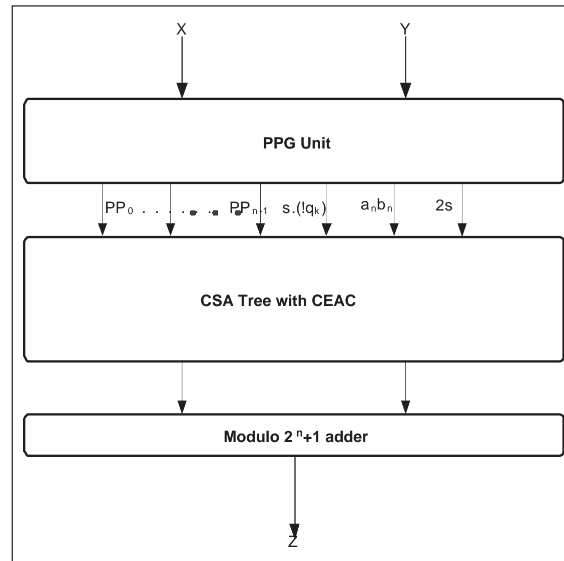

Figure 4.14: Proposed novel modulo $2^n + 1$ multiplier

$$
\begin{aligned}
PP_0 &= x_3y_02^3 + x_2y_02^2 + x_1y_02^1 + x_0y_02^0 \\
PP_1 &= x_2y_12^3 + x_1y_12^2 + x_0y_12^1 + \overline{x_3y_1}2^0 \\
PP_2 &= x_1y_22^3 + x_0y_22^2 + \overline{x_3y_2}2^1 + \overline{x_2y_2} \\
PP_3 &= x_0y_32^3 + \overline{x_3y_3}2^2 + \overline{x_2y_3}2^1 + \overline{x_1y_3}2^0 \\
sq_0 &= (x_4 \oplus y_4)(\overline{x_3 + y_3})2^3 + (x_4 \oplus y_4)(\overline{x_2 + y_2})2^2 \\
&\quad + (x_4 \oplus y_4)(\overline{x_1 + y_1})2^1 + (x_4 \oplus y_4)(\overline{x_0 + y_0})2^0 \\
2s &= (x_4 \oplus y_4)2^1
\end{aligned}
\tag{4.24}
$$

It should be noted that for the proposed architecture, there is a requirement of adding three additional terms compared to a multiplier having n partial products [11]. This somewhat widens the design of the MOMA unit without necessarily affecting the depth and latency. Extra delay will be incurred only in the case where the number of operands falls in a range such that the addition of 3 terms more causes an extra level of adders to be inserted in the tree. An example would be the 8-bit modulo multiplier. For the modulo 2^n and $2^n - 1$ cases, an 8 operand MOMA would be needed and the number of levels would be 4. The need to add three terms for modulo $2^n + 1$ case leads to a 5-level MOMA. Thus an additional stage of CSA delay is incurred. However, this overhead for these special cases is expected to be assuaged by the fact that no extra correction units are required.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.7 Modulo multiplier generation using BuRNS

BuRNS was used to generate parameterizable Verilog RTL code for modulo multipliers to be evaluated. Adoption of the Object Oriented approach eases the implementation of the code generators. A class diagram of BuRNS detailing the classes used for modulo multiplier generation is shown in Fig. 4.15. The components of modulo multiplier are each encapsulated as a class with its own data and behavior. The classes particularly relevant to modulo multiplier generation are:

1. CSANode: This class implements a CSA unit for the specified number of bits. The data *BitWidth* sets the number of input bits to this CSA node.

2. MOMA: This class implements the Multi Operand Modulo Adder generator. This class instantiates objects belonging to the class CSANode to build a Wallace tree of CSA. This class can be configured for varying number of input operands by setting the data associated with this class.

3. PPGUnit: This class implements a Partial Product Generation unit. The bit width of the input operands can be configured by setting the data structures associated with this class.

4. ModuloAdder: This class instantiates the 2-operand modulo adder. This class was discussed in more detail in Chapter 3.

5. IncUnit: This class creates an incrementing unit.

6. DimUnit: This class creates a diminishing unit.

7. ModuloMultiplier : This class is the parent class which calls objects of other classes to create modulo multipliers. The algorithms for different kinds of modulo multiplier generation is encapsulated in this class.

4.8 Experimental Evaluation

Modulo multipliers presented in this chapter were evaluated. The evaluation setup was standardized for all modulo multipliers. The salient features of this setup are:

- Modulo multipliers have been evaluated as complete standalone systems that work across the complete range of inputs. Thus, when evaluating

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

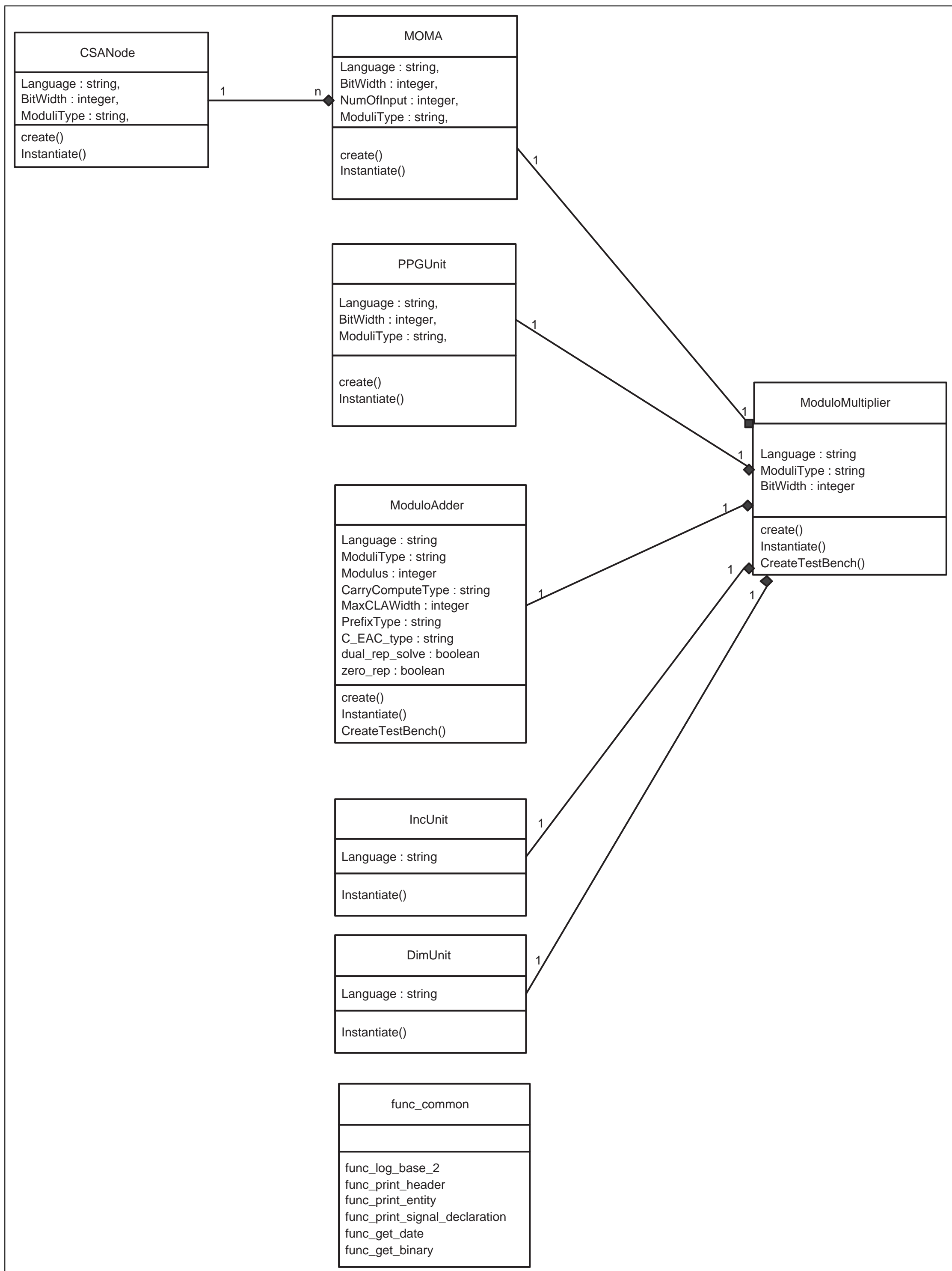

Figure 4.15: Class diagram for BuRNS for generation of modulo multipliers

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

diminished-one modulo $2^n + 1$ multipliers, decrementing units have been considered a part of the modulo multiplier. Similarly, other cases arise when some additional correction units are necessary and these have been included as part of the modulo multiplier units.

- For all the two operand modulo adders, a Sklansky type [30] prefix tree with no carry unwrapping has been used. This standardizes the two-operand modulo adders used for all modulo multipliers.
- The MOMA used in modulo multipliers for evaluation is of the Wallace tree type.
- Bit widths of 4, 8, 16 and 32 have been considered.
- BuRNS has been used to generate the Verilog RTL description of modulo multipliers.

Performance evaluation has been considered from the perspective of VLSI metrics of area, critical path delay and dynamic power. The following subsections set out the evaluation results obtained and discussion of what these results suggest. Modulo multipliers evaluated are:

- Modulo 2^n multiplier as a binary multiplier with bits of weight greater than 2^n discarded.
- Modulo $2^n - 1$ multiplier as discussed in Section. 4.4.
- Modulo 2^n+1 multipliers proposed by [11], [12] and the proposed multiplier in Section. 4.6.

4.8.1 Cell Area

The standard cell area of a design is a measure of the silicon cost of the design. The more complex a design is, the more the silicon cost. Table. 4.8.1 tabulates the cell area results obtained for modulo multipliers. Fig. 4.16 presents the tabulated results in a graphical form for easy analysis. Some important conclusions can be drawn.

- The best performing modulo $2^n + 1$ multiplier in terms of cell area is the one proposed in [11]. The reasons for this are manifold. Firstly, the modulo 2^n+1 multiplier proposed in [12] is for a diminished-one system and for the purpose of its evaluation as a standalone complete modulo 2^n+1 multiplier,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Type of Modulo multiplier	n = 4	n = 8	n = 16	n =32
2^n	3160.08	12464.02	55135.08	210404.7
$2^n - 1$	4011.64	17866.10	71950.03	242188.5
$2^n + 1$[12]	9699.78	29797.89	90760.82	293847.5
$2^n + 1$[11]	6390.01	23966.71	82980.37	259272.7
$2^n + 1$Proposed	9387.10	28896.44	88784.95	266082.0

Table 4.1: Cell area (in μm^2)for modulo multipliers

binary decrementers have been used at the input. This is expected to contribute a significant portion to the cell area of the modulo multiplier.

- The proposed modulo $2^n + 1$ multiplier requires additional area compared to the modulo multiplier in [11]. This occurs because of the need for extra partial products in the proposed design compared to that proposed in [11].

- An interesting aspect of the proposed modulo $2^n + 1$ multiplier can be observed. For increasing operand sizes, the difference between the proposed design and the design [11] narrows. In percentage terms, the difference is 31.9% for $n = 4$, 17.0% for $n = 8$ and 6.5% for $n = 16$. This narrowing of the area difference is explainable by considering that as the operand sizes increase, the impact of just two additional terms in the partial products becomes less pronounced.

- There is a fairly large gap in area usage of a modulo $2^n + 1$ multiplier as opposed to modulo $2^n - 1$ and 2^n multipliers. This suggests that bridging this performance gap should be a high priority for designing efficient RNS architectures.

4.8.2 Timing

Critical path delay of a design sets out the maximum frequency at which the design can be operated and is thus a very important metric for measurement of VLSI design performance. Table. 4.8.2 tabulates the critical path delay for the various modulo multipliers evaluated. Fig. 4.17 presents the tabulated result in an easily analyzable graphical form. Some important conclusions can be drawn from these results.

- For modulo $2^n + 1$ multipliers evaluated, the maximum delay is for the [12] multiplier. This is due to the presence on the critical path of the decrementers at the input operand.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

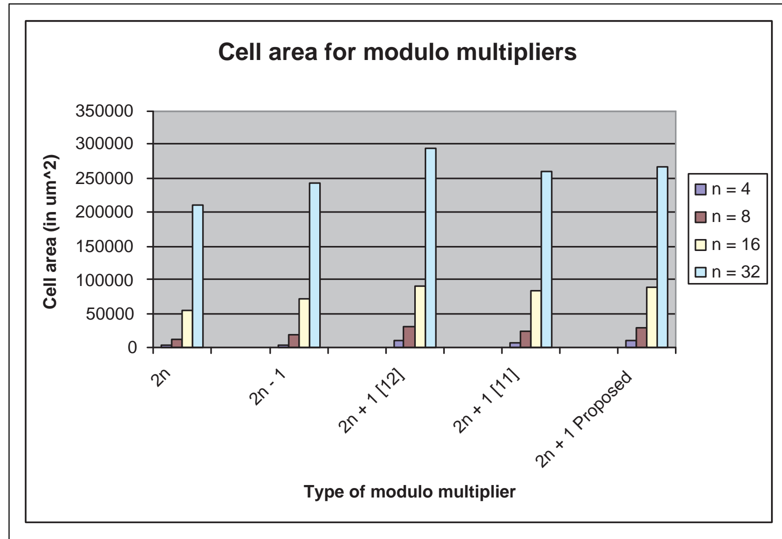

Figure 4.16: Cell area in μm^2 for modulo multipliers

- The proposed multiplier outperforms the [11] multiplier for an operand size of 16, while being outperformed for smaller bit width. This brings to the fore an interesting phenomenon associated with our proposed multiplier. Consider the case for $n = 4$. For the modulo $2^n + 1$ multiplier [11], the number of partial products to be reduced is $n + 1 = 5$. Using a Wallace tree, this translates to 3 levels for reduction to 2 operands. For the proposed modulo $2^n + 1$ multiplier, the number of partial products is $n + 3 = 7$. This translates to 4 levels for the Wallace tree implementation of MOMA. This extra delay level is reflected in the critical path delay results obtained. The same phenomenon holds for $n = 8$, where modulo $2^n + 1$ multiplier of [11] requires 4 levels while modulo $2^n + 1$ multiplier proposed in this chapter requires 5 levels. However, for $n = 16$, addition of an extra two partial product terms do not lead to extra levels on the CSA tree. In this scenario, the proposed multiplier outperforms the other modulo $2^n + 1$ multiplier.

- A performance gap exists between modulo multipliers for moduli of type $2^n + 1$ and for moduli of types $2^n - 1$ and 2^n. Thus there exists further scope for improvement of modulo multiplier design.

4.8.3 Dynamic Power

Dynamic power is the major contributor to power consumption at the 0.18μm technology being considered here. Table. 4.8.3 tabulates the dynamic power con-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Type of Modulo multiplier	n = 4	n = 8	n = 16	n = 32
2^n	3.25	4.77	6.36	7.84
$2^n - 1$	3.82	5.33	6.98	8.97
$2^n + 1$ [12]	6.26	8.09	10.3	12.54
$2^n + 1$ [11]	4.74	5.88	7.53	9.6
$2^n + 1$ Proposed	5.05	6.13	7.46	9.73

Table 4.2: Critical path delay (in ns) for modulo multipliers

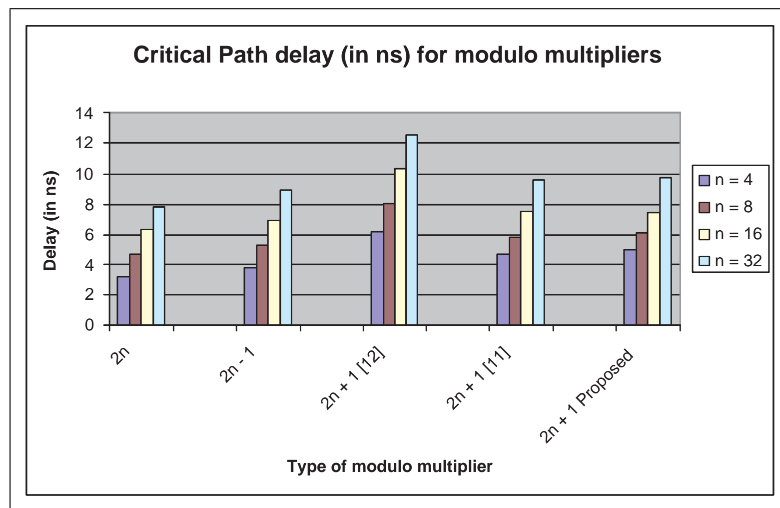

Figure 4.17: Delay in ns for modulo multipliers

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Type of Modulo multiplier	n = 4	n = 8	n = 16	n = 32
2^n	101.79	374.409	1896.3	8533.5
$2^n - 1$	190.959	938.349	3796.6	19362.6
$2^n - 1$ [12]	602.10	1763.8	5135.9	25936.2
$2^n - 1$ [11]	310.04	1281.9	4648.5	22312.8
$2^n - 1$ Proposed	449.23	1517.3	4582.4	21766.4

Table 4.3: Dynamic power consumption (in ns) for modulo multipliers

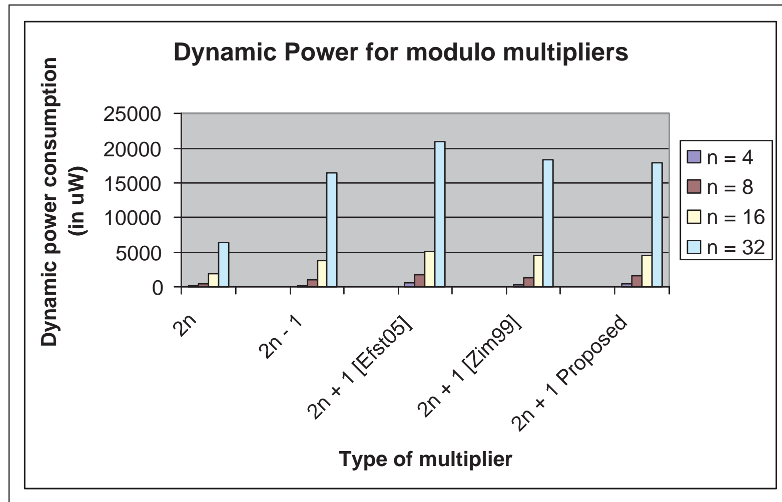

Figure 4.18: Dynamic power consumption (in μW) for modulo multipliers

sumption of modulo multipliers under evaluation. Fig. 4.18 shows the results in a graphical format. Some trends can be observed based on these results.

- Dynamic power consumption closely tallies with the area/delay performance for modulo multipliers. Thus we see that the [12] modulo $2^n + 1$ multiplier has the highest dynamic power consumption among all modulo multipliers. The reason for this is the presence of the decrementers at the outset to convert the numbers to a diminished-one representation.

- The modulo $2^n + 1$ multiplier proposed in this thesis outperforms the other modulo multipliers evaluated for large operands.

- The performance gap seen between modulo multipliers for moduli of type $2^n + 1$ and modulo multipliers for moduli of types $2^n - 1$ and 2^n suggests that for dynamic power consumption as a metric there is considerable scope for improvement of modulo $2^n + 1$ multipliers.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.9 Summary

This chapter covers modulo multipliers with special emphasis on moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$. Well known designs for modulo multipliers have been detailed and evaluated for VLSI performance metrics. Additionally, a novel modulo $2^n + 1$ multiplier that outperforms similar modulo multipliers for large values of n has been articulated. Important aspects of code generation using BuRNS have also been covered.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 5

Shared moduli architectures

This chapter presents shared moduli architectures for modulo arithmetic units based on moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$ and the motivation behind the development of these architectures.

5.1 Definition of shared moduli architectures

The architectures that have been discussed so far in this thesis work on individual modulus, usually belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$. However, many applications use not a single modulus but a set of moduli [23, 55, 24, 25, 59, 21]. These applications thus work on a multimodulus basis. In most cases, all modulo arithmetic units for the various moduli are expected to work in parallel to achieve high performance. However, there are special classes of applications, where we can consider modulo arithmetic operations on a serial basis. These applications will be detailed in the next section. For such applications we propose the use of shared moduli architectures. As the name suggests, these architectures are built in a way that shares multiple modulus for arithmetic operations. These shared moduli structures will work for any moduli selectable from a predefined set. In particular, a predefined set from the most common special moduli sets $\{2^n - 1, 2^n, 2^n + 1\}$ is considered. The shared moduli architectures are thus single blocks that can work as arithmetic unit on a modulus selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$.

5.2 Motivation for shared moduli architectures

The need for shared moduli architectures can be articulated based on the following aspects:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

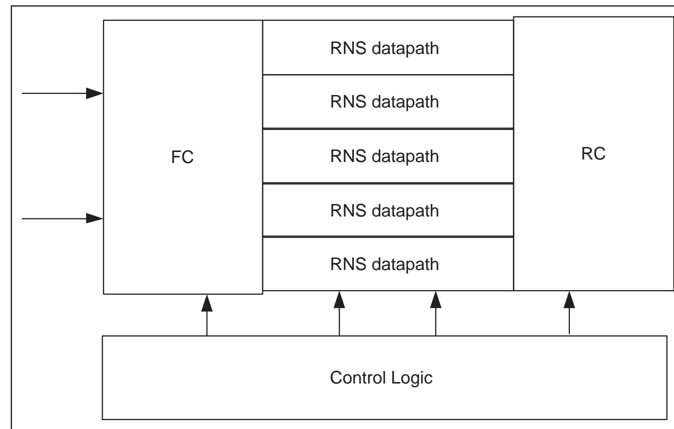

Figure 5.1: Programmable RNS datapaths in a reconfigurable system as proposed in [14]

1. Programmable RNS datapaths

   Datapaths in which channels can be activated individually have been used within a framework of reconfigurable computers [14]. The requirements of the datapath are that they be programmable in terms of dynamic range, function as well as sequence. A typical reconfigurable processor with such a configurable RNS datapath is shown in Fig. 5.1. With the ubiquity of special moduli sets of type $\{2^n - 1, 2^n, 2^n + 1\}$ in modern RNS, it is worthwhile considering RNS datapaths in which most of the moduli belong to the type $\{2^n - 1, 2^n, 2^n + 1\}$. Possessing the inherent ability to switch between moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$, shared-moduli architectures fit snugly into the reconfigurable framework.

2. Serial By Modulus DSP

   RNS DSP kernels that sacrifice speed for efficient utilization of resources are known [91]. These kernels use a serial by modulus RNS architecture to implement a Variable Word Length (VWL) processor. A serial by modulus RNS involves processing the residue digits serially to achieve a compromise between the slow speed of bit-serial binary arithmetic and the high cost of bit parallel binary arithmetic. An architecture that allows the moduli to be selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$ can form an integral component of a serial by modulus architecture. For every residue bit that is processed serially, a modulus belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ can be chosen.

3. Multifunction architectures

   RNS architectures that accomplish multiple modulo arithmetic functions

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

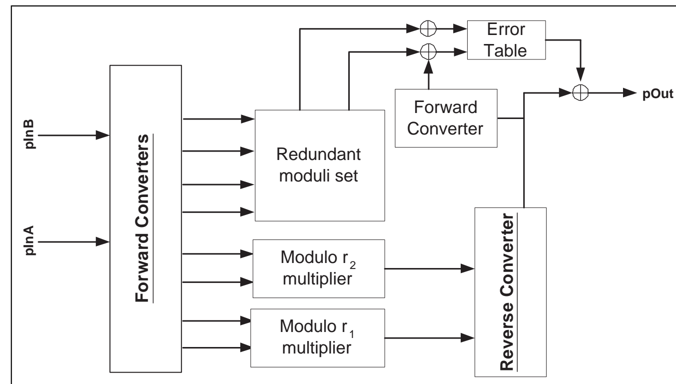

Figure 5.2: A modulo multiplier with error detection and coding

use sharable sub-blocks for accomplishing residue operations [45]. Variable Multifunction Architectures (VMA) that produce a residue modulo m_i that is selectable from a set $\{m_1, m_2 \cdots m_k\}$ achieve hardware savings at the cost of decreased parallelism. A shared-moduli architecture for moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ can act as a part of a VMA for a special moduli set.

4. Redundant structures for error detection and correction in RNS

   Error detection and correction using RNS [92, 93] relies on redundant residue channels. A general structure for an RNS modulo multiplier with error correction and detection is shown in Fig. 5.2. As can be seen from this figure, the non redundant channels go through a much longer path than the redundant channels to the output. Thus, the redundant channels will not be on the timing critical path. These redundant channels are thus prime candidates to be considered for shared moduli architectures.

5. Standalone Modulo arithmetic

   Modulo arithmetic outside of the RNS framework have applications in cryptography [94] such as the IDEA (International Data Encryption Algorithm) block cipher [11]. Such applications usually work on Mersenne numbers $(2^n - 1)$ and Fermat numbers $(2^n + 1)$. A configurable modulo arithmetic unit with shared moduli architectures that can accomplish computations for both Mersenne and Fermat numbers without much overhead can find use in cryptographic processors.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

5.3 Shared moduli architectures

Configurable modulo architectures that work for moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$ constitute shared moduli architectures in this thesis. This section articulates the framework for obtaining such architectures. A hierarchical building block approach has been followed. Thus firstly shared moduli architecture for modulo adders is presented. Going higher up the hierarchy, shared moduli architecture for multioperand modulo adders is presented. This is followed by the articulation of shared moduli architecture for modulo multipliers.

5.3.1 Shared moduli architecture for modulo adder

Modulo adders for moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ can take many forms. These architectures were discussed in detail in Chapter 3. While considering implementations of modulo adders that are amenable to sharing of architectures a few salient points are immediately obvious.

- Modulo $2^n + 1$ adders that work on the normal representation of modulo 2^n+1 numbers require adders that are at least $n+1$-bits wide. In some cases [4], the adders can be even wider. To use this representation for shared-moduli architectures is thus very inefficient as modulo adders for modulo $2^n - 1$ and modulo 2^n have no need of these superfluous bits. Modulo $2^n - 1$ and 2^n adders use n-bit wide adders and in order to have modulo $2^n + 1$ representation in the shared moduli architecture without adding too much overhead, the diminished-one representation is best suited.
- Modulo adders using carry unwrapping techniques for carry computation have structures that are not amenable to shared architectures. Unwrapping carries for either EAC or CEAC is based solely on the properties of EAC or CEAC and lead to structures that have either EAC or CEAC percolating down to all individual bits of the carry computation units. To switch between EAC and CEAC for such a structure is most inefficient as switching will need to be considered for each bit in the carry computation unit.

Modulo adders that rely on carry computation as direct implementation of EAC/CEAC suit shared moduli architectures owing to the need to switch only at the EAC/CEAC bit. Shared moduli architectures for modulo adders thus boil down to switching the end around carry bit. The structure of a prefix-based implementation of shared moduli architecture for modulo adder is shown

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

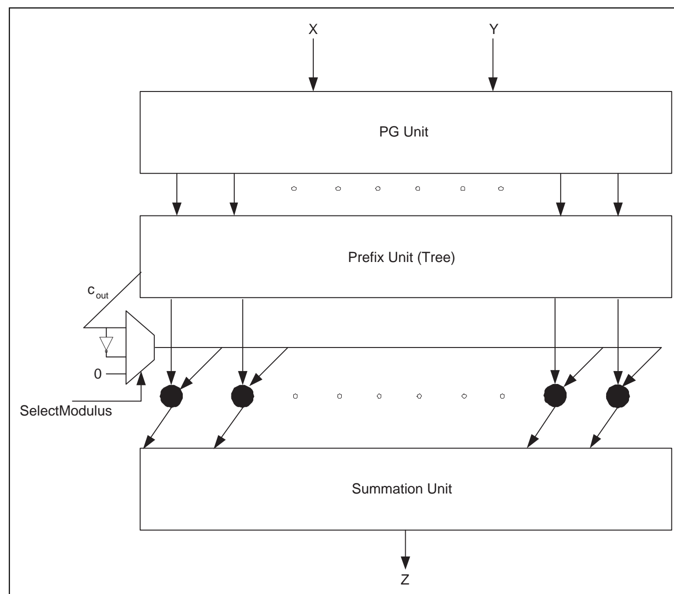

Figure 5.3: Shared moduli architecture for modulo adders using a prefix adder based implementation

in Fig. 5.3. CLA based implementations have a similar structure and is shown in Fig. 5.4.

As can be seen from the presented shared moduli modulo adders, the overhead incurred is only an additional level of multiplexers for switching the end around carry bit. This multiplexer is controlled by a signal that chooses the carry bit that is fed as end around. For modulo $2^n - 1$ addition, the carry out bit is fed as end around carry. For modulo $2^n + 1$ addition, the carry out bit is inverted before feeding it as end around carry, thus accomplishing the complemented end around carry operation. For modulo 2^n operation a zero is fed as end around carry.

5.3.2 Shared moduli architecture for MOMA

Multioperand modulo adder architectures that can be used for shared moduli implementations use multiplexers to switch between EAC and CEAC structures. Articulation of such architecture for MOMA uses a novel structure called the Composite Carry Save Adder (CCSA) which amalgamates the EAC and CEAC operation with the ordinary CSA [88]. The structure of CCSA is shown in Fig. 5.5.

Notice the two-input multiplexer that is used to select between the EAC and the CEAC. This multiplexer is controlled by an external input that can switch between the EAC and CEAC. Thus a common CCSA tree of the sort shown in Fig. 5.5 can replace the CSA tree for modulo adders belonging to moduli of the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

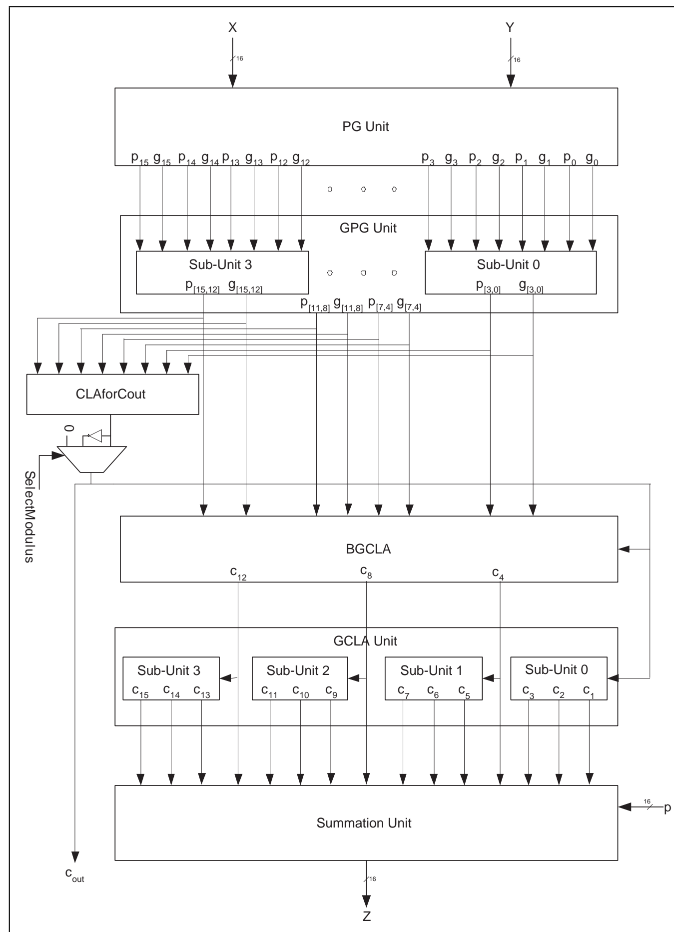

Figure 5.4: Shared moduli architecture for modulo adders using a CLA based implementation

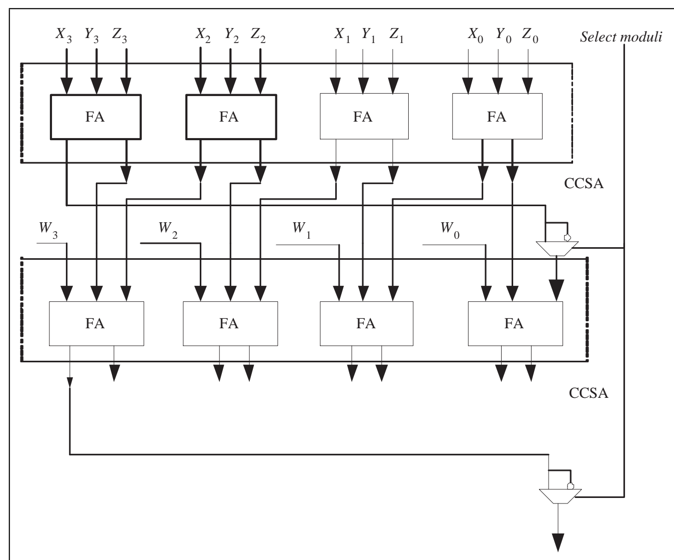

Figure 5.5: Architecture of a CCSA

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

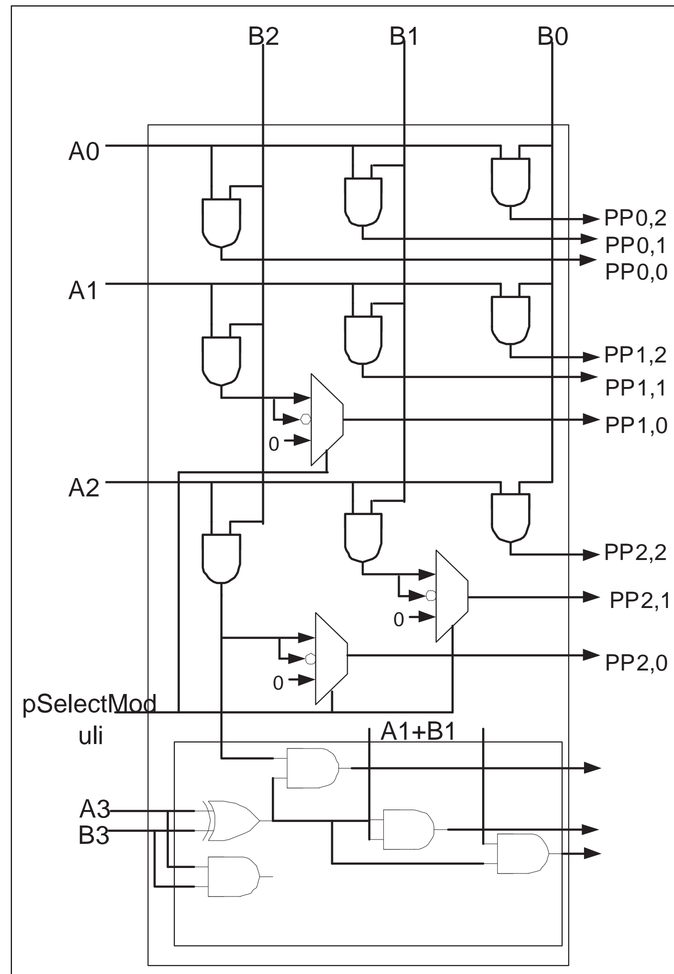

Figure 5.6: Shared moduli architecture for the PPG unit

set $\{2^n - 1, 2^n, 2^n + 1\}$. Overhead of only an additional multiplexer for each level of CCSA is incurred.

5.3.3 Shared moduli architectures for modulo multipliers

A modulo multiplier in shared moduli architecture needs to have all the three structures that make up the multiplier to be shared moduli structures. Shared moduli architectures for MOMA and two-operand modulo CPA have been presented in the following sections. The only structure that needs to be articulated as shared moduli architecture is the PPG Unit. A PPG unit based on the proposed novel modulo $2^n + 1$ adder in Chapter 4 can be modified to include the PPG for the modulo $2^n - 1$ adder as well. Such a composite PPG structure is shown in Fig. 5.6.

A complete standalone modulo multiplier for shared moduli operations thus

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

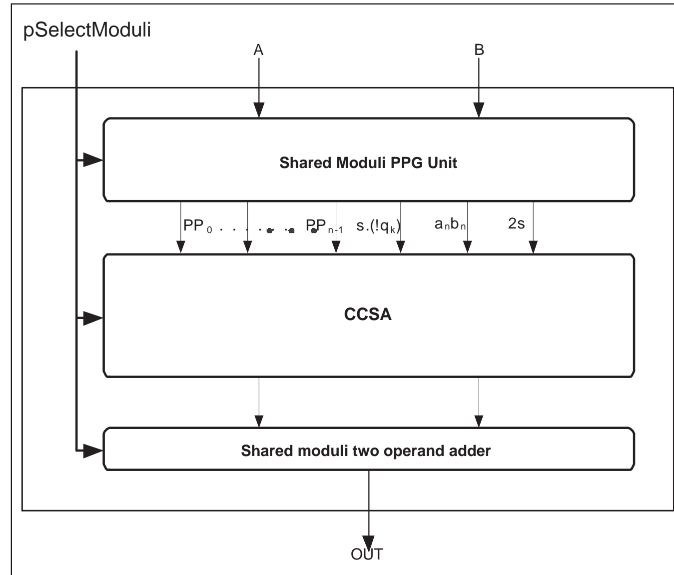

Figure 5.7: Shared moduli architecture for modulo multiplier

has the structure shown in Fig. 5.7. This modulo multiplier can function for moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$.

5.4 Experimental Results

The shared moduli multiplier proposed was synthesized using Synopsys Design Compiler for the TSMC 0.18μm standard cell library. Comparisons have been drawn between the worst performing single modulo multiplier (modulo $2^n + 1$ multiplier) and the shared moduli multiplier. This gives a very good estimate of the sort of overheads associated with shared moduli architectures. The results have been reported for different prefix structures.

Table. 5.4 shows the cell area results obtained. The results show that an average area overhead of 21.9% is incurred for shared moduli multipliers over modulo $2^n + 1$ multipliers. This overhead comes about mainly due to having additional multiplexers for switching. The worst case overhead is 30.74% and comes about for a bit width of 8 using the BK structure. When considering the area overhead, it needs to be understood that this area overhead is balanced by the fact that shared moduli architecture can perform modulo multiplication for any of the moduli from the set $\{2^n-1, 2^n, 2^n+1\}$. Thus a single shared moduli multiplier can replace three individual modulo multipliers and when this is considered the area overhead of the shared moduli multiplier is not a disadvantage but a low price for configurability and versatility.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Type	n	SKL	BK	KS
$2^n + 1$	8	21235.75	20320.98	20889.80
	16	67236.73	67552.77	69555.17
	32	236569.6	228273.2	244076.0
Shared Moduli	8	25413.7	26567.9	24948.0
	16	86220.4	86649.6	84590.5
	32	279023.0	272033.1	274080.8

Table 5.1: Cell area for modulo multipliers (in μm^2): shared moduli vs. modulo $2^n + 1$

Type	n	SKL	BK	KS
$2^n + 1$	8	3.05	3.12	3.11
	16	3.69	3.77	3.70
	32	4.63	4.77	4.74
Shared Moduli	8	3.22	3.22	3.28
	16	3.85	3.92	3.83
	32	4.76	4.93	5.06

Table 5.2: Critical path delay for modulo multipliers (in ns): shared moduli vs. modulo $2^n + 1$

Table. 5.4 tabulates the critical path delay results of the shared moduli multiplier compared against the slowest of the single moduli multiplier (the $2^n + 1$ multiplier). The results show an average overhead of 4.33% for critical path delay. This is very minimal and is incurred because of the multiplexers for switching. The worst case overhead is 6.75% and occurs for a multiplier of bit width 32 using a KS structure.

The performance overheads as denoted by critical path delay is thus very minimal and is not expected to overly diminish the performance of a system utilizing shared moduli architectures.

5.5 Summary

This chapter has presented shared moduli architectures, which can be defined as architectures that work for moduli selectable from the set of $\{2^n - 1, 2^n, 2^n + 1\}$. Experimental results have proven that performance tradeoff for such circuits is minimal. It has also been shown that there are various applications in which such shared moduli architectures can be very useful.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

Residue Number systems dealing with the residue representation of numbers form an important aspect of research into high performance DSP systems. Overheads associated with converters to/from the ordinary binary systems are usually overcome by the usage of moduli belonging to special sets of moduli. The most common of these sets is the set $\{2^n - 1, 2^n, 2^n + 1\}$. The popularity of moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ can be gaged from the fact that most modern implementations consider moduli of this type. Modulo arithmetic circuits are the basic building blocks of RNS and thus efficient design of such circuits will contribute towards building high performance RNS. Modulo adders and multipliers form the basis of RNS blocks like the RAU (Residue Arithmetic Unit), FC (Forward Converter) and RC (Reverse Converter).

Combining the observation that modulo arithmetic circuits are at the root of efficient RNS implementation and the inherent advantages of using moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$, this body of work is set up to design, develop and appraise various implementations of modulo adders and multipliers for moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$.

A multitude of modulo adders and multipliers exist for moduli of type $\{2^n - 1, 2^n, 2^n + 1\}$ and this translates to varied choice for RNS system design. However, a standardized framework for a VLSI evaluation of these architectures is not available. This work started out as an effort to plug this gap by providing a standardized evaluation of commonly used modulo adders and multipliers. Evaluation using a standard cell pre layout implementation on the TSMC 0.18 μm

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

library was done for well known modulo adder and multiplier architectures. The evaluation of modulo adders and multipliers showed that a performance gap exists between modulo arithmetic circuits for modulo $2^n + 1$ operands and modulo arithmetic circuits for the moduli 2^n and $2^n - 1$. A novel modulo $2^n + 1$ multiplier was proposed as a consequence of this analysis. Performance evaluation using VLSI performance metrics of area/power/timing showed that this modulo $2^n + 1$ multiplier outperforms the other well known modulo $2^n + 1$ multipliers for high operand sizes.

An important aspect of evaluation of a multitude of architectures is the ability to generate EDA tool readable circuit descriptions in an automated and parameterizable manner. The articulation of RNS system design as a building block approach leads to the adoption of an Object Oriented approach to building such a code generator. In this work, we have built a code generator called BuRNS (Build RNS). This tool presently generates synthesizable Verilog RTL codes. This generator was built using PERL language.

Moduli arithmetic architectures that can be shared among multiple moduli can have interesting applications in RNS, where some performance sacrifices might be acceptable to gain efficiency in terms of area usage for configurability. Some such applications were discussed in Chapter 5. In particular, shared-moduli architectures for moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$ can find uses in these applications. A proposed framework for shared-moduli architectures among moduli belonging to the set $\{2^n - 1, 2^n, 2^n + 1\}$ was presented. This is expected to lead to efficient shared-moduli architectures for the $\{2^n - 1, 2^n, 2^n + 1\}$ moduli set.

6.2 Future Work

The future effort that can be foreseen as a result of the work done so far can be encapsulated as follows.

- While possible applications for the shared moduli architectures have been presented, an evaluation of the effect of shared moduli architectures in these applications using VLSI metrics need to be done. This will complete the work on shared moduli architectures for moduli selectable from the set $\{2^n - 1, 2^n, 2^n + 1\}$ by providing detailed information on the effects of using shared moduli structures. Such a study will thus set out the range and type of applications for which shared moduli structures provide the best value.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- BuRNS at the present moment generates RTL Verilog code for modulo adders and multipliers. The object oriented philosophy being adopted now can be extended to generation of reverse and forward converters too. With that BuRNS can be a standalone design and evaluation tool for RNS architectures. Additional improvements like adding a Graphical User Interface (GUI) can also be done to increase the ease of use of BuRNS

- There still exists a substantial performance gap between modulo arithmetic circuit for moduli of type 2^n+1 and other special moduli. This translates to the fact that the critical path of any design using special moduli belonging to the type $\{2^n-1, 2^n, 2^n+1\}$ is through the 2^n+1 channel. More effort needs to be put into efficient design of modulo arithmetic for moduli of type 2^n+1 to increase the performance of RNS systems as a whole.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Author's Publications

C. H. Chang, S. Menon, B. Cao and T. Srikanthan, "A configurable dual moduli multi-operand modulo adder," in *Proc. IEEE Int. Symp. on Circuits and Systems* (ISCAS-2005), Kobe, Japan, pp. 1 630-1633, May 23-26, 2005.

S. Menon and C. H. Chang, "A reconfigurable multi-modulus modulo multiplier," in *Proc. 2006 IEEE Asia-Pacific Conf. on Circuits and Systems* (APCCAS-2006), Singapore, pp. 1170-1173, December 4-7, 2006.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Bibliography

[1] M. Bayoumi, G. Jullien, and W. Miller, "A VLSI implementation of residue adders," *IEEE Trans. Circuits Syst.*, vol. 34, pp. 284–288, March 1987.

[2] M. Dugdale, "VLSI implementation of residue adders based on binary adders," *IEEE Trans. Circuits Syst. II: Analog and Digital Signal Processing*, vol. 39, pp. 325–329, May 1992.

[3] K. M. Elleithy and M. A. Bayoumi, "A θ (1) algorithm for modulo addition," *IEEE Trans. Circuits Syst.*, vol. 37, pp. 628–631, May 1990.

[4] A. Hiasat, "High-speed and reduced-area modular adder structures for RNS," *IEEE Trans. Comput.*, vol. 51, pp. 84–90, January 2002.

[5] L.Kalampoukas, D.Nikolos, C.Efstathiou, H.T.Vergos, and J.Kalamantianos, "High speed parallel-prefix modulo $2^n - 1$ adders," *IEEE Trans. Comput.*, vol. 49, pp. 673–680, July 2000.

[6] H.T.Vergos, C.Efstathiou, and D.Nikolos, "Diminished-one modulo $2^n + 1$ adder design," *IEEE Trans. Comput.*, vol. 51, pp. 1389–1399, December 2002.

[7] C. Efstathiou, H. Vergos, and D. Nikolos, "Handling zero in diminished-one modulo $2^n + 1$ adders," *International Journal of Electronics*, vol. 90, pp. 133–144, February 2003.

[8] G. Alia and E. Martinelli, "A VLSI modulo m multiplier," *IEEE Trans. Comput.*, vol. 40, pp. 873–878, July 1991.

[9] T. Stouraitis, S. W. Kim, and A. Skavantzos, "Full-adder based arithmetic units for finite integer rings," *IEEE Trans. Circuits and Syst. - II*, vol. 40, pp. 740–745, Nov 1993.

[10] A. A. Hiasat, "New efficient structure of a modular multiplier for RNS," *IEEE Trans. Comput.*, vol. 49, pp. 170–174, Feb 2000.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[11] R. Zimmermann, "Efficient VLSI implementation of modulo $2^n \pm 1$ addition and multiplication," in *Proc. 14th IEEE Symp. Computer Arithmetic*, pp. 158–167, April 1999.

[12] C.Efstathiou, H.T.Vergos, G.Dimitrakopoulos, and D.Nikolos, "Efficient diminished-one modulo $2^n + 1$ multipliers," *IEEE Trans. Comput.*, vol. 54, pp. 491–496, Apr 2005.

[13] A. Wrzyszcz and D. Milford, "A new modulo $2^a + 1$ multiplier," *Proc. Intl Conf. Computer Design*, pp. 614–617, 1993.

[14] G. C. Cadarillli, A. D. Re, A. Nannarelli, and M. Re, "Residue number system reconfigurable datapath," in *Proc. IEEE Int. Symp. on Circuits and Syst.*, vol. 2, pp. 756–759, May 2002.

[15] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing : A practical approach*. New York: Prentice Hall, 2nd ed., 2002.

[16] N.S.Szabo and R.I.Tanaka, *Residue Arithmetic and its Applications to Computer Technology*. New York: McGraw Hill, 1967.

[17] B. Parhami, *Computer arithmetic : Algorithms and hardware designs*. New York: Oxford University Press, 2000.

[18] A. Mohan, *Residue Number Systems : Algorithms and Applications*. Boston: Kuwer Academic Publishers, 2002.

[19] C.Efstathiou, D.Nikolos, and J.Kalamatianos, "Area-time efficient modulo $2^n - 1$ adder design," *IEEE Trans. Circuits Syst.*, vol. 41, pp. 463–467, July 1994.

[20] W. Wang, M. Swamy, M.O.Ahmad, and Y. Wang, "A comprehensive study of three moduli sets for residue arithmetic," in *Proc. 1999 IEEE Canadian conference on Electrical and Computer Engineering*, pp. 513–518, May 1999.

[21] B. Cao, *VLSI efficient architectures for triple moduli based RNS computations*. PhD thesis, Nanyang Technological University, 2006.

[22] Y.Wang, M.N.S.Swamy, and M. O. Ahmad, "Three number moduli sets based residue number systems," *IEEE Trans. Circuits Syst. II*, vol. 46, Feb 1999.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[23] D. Gallaher, F. E. Petry, and P. Srinivasan, "The digit parallel method for fast RNS to weighted number system conversion for special moduli $(2^n - 1, 2^n, 2^n + 1)$," *IEEE Trans. Circuits Syst. II*, vol. 44, pp. 53–57, January 1997.

[24] 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. Circuits Syst. I*, vol. 50, pp. 1296–1303, October 2003.

[25] Y.Wang, X.Song, M.Aboulhamid, and H.Shen, "Adder based residue to binary number converters for $(2^n - 1, 2^n, 2^n + 1, 2^{2n} + 1)$," *IEEE Trans. Signal Processing*, vol. 50, pp. 1772–1779, July 2002.

[26] A. P. Vinod and A. B. Premkumar, "A memoryless reverse converter for the 4-moduli superset $2^n - 1, 2^n, 2^n + 1, 2^{n+1} + 1$," *Journal of Circuits, Systems, and Computers*, vol. 10, no. 1 & 2, pp. 85–89, 2000.

[27] V. Paliouras and T. Stouraitis, "Novel high-radix residue number system architectures," *IEEE Trans. Circuits Syst. - II: Analog and Digital Signal Processing*, vol. 47, pp. 1059–1073, Oct 2000.

[28] R.Zimmerman, *Binary adder architectures for cell-based VLSI and their synthesis*. PhD thesis, Swiss Federal Institute of technology, 1998.

[29] K. Hwang, *Computer Arithmetic : Principle, Architecture and Design*. New York: Wiley, 1979.

[30] J. Sklansky, "Conditional sum addition logic," *IRE. Trans. Electron. Comput.*, vol. 9(6), pp. 226–231, June 1960.

[31] R. Brent and H. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. 31(3), pp. 260–264, March 1982.

[32] P.M.Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22(8), pp. 783–791, August 1973.

[33] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc. 8th Computer Arithmetic Symp.*, pp. 49–56, May 1987.

[34] S. Knowles, "A family of adders," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 30–34, 1999.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[35] R.E.Ladner and M.J.Fisher, "Parallel prefix computation," *J. of the ACM*, no. 27, pp. 831–838, 1980.

[36] H. Ling, "High-speed binary adder," *IBM J. Research and Development*, vol. 25, pp. 156 – 166, 1981.

[37] G. Dimitrikapoulos and D. Nikolos, "High-speed parallel prefix vlsi ling adders," *IEEE Trans. Comput.*, pp. 22–22, Feb 2005.

[38] I. Koren, *Computer Arithmetic Algorithms.* Prentice-Hall, 1993.

[39] C.S.Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Comput.*, vol. 13, pp. 14–17, 1964.

[40] L.Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[41] A. Booth, "A signed binary multiplication technique," *Quarterly J. Mechanics and Applied Mathematics*, vol. 4, pp. 236–240, June 1951.

[42] B. Parhami and C. Huang, "Optimal table lookup schemes for VLSI implementation of input/output conversions and other residue number operations," *Workshop on VLSI Signal Processing VII*, pp. 470–482, 1994.

[43] F. Pourbigharaz and H. M. Yassine, "Simple binary to residue transformation with respect to $2^m + 1$ moduli," vol. 141, pp. 52–526, December 1994.

[44] A.B.Premkumar, "A formal framework for conversion from binary to residue numbers," *IEEE Trans Circuits Syst. II*, vol. 49, pp. 135–144, February 2002.

[45] V. Paliouras and T. Stouraitis, "Multifunction architectures for RNS processors," *IEEE Trans Circuits Syst. II*, vol. 46, pp. 1041–1054, August 1999.

[46] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Comput.*, vol. 423, pp. 68–77, January 1994.

[47] A.B.Premkumar, E.L.Ang, and E. M. K. Lai, "Improved memoryless RNS forward converters based on the periodicity of residues," *IEEE Trans Circuits Syst. II: Express briefs*, vol. 53, pp. 133–137, February 2006.

[48] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, eds., *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[49] S. J. Meehan, S. D. O'Neil, and J. J. Vaccaro, "An universal input and output RNS converter," *IEEE Trans. Circuits Syst.*, vol. 37, pp. 799–803, June 1990.

[50] G. C. Cardiralli, M. R. R. Lojacono, and G. Ferri, "A systolic architecture for high-performance scaled residue to binary conversion," *IEEE Trans. Circuits Syst. I*, vol. 47, pp. 1523–1526, October 2000.

[51] K. M. Elleithy, M. A. Bayoumi, and K. P. Lee, "$\theta(logn)$ architectures for RNS arithmetic decoding," No. 202-209, pp. 1523–1526, September 1989.

[52] Y. Wang, "Residue-to-binary converters based on new chinese remainder theorems," *IEEE Trans. Circuits Syst. II*, vol. 47, pp. 197–205, March 2000.

[53] C. H. Huang, "A fully parallel mixed-radix conversion algorithm for residue number applications," *IEEE Trans. Comput.*, vol. C-32, pp. 398–402, April 1983.

[54] H. M. Yassine and W. R. Moore, "Improved mixed-radix conversion for residue number system architectures," in *IEE Proc. -G*, vol. 138, pp. 120–124, February 1991.

[55] A. Hiasat and H. Abdel-Aty-Zohdy, "Residue to binary arithmetic converter for the moduli set $(2^k, 2^k-1, 2^{k-1}-1)$," *IEEE Trans. Circuits Syst.*, vol. 45, pp. 204–208, Feb 1998.

[56] 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 $2^n-1, 2^n, 2^n+1, 2^{n-1}-1$," in *IEE Proceedings, Computers and Digital Techniques*, pp. 687–696, September 2005.

[57] B. Cao, T. Srikanthan, and C. H. Chang, "A new design method to modulo 2^n-1 squaring," in *Proc. 2005 IEEE Int. Symp. on Circuits and Systems*, pp. 664–667, May 2005.

[58] B. Cao, C. H. Chang, and T. Srikanthan, "A new formulation of fast diminished-one multioperand modulo 2^n-1 adder," in *Proc. 2005 IEEE Int. Symp. on Circuits and Systems*, pp. 656– 659, May 2005.

[59] B. Cao, C. H. Chang, and T. Srikanthan, "A residue-to-binary converter for a new 5-moduli set," *IEEE Trans. Circuits Syst.-I: Regular Papers (Accepted for publication)*, 2006.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[60] A. B. Premkumar, “An RNS to binary converter in $2n+1, 2n, 2n-1$ moduli set,” *IEEE Trans. Circuits Syst. II*, vol. 39, pp. 480–482, July 1992.

[61] A. B. Premkumar, M. Bhardwaj, and T. Srikanthan, “High-speed and low cost reverse converters for the $2n-1, 2n, 2n+1$ moduli set,” *IEEE Trans. Circuits Syst. II*, vol. 45, pp. 903–908, July 1998.

[62] M. Bhardwaj, T. Srikanthan, and C. T. Clarke, “A reverse converter for the 4-moduli superset $2^n-1, 2^n, 2^n+1, 2^{n+1}+1$,” in *Proc. 14th IEEE Symposium on Computer Arithmetic*, pp. 168–175, April 1999.

[63] D.J.Soudris, M.M.Dasigenis, S.K.Vasilopoulou, and A.T.Thanailakis, “A cad tool for architecture level exploration and automatic generation of rns converters,” in *Proc. IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 730 – 733, May 2001.

[64] H.Henkelmann, A.Drolshagen, H.Bagherinia, H.Ahrens, and W.Anheier, “Automated implementation of RNS-to-binary converters,” in *Proc. of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 137 – 140, June 1998.

[65] N. Kostaras and H. Vergos, “Kover : A sophisticated residue arithmetic core generator,” *16th IEEE International Workshop on Rapid System Prototyping (RSP2005)*, pp. 261–263, June 2005.

[66] C. B. K. Hong, *PERL 5 tutorial.* www.ckihong.com, 1st ed., 2003.

[67] D.E.Thomas and Philip.R.Moorby, *The Verilog Hardware Description Language.* New York: Kluwer academic publishers, 1998.

[68] *Design Compiler User Guide V2004.12.*

[69] W.K.Jenkins, “A highly efficient residue-combinatorial architecture for digital filters,” in *Proc. IEEE*, vol. 66, pp. 700–702.

[70] F.Taylor, “A VLSI residue arithmetic multiplier,” *IEEE Trans. Comput.*, vol. C-31, pp. 540–546, June 1982.

[71] G. Dimitrakopoulos, D. Nikolos, H.T.Vergos, D.Nikolos, and C.Efstathiou, “New architectures for modulo 2^n-1 adders,” *12th IEEE Conference on Electronics, Circuits and Systems*, December 11-14 2005.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[72] D.Nikolos, A.M.Paschalis, and G.Philokyprou, "Efficient design of totally self-checking self checkers for all low cost arithmetic codes," *IEEE Trans. Comput.*, vol. C-37, pp. 807–814, July 1988.

[73] C. Efstathiou, H. Vergos, and D. Nikolos, "Fast parallel-prefix modulo 2^n+1 adders," *IEEE Trans. Comput*, vol. 53, September 2004.

[74] L. Leibowitz, "A simplified binary arithmetic for the fermat number transform," *IEEE Trans. on Accoustics, speech and signal processing*, vol. ASSP-24, pp. 356–359, October 1976.

[75] H. Vergos, C. Efstathiou, and D. Nikolos, "Fast parallel-prefix modulo 2^n+1 adders," *XVII Conference on Design of Circuits and Integrated Systems*, November 19-22 2002.

[76] M. Dugdale, "Residue multipliers using factored decomposition," *IEEE Trans. Circuits and Systems - II: Analog and Digital processing*, vol. 41, pp. 623–627, Sep 1994.

[77] G.A.Jullien, "Implementation of multiplication modulo a prime number with appications to number theoretic transforms," *IEEE Trans. Comput.*, vol. C-29, pp. 899–905, Oct 1980.

[78] D. Radhakrishnan and Y. Yuan, "A fast RNS galois field multiplier," in *Proc. 1990 IEEE International Symposium on circuits and systems*, vol. 4, pp. 2909–2912, May 1990.

[79] Y. Ma, "A simplified architecture for modulo $2^n + 1$ multiplication," *IEEE Trans. Comput.*, vol. 47, pp. 333–337, March 1998.

[80] A. Hiasat, "A memoryless mod $2^n \pm 1$ residue multiplier," *Electronics Letters*, vol. 28, pp. 314–315, March 1992.

[81] L. Sousa and R. Chaves, "A universal architecture for designing efficient modulo $2^n + 1$ multipliers," *IEEE Trans. Circuits Syst. - I: Regular papers*, vol. 52, pp. 1166–1178, June 2005.

[82] A. V. Curiger, H. Bonnenberg, and H. Kaeslin, "Regular VLSI architectures for multiplication modulo 2^n+1," *IEEE Journal of solid state circuits*, vol. 26, pp. 990–994, July 1991.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[83] M. A. Soderstrand and C. Vernia, "A high-speed low-cost modulo pi multiplier with RNS arithmetic applications," *Proc. IEEE*, vol. 68, pp. 529–532, Apr 1980.

[84] C. Efstathiou, H. Vergos, and D. Nikolos, "Modified booth modulo $2^n - 1$ multipliers," *IEEE Trans. Comput.*, vol. 53, pp. 370–374, March 2004.

[85] C. Efstathiou and H. Vergos, "Modified booth 1's complement and modulo $2^n - 1$ multipliers," *7th IEEE International Conference on Electonics, Circuits and Systems (ICECS)*, vol. 4, pp. 637–640, December 2000.

[86] A. Skavantzos, "New multipliers modulo $2^n - 1$," *IEEE Trans. Comput.*, vol. 41, pp. 957–961, Aug 1992.

[87] Z. Wang, G. A. Jullien, and W. C. Miller, "An algorithm for multiplication modulo $(2^n - 1)$," *EEE 39th Midwest Symp. on Circuits and Systems*, vol. 3, pp. 1301–1304, Aug 1996.

[88] S. Menon and C. H. Chang, "A reconfigurable multi-modulus modulo multiplier," in *Proc. 2006 IEEE Asia Pacific Conference on Circuits and Systems*, pp. 1170–1173, December 2006.

[89] H. Vergos and C. Efstathiou, "Novel modulo $2^n + 1$ multipliers," *9th Euromicro conference on Digital System Design*, pp. 561–566, September 2006.

[90] H. Vergos and C. Efstathiou, "Design of efficient modulo $2^n + 1$ multipliers," *IET Computers and Digital techniques*, vol. 1, pp. 49–57, January 2007.

[91] W. Jenkins, B. Schnaufer, and A. Mansen, "Combined system-level redundancy and modular arithmetic for fault tolerant digital signal processing," *Proc. IEEE Int. Symp. on Computer Arithmetic*, pp. 28–35, June 1993.

[92] M. H. Etzel and W. K. Jenkins, "Redundant residue number systems for error detection and correction in digital filters," *IEEE Trans. Accoustics, Speech and Signal Processing*, vol. ASSP-28, pp. 538–545, Oct 1980.

[93] A. P. Preethy, D. Radhakrishnan, and A. Omondi, "Fault-tolerance scheme for an RNS MAC: performance and cost analysis," in *Proc. IEEE Int. Symp. on Circuits and Syst.*, vol. 2, pp. 717–720, May 2001.

[94] X. Lai and J. L. Massey, *A proposal for a new block encryption standard.* Berlin: SpringerVerlag, 1990.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix A

Synthesis scripts

```
# Set search path
set search_path  [concat . \$env(SYNOPSYS)/libraries/syn/ \
~/Artisan_library/fe_tpz973g_230b/TSMCHOME/digital/synopsys/
    tpz973g_230a \
~/Artisan_library/aci/sc/synopsys]
# Set target library
set target_library  [concat slow.db  tpz973gtc.db ]
# Set synththetic_library
set synthetic_library  [list dw01.sldb dw02.sldb dw03.sldb dw06.
    sldb]
# Set the link library
set link_library [concat {*} \$target_library \$synthetic_library]
# Define design library
define_design_lib WORK -path ./SynWORK
source set_design_parameter.scr
set power_preserve_rtl_hier_names TRUE
# Analyze the files in the design list and elaborate to GTECH
foreach MODULE "\$DESIGN_LIST" {
  analyze -format verilog -lib WORK      \
    \${MODULE}
}
elaborate \${TOP_DESIGN}
set_structure false -design [get_designs *]
# Set constraints
set_max_delay 1 -from [all_inputs] -to [all_outputs]
set_max_area 0
set_operating_conditions slow
set_wire_load_mode top
set_wire_load_model -name tsmc18_wl30
set_load [expr [load_of slow/BUFX4/A] * 4] [all_outputs]
set_drive [expr [drive_of slow/BUFX4/Y] * 4] [all_inputs]
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library


```
# Removes multiply-instantiated hierarchyin the current_design by
    creating a
# unique design for each cell instance
uniquify
# Resolve design references
link
# Output RTL SAIF
rtl2saif -output rtl_fwd.saif
# Perform logic and gate level synthesis and optimization on the
    current design
compile -map_effort medium
# Report Timing and Area
report_timing -delay max -nworst 1 > syn_output/report_timing.log
report_area > syn_output/report_area.log
# Write synthesized netlist
write -f verilog -hier -o syn_output/top_netlist.v
# Write output db file
write -f db -o syn_output/top.db
# Quit
exit
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix B

Power Analysis scripts

```
# Set search pat
set search_path  [concat . \$env(SYNOPSYS)/libraries/syn/ \
~/Artisan_library/fe_tpz973g_230b/TSMCHOME/digital/synopsys/
    tpz973g_230a \
~/Artisan_library/aci/sc/synopsys]
# Set libraries for DC
set target_library  [concat slow.db tpz973gtc.db ]
set synthetic_library  [list dw01.sldb dw02.sldb dw03.sldb dw06.
    sldb]
set link_library [concat {*} \$target_library \$synthetic_library]
#Define design library
define_design_lib WORK -path ./SynWORK
source set_design_parameter.scr
# Read the synthesized netlist
read_verilog syn_output/top_netlist.v
link
current_design  \${TOP_DESIGN}
# Read the back-annotated SAIF
read_saif -input sim_back.saif -instance_name \${TOP_DESIGN}_tb/DUT
report_saif -type rtl > syn_output/report_saif.log
# Report Power
report_power -verbose > syn_output/report_power.log
#Quit
exit
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix C

Simulation Scripts

```
vmap work SimWORK
vlog *.v
source set_design_parameter.scr
vsim -pli /synopsys/2004.06_SP2/auxx/syn/power/vpower/lib-sparcOS5/
    libvpower.so \${TOP_DESIGN}_tb
run -all
quit
```