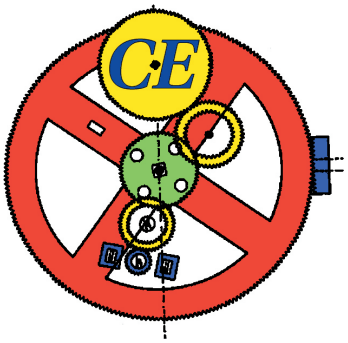

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Cost Effective Modular Adders for RNS-based Processors

Ondy Dharma Indra Sukma

CE-MS-2010-2010-26

Abstract

RNS can distribute the computation on long operands over small word-width RNS functional units able to operate in parallel. This property is the ground to develop fast arithmetic units. While RNS can boost addition and multiplication performance, other arithmetic operations like division, magnitude comparison, and sign detection are more difficult, when compared with their counterparts in the conventional binary number system. In view of that RNS is mostly utilized for special-purpose applications, e.g., digital filters, which are addition and multiplication dominated. For such applications the RNS capability to represent large numbers and the carry-free nature of arithmetic operations are of interest and can potentially enable fast and low-power arithmetic computation. The overall performance of any RNS based processor is mostly determined by the selected moduli set and the way the modular operations, i.e., addition and multiplication, are implemented in hardware (note that this two issues are intertwined). In this thesis we concentrate on the design of fast and energy effective modular adders able to compute $|A+B|_m = (A+B$ if $A+B<m$, if otherwise $A+B-m)$ as they are the fundamental building block for any RNS processor. We base our solution on a state of the art approach, i.e., ELM Modular Addition (ELMMA), which utilize anticipated computation in conjunction with fast parallel prefix addition. Our method follows the same anticipation principle but reduces the overall complexity by proposing an alternative design for the adders, which can now directly handle three inputs instead of two. In this way the initial carry-save addition required for ELMMA for the evaluation of the A+B-m is not longer required and this may potentially result in faster and more area and power effective designs. To evaluate the impact of our proposal we considered a number of moduli of practical interest as follows: $2^n-(2^{n-2}+1)$, 2^n-2^{n-2}, and $2^n-(2^{n-3}+1)$. For the considered moduli we implemented in VHDL two sets of implementations, i.e., one for the state of the art ELMMA and one for our proposal, for the n=16 case. We simulated, debug, and synthesized the designs using Cadence Encounter RTL Compiler for ASIC Designs for 90 nm CMOS technology. Our results indicate that for moduli $2^n-(2^{n-2}+1)$, 2^n-2^{n-2}, and $2^n-(2^{n-3}+1)$, our proposal requires 13%, 32%, and 28% smaller area, is 14%, 3%, and 9% faster, and is 15%, 20%, and 13% more power efficient, respectively, when compared with the state of the art.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Cost Effective Modular Adders for RNS-based Processors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ondy Dharma Indra Sukma
born in Medan, Indonesia

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

by Ondy Dharma Indra Sukma

Abstract

In the recent years Residue Number Systems (RNS) have received increased interest due to their ability to limit the carry propagation chain thus to enable parallel and fast arithmetic. Within RNS any integer is represented with a set of its residues with respect to a given base that comprises a set of relatively prime integers. In this way RNS can distribute the computation on long operands over small word-width RNS functional units able to operate in parallel. Moreover the RNS representation provides some intrinsic support for fault tolerance as it isolates the individual digits, thus potential errors cannot affect other digits. The first property is the ground to develop fastarithmetic units, whereas the later one can be utilized to increase the system fault tolerance. While RNS can boost addition and multiplication performance, other arithmetic operations like division, magnitude comparison, and sign detection are more difficult, when compared with their counterparts in the conventional binary number system. In view of that RNS is mostly utilized for special-purpose applications, e.g., digital filters, which are addition and multiplication dominated. For such applications the RNS capability to represent large numbers and the carry-free nature of arithmetic operations are of interest and can potentially enable fast and low-power arithmetic computation. The overall performance of any RNS based processor is mostly determined by the selected moduli set and the way the modular operations, i.e., addition and multiplication, are implemented in hardware (note that this two issues are intertwined). In this thesis we concentrate on the design of fast and energy effective modular adders able to compute

$$|A+B|_m = \begin{cases} A+B & \text{if } A+B < m \\ A+B-m & \text{otherwise} \end{cases}$$

as they are the fundamental building block for any RNS processor. We base our solution on a state of the art approach, i.e., ELM Modular Addition (ELMMA), which utilize anticipated computation in conjunction with fast parallel prefix addition. Our method follows the same anticipation principle but reduces the overall complexity by proposing an alternative design for the adders, which can now directly handle three inputs instead of two. In this way the initial carry-save addition required for ELMMA for the evaluation of the A+B-m is not longer required and this may potentially result in faster and more area and power effective designs. To evaluate the impact of our proposal we considered a number of moduli of practical interest as follows: $2^n - (2^{n-2}+1)$, $2^n - 2^{n-2}$, and $2^n - (2^{n-3}+1)$. For the considered moduli we implemented in VHDL two sets of implementations, i.e., one for the state of the art ELMMA and one for our proposal, for the n=16 case. We simulated, debug, and synthesized the designs using Cadence Encounter RTL Compiler for ASIC Designs for 90 nm CMOS technology. Our results indicate that for moduli $2^n - (2^{n-2}+1)$ our proposal requires 13% smaller area, is 14% faster, and is 15% more power efficient when compared with the state of the art. For the moduli set $2^n - 2^{n-2}$ and for the moduli set $2^n - (2^{n-3}+1)$, our proposal is 3% and 9% faster, requires 32% and 28% lesser area cost, and consumes 20% and 13% lesser power, respectively, when compared with the state of the art.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-2010-26

Committee Members :

Advisor:	Sorin Cotofana, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Rene van Leuken, CE, TU Delft
Member:	Arjan van Genderen, CE, TU Delft
Member:	Sorin Cotofana, CE, TU Delft

To my wife and daughter for their unflagging support

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I offer my heartily gratitude to my supervisor, Sorin Cotofana, who has guided me through the completion of my thesis and working in my own way. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis would not have been completed or written. My sincere thanks also goes to Kazeem Gbolagade for the fruitful discussion and thesis document feedback despite distance working.

I would like to thank my dearly love wife and daughter for being there as the source of inspiring spirit to complete this thesis.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Ondy Dharma Indra Sukma
Delft, The Netherlands
August 26, 2010

1 Introduction

1.1 Motivation

A Residue Number System (RNS) is characterized by a base which is an N-tuple of relatively prime integers (m_{N-1}, $m_{N-2}, ..., m_0$) and each m_i ($i = 0, 1, 2, ..., N-1$) is called a modulus. An integer X is represented in this system by an N-tuple(x_{N-1}, $x_{N-2}, ..., x_0$) where x_i is a non-negative integer number, computed as the difference of the X and $m_i \times q_i$, where q_i is the largest integer such that $0 \leq x_i \leq (m-1)$. The residue of x_i with respect to the i^{th} modulus m_i is akin to a digit and the entire N residue representation of X can be viewed as an N-digit number, where the digit set for the i^{th} position is $[0, m_1 - 1]$.

RNS has two most significant advantageous properties: (i) no carry propagation may occur between the RNS digits during arithmetic operations as addition and multiplication, and, (ii) potential errors cannot pass the digit boundaries due to the very nature of the RNS representation with isolates individual digits. The first property provides the ground to develop fast arithmetic units, whereas the later one can be utilized to increase the system fault tolerance. While RNS can boost addition and multiplication performance, other arithmetic operations like division, magnitude comparison, and sign detection are more difficult [5], when compared with their counterparts in the conventional binary number system. The difficulty is induced by the fact that the RNS digit does not store any information related to its relative weight to other existing digits, a property which is utilized to easily compare two numbers, to detect the sign a number based on its relative magnitude to another number magnitude, and the combination of those two in order to perform overflow detection. An example to this problem is given as follows: Given a range of [-420, 419], the sorting result in ascending order of $a = (0|1|3|2)_{RNS}$, $b = (0|1|4|1)_{RNS}$, $c = (0|6|2|1)_{RNS}$, $d = (2|0|0|2)_{RNS}$, $e = (5|0|1|0)_{RNS}$, and $f = (7|6|4|2)_{RNS}$ is $d(-70) < c(-8) < f(-1) < a(8) < e(21) < b(64)$. In order to obtain correct sorting order, RNS number must be converted to digit weighted position, leading to large overhead. For division, more problematic situation occurs when the quotient must be denoted as a floating point number.

In view of the previous discussion, RNS is currently not utilized in general computing purpose but represents an attractive choice for specific-purposed applications which are addition and multiplication dominated and requires robustness against error. A leading candidate for the former is in the field of digital signal processing. Finite impulse response filter and Discrete Fourier Transform are some examples of application which heavily uses addition and multiplication. The later is an area that was the subject of much research in early computing era, but faded out from main stream research as the fabrication (CMOS) technology become more reliable. Now we entered the nano-era and the need of fault tolerance increases again due to the fact that devices are less reliable and exhibit large

delay variations thus we can expect a revival of the fault tolerant related RNS research. In the context of this thesis we concentrate on the RNS capability to provide support for fast and energy effective computer arithmetic and we do not dive into fault-tolerant related issues.

The overall performance of any RNS based processor is mostly determined by the selected moduli set and the way the modular operations, i.e., addition and multiplication, are implemented in hardware. The two issues are very much related as the moduli nature has a large influence on the structure and complexity of the functional units hardware. It is well known that some moduli, e.g., 2^n and $2^n - 1$, result in simple modular adders however one cannot limit the moduli choice only to such restricted moduli due to other reasons, e.g., the requirement to have a well balanced RNS system. Another challenging circumstance that RNS is facing is the need to represent large number involved in computation. Referring to the definition of RNS given in the beginning of this chapter, the number of different representation of N-tupple RNS, i.e, the system dynamic range, is equal to the product of its moduli, by assuming that all moduli are pairwise relatively prime one to each other. In order to enable RNS representations of large numbers one need to increase the the dynamic range which results in more moduli and/or in larger magnitude for the digits. One can easily observe that the increase of digit magnitude has a negative effect on the RNS processor performance as it increases the per digit position computation delay, since even though RNS has no inter digit carries, carry propagation occur at the digit level and the larger the maximum digit magnitude the slower the calculation. In view of this fast and optimized conventional binary arithmetic unit are essential for RNS processors.

In this thesis we concentrate on the design of fast and energy effective modular adders able to compute

$$|A+B|_m = \begin{cases} A+B & \text{if } A+B < m \\ A+B-m & \text{otherwise} \end{cases}$$

as they are the fundamental building block for any RNS processor.

Up to date many studies have been conducted on fast addition and parallel prefix seems to be the most effective way to construct fast adders. Hence, incorporating parallel prefix into modular addition can bring direct benefits to the performance of the modular adder. Many parallel prefix have been introduced under different scheme, whose performance heavily depends on the construction of carry network. Examples of some Parallel Prefix Adders (PPA) include Kogge-Stone, Brent-Kung, Han-Carlson, and Ladner-Fischer [4, 1, 6].

Amongst these existing PPAs, Ladner-Fischer PPA (LFPPA) is considered to provide the best tradeoffs and has been utilized as a a basic element in the most effective state of the art modular adder, ELM Modular Adder (ELMMA) [11]. ELMMA is based on the principle of anticipated computation, i.e., it computes both $a+b$ and $a+b-m$ in parallel and inherits the LFPPA capability to perform fast additions. Moreover, depending on the selection of the modulo value m the structure of the design various tradeoffs are possible in order to decrease the energy consumption and the occupied area. Generally speaking ELMMA consists of 3 modules: the Simplified Carry Save Adder (SCSA) module, the ELM module, and the MUX module. The SCSA module functions to perform 3-to-2

operand reduction, in order to prepare the input fitted to ELM module. In the ELM module, the two tentative modular additions takes place. Finally, the correct result is obtained based on the carry out of ELM. As reported in [11], ELMMA is used for unrestricted modulo addition. When modulo restriction is applied upon ELMMA, the use of SCSA can be avoided, resulting some extent of area saving, speed and power improvement.

The main research question we address in this thesis are the following:

- Can we propose an alternative technique for modular addition which can outperform ELMMA?
- In case we can, which is the actual impact of our proposal on the modular adder performance in terms of area, delay, and energy consumption?

1.2 Contribution

We addressed the previously mentioned questions and in this section we present the main contribution of the thesis, which consist of:

1. We proposed and enhanced ELM Modular Addition (ELMMA), which also utilize anticipated computation in conjunction with fast parallel prefix addition but reduces the overall complexity by proposing and alternative design for the adders. Our ELM adders can now directly handle three inputs instead of two. In this way the initial carry-save addition required for ELMMA for the evaluation of the $a+b-m$ is not longer required and this may potentially result in faster and more area and power effective designs.

2. We identified the moduli in the form of $(2^n - 2^{n-2})$, $(2^n - (2^{n-2}+1))$, and $(2^n - (2^{n-3}+1))$ as relevant candidates for the modulo adder implementations.

3. For the considered moduli we implemented in VHDL two sets of implementations, i.e., one for the state of the art ELMMA and one for our proposal, for the n=16 case. We simulated, debug, and synthesized the designs using Cadence Encounter RTL Compiler for ASIC Designs for 90 nm CMOS technology and our results indicate the following:

    - For the moduli $(2^n-(2^{n-2}+1))$, our scheme is 14% faster, requires 13% lesser area cost, and consumes 15% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 27%, 26%, 25%, 35%, respectively, when compared with the existing ELMMA.
    - For the moduli $(2^n - 2^{n-2})$, our scheme is 3% faster, requires 32% lesser area cost, and consumes 20% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 22%, 33%, 19%, 24%, respectively, when compared with the existing ELMMA.

- For the moduli $(2^n - (2^{n-3} + 1))$, our scheme is 9% faster, requires 28% lesser area cost, and consumes 13% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 22%, 22%, 24%, 27%, respectively, when compared with the existing ELMMA.

1.3 Thesis Outline

This remainder of the thesis is organized as follows:

Chapter 2 contains a brief overview of Residue Number System (RNS). Modular adders are also discussed by focussing on implementations for: (1) arbitrary moduli, and (2) restricted moduli. This chapter is closing with a discussion of several parallel adders based on the parallel prefix algorithm.

Chapter 3 provides the explanation of ELM algorithm. All signal equations involved are given, including the theory which support the corresponding equations. An enhancement of ELM is presented next by elaborating the extension of this technique in modular addition. Detailed explanations are given to describe how ELMMA computes the modulo addition on the based of ELM algorithm. Several properties such as resource sharing and the depth of the tree are also discussed, exposing the degree of improvement that ELMMA has if compared with the underlying and some relevant parallel prefix algorithm.

The enhanced ELMMA scheme is introduced in Chapter 4. The discussion starts with the additional and/or modified signal equation coming from the consequence of removing one ELMMA's block. Detailed supporting arguments for the modification are also presented, to give clearer description on how the improvement in proposed design is achieved. Results of synthesis of ELMMA and proposed design obtained from a synthesis tool are reported. A performance comparison between those design is provided, encompassing the improvement of delay, area, and total power. The comparison of compound metrics is also provided, to give an indication the superiority of the enhanced ELMMA over state of the art ELMMA in when comparison is performed to the combination of (delay, area), (delay, power), and (area, power).

Chapter 5 is the closing chapter of this report, containing conclusion and discussion of future research.

Overview of Residue Number Systems

2

This chapter provides fundamental information about the basic notions and several important properties of RNS which will give foundation to the succeeding chapters.

2.1 Weighted Number System

Numbers play an important role in computer systems and are the basis and object of computer operations. "The main task of computer is computing, which deals with numbers all the time" [7]. Digital computers very much depends on the utilized number systems and the rules which define the relationships among numbers. The commonly used number systems are decimal and binary number systems, which are examples of Weighted Number System (WNS). A number system is a WNS if for any number y in the system, y can be expressed as:

$$y = \sum_{i=1}^{n} x_i w_i,$$

where x_i is a digit in the permissible digits and w_i are consecutive powers of the same (fixed radix) or different (mixed radix) numbers. The following are the general characteristics of weighted number systems [12]:

- Relative magnitude comparison of 2 numbers can easily be carried out.
- Multiplication and division can be easily manually performed by moving the binary/decimal point in case of binary or decimal number system.
- The range of the number can easily be extended by the addition of more digit positions.
- Overflow detection is very easy.
- Analog to digital conversion is very easy.

Despite of all these advantages, WNS based arithmetic has limited performance due to the fact that it cannot support parallel arithmetic in which all the digits are simultaneously processed is not possible. This is due to the fact that in any WNS the addition requires the propagation of carry values between consecutive digit positions and this makes it impracticable to implement parallel addition, subtraction, multiplication, and division. This is a barrier on the speed of arithmetic operations in WNS and attempts have been made to overcome the speed limitations:

- Speedup the calculation of carries by adding specialized look-ahead carry circuitry.

- Rely on alternative number system representations which break/limit the carry propagation chains.

In this thesis, we focus on the later avenue and address, Residue Number System (RNS)[12] is such a number system with the following features: parallelism, modularity, fault tolerance, and carry free operations. In the next section, we present more details the general RNS background.

2.2 Residue Number System (RNS)

In RNS [9], a number x is represented by the list of its residues with respect to k pairwise relatively prime moduli ($m_{k-1} \geq ... \geq m_1 \geq m_0$). The residue x_i of x with respect to the ith modulus m_i is akin to a digit and the entire k-residue representation of x can be viewed as a k-digit number, where the digit set for the ith position is $[0, m_i - 1]$. Notationally, given any integer x, the residue x_i with respect to m_i is computed as:

$$x_i = x \; mod \; m_i \; = |x|_{m_i} \tag{2.1}$$

The RNS representation of x can be given by the enclosure of RNS digits in parenthesis. For example,

$$x = (2|3|2)_{RNS(7|5|3)} \tag{2.2}$$

is the RNS representation of 23 with respect to the moduli 7, 5, and 3.

The RNS dynamic range is represented by the product of all k relatively prime moduli ($M = m_0 \times m_1 ... \times m_{k-1}$). It denotes the interval over which every integer can be represented by the system without having two numbers with the same representation, viz. [0, M-1], or any other interval of M consecutive integers. For instance, when a negative number is desired with symmmetric compotition, the range can be set to [-(M-1)/2, (M-1)/2] if M is odd, or [-M/2, (M-1)/2] if M is even. The residues of negative numbers are evaluated by complementing each of the digits x_i with respect to its corresponding modulus m_i.

2.3 Modular Adders

Essentially speaking, modular adders are built using similar principles as the traditional binary adders. All improvement techniques found in binary addition can be utilized to construct modular adders [8]. Further improvements can be obtained from the fact that the modulus is known at design time. Modulo adders can be categorized into two groups, depending on the type of modulus they are designed for: (a) arbitrary (unrestricted) moduli and (b) restricted moduli.

2.3.1 Arbitrary Moduli

The result of adding, modulo-m, two numbers, A and B, (those can be digits of a residue representation but also two positive integers smaller than m), where $0 \leq A, B < m$, is defined as [8]:

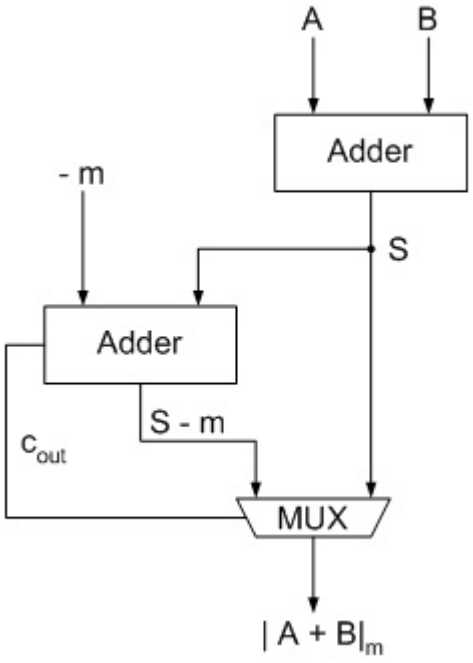

Figure 2.1: Arbitrary basic modulo-m adder

$$|A+B|_m = \begin{cases} A+B & \text{if } A+B < m \\ A+B-m & \text{otherwise} \end{cases} \tag{2.3}$$

The design of a modulo m adder may be based on Equation (2.3). The most straightforward implementation is realized by assigning one adder to compute the addition of A and B and a second adder to compute the subtraction of the modulus m from the previous addition result as depicted in Figure 2.1. We note that the result of the comparison of the addition result and the modulus is computed as c_{out} and this signal is utilized to select the right output value. If the addition of A and B is less than m, then this is the correct result. If otherwise, the addition of A and B is corrected by the subtraction of m to produce the correct result. We note that this computation can be also achieved with one adder only at the expense of a slightly increased delay and a register and two multiplexors.

By replacing the subtrahend m with an addition of additive inverse of m mod 2^n, $t(=2^n - m)$, Equation (2.3) can be rewritten as follows:

$$|A+B|_m = \begin{cases} A+B & \text{if } A+B < 2^n \\ A+B+t & \text{otherwise} \end{cases} \tag{2.4}$$

2.3.2 Restricted Moduli

Moduli in the form of $2^n - 1$, 2^n, and $2^n + 1$ are a special moduli because they greatly simplify the modular adder hardware implementation. Designing adders for moduli 2^n requires no extra work since the correct result is always produced for both cases in Equation (2.3) by the first adder in Figure 2.1. This holds true due to the fact that the substraction of 2^n is equivalent with dropping the carry out of the $A + B$ addition. Thus by just ignoring this carry the first adder always produces the correct result.. To design an adder with modulo $2^n - 1$ adder, the scheme of arbitrary adders can be used by replacing the complement m with constant 1. Essentially speaking the idea is to ignore the carry out of the $A + B$ addition but in this case an extra 1 has to be compensated somehow as by dropping the carry out we subtracted 2^n instead of $2^n - 1$. By assuming that the underlying adder is a carry ripple adder and it is reused as suggested at the end of the previous section, the correct addition is obtained by feeding back the carry generated by the addition of A and B. The implementation of $2^n + 1$ modulo adders also depends on the underlying conventional adder. In this case, to reduce the additional logic, arithmetic operation modulo $2^n + 1$ has frequently been implemented through the use of a different representation, e.g., diminished one representation. With the diminished-one representation, the addition of two numbers, A and B, with diminished-one equivalents $\hat{A}$ $(= A - 1)$ and $\hat{B}$ (=B-1), is $A + B = (\hat{A} + 1) + (\hat{B} + 1) = (\hat{A} + \hat{B} + 1) + 1$. If $\hat{A} + \hat{B} + 1 < 2^{n+1}$, then the result is correct in diminished-one form; if otherwise, $2^n + 1$ must be subtracted to get the correct result. The main drawback of the diminished-one representation is that it requires conversion of the operands and results.

2.4 Parallel Prefix Adders

Parallel-prefix adders allow more efficient implementations of the carry-lookahead technique and are, essentially, variants of carry-lookahead adders. Indeed, in current technology, parallel-prefix adders are among the best adders, with respect to $area \times time$, and are particularly suited for high-speed addition of large numbers. The construction of parallel prefix adders is based on carry operator, ¢, which is defined as follows:

$$(G", P") \, ¢(G', P') = (G" + P"G', P"P') \tag{2.5}$$

where P", P' and G", G' indicate propagate and generate signal, respectively. The fundamental of carry operator is represented in Figure 2.2.

A parallel prefix can be represented as a graph consisting of carry operator nodes. For the sake of supporting the explanation of ELM algorithm in Chapter 3, the Ladner Fischer Parallel Prefix Adder (LFPPA) [3] is presented because it has similar scheme to ELM. Figure 2.3 graphically represent the connection of carry operator node in LFPPA for the case of n = 8. All the signals produced by this LFPPA along the addition computation are given as follows:

Stage 0:

$p_{i,i} = a_i \oplus b_i, \qquad g_{i,i} = a_i b_i, \quad$ where $0 \leq i \leq 7$.

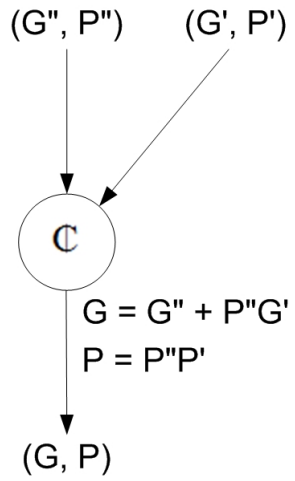

Figure 2.2: Carry Operator

Stage 1:

$g_{i,i-1} = g_i + p_i g_{i-1}$, where $i = 1, 3, 5, 7$.
$p_{i,i-1} = p_i p_{i-1}$, where $i = 3, 5, 7$.

Stage 2:

$g_{2,0} = p_2 g_{1,0}$ $\quad g_{3,0} = g_{3,2} + p_{3,2} g_{1,0}$
$g_{6,4} = p_6 g_{5,4}$ $\quad g_{7,4} = g_{7,6} + p_{7,6} g_{5,4}$
$p_{6,4} = p_6 p_{5,4}$ $\quad p_{7,4} = p_{7,4} p_{5,4}$

Stage 3:

$g_{4,0} = g_4 + p_4 g_{3,0}$ $\quad g_{5,0} = g_{5,4} + p_{5,4} g_{3,0}$
$g_{6,0} = g_{6,4} + p_{6,4} g_{3,0}$ $\quad g_{7,0} = g_{7,4} + p_{7,4} g_{3,0}$

In the particular case of modular addition, parallel prefix can be easiliy modified for addition with respect to the special moduli $2^n - 1$ and $2^n + 1$ [8]. Furthermore, if modulo addition is constructed for arbitrary moduli, the concurrent computation of $A + B$ and $A + B + t$ (t = correction term) can be fully supported by parallel prefix as suggested in [8]. Worth to mention that for a certain type of parallel prefix algorithms, there is a possibility to let those concurrent additions performed by shared resources as suggested in [11]. Note that this also leads to the reduction of power consumption as the parallel prefix adder occupies less area while preserving the computation speed.

2.5 Conclusion

In this chapter we briefly discussed how Residue Number Systems can be utilized in order to alleviate the speed limitation due to carry propagation. By selecting relatively smaller digits to represent large number, RNS can perform fast computation. Furthermore,

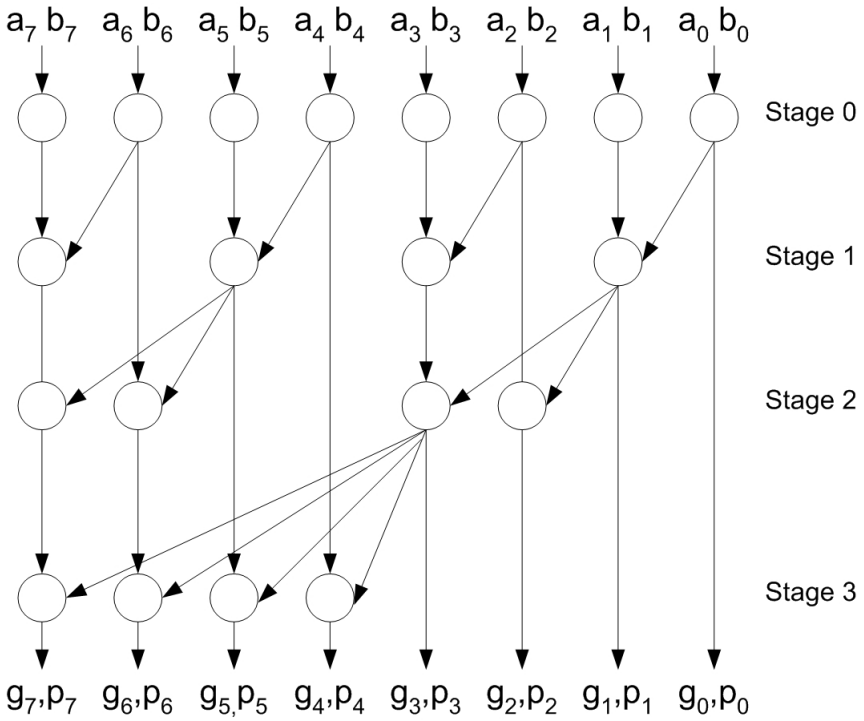

Figure 2.3: Ladner Fischer Adder

since no carry propagation produced in each RNS digit when operating arithmetic (e.g., addition), each digit can be processed independently and results are not affected by other digit positions if error occurs. We also analyzed the modulo addition operation and its implementation via especially in addition, well-know conventional binary addition methods and algorithms. We gave special attention to the utilization of parallel prefix algorithms to perform modular addition for arbitrary and restricted modulo adder, as depending on the characteristic of carry operator network that the parallel prefix has, this approach can result in a fast, small, and power-efficient modular adders.

ELM Modular Adder (ELMMA)

3

In this chapter, detailed information about the ELM algorithm [13] and the construction of ELMMA based on ELM tree [11] are given, including the derivation of equations, the equations to compute propagate, generate, and partial sum signals, and the components utilized in the construction of the modular adder trees.

3.1 ELM

The ELM algorithm utilizes a binary tree of simple processors to perform standard binary addition [13]. The structure of the tree is depicted in Figure 3.1 for the case 8-bit operands. Inputs are received at the lowest leaves of the tree, which compute the partial sums for each bit position. At this level, the partial sums are equal to the propagate signals. These partial sums are then passed up to the next level of the tree. Each leaf also computes a generate signal and passes it up to the next level. At higher levels of the tree, nodes receive partial sums from the adjacent lower level, as well as information necessary to update the partial sums at that level of the tree. New partial sums and update information are computed, which are passed up to the next higher level of the tree.

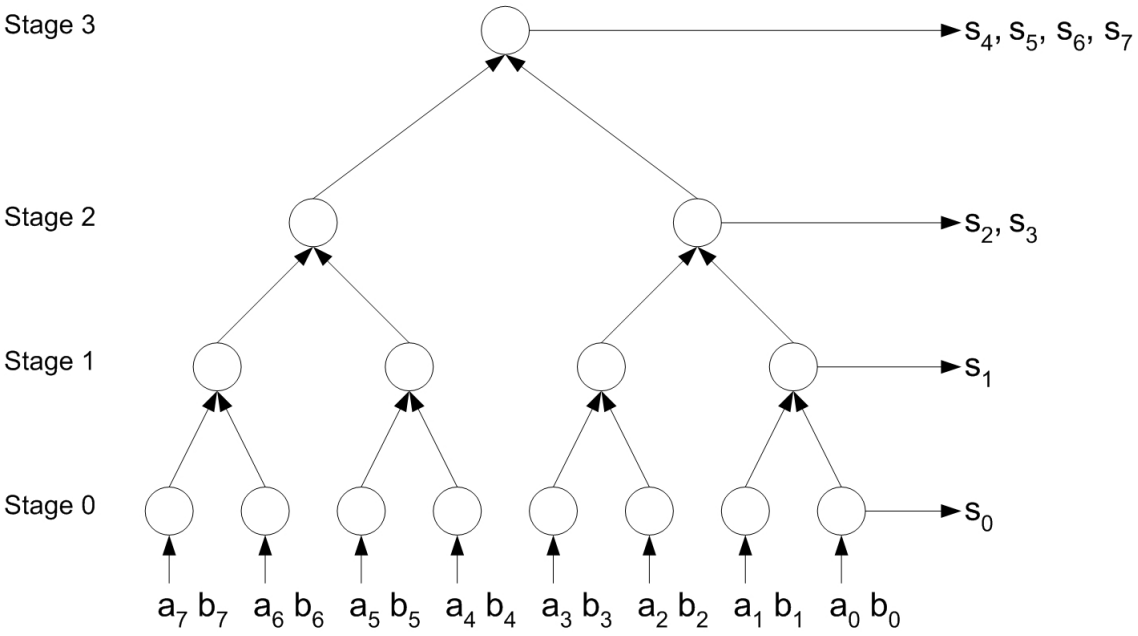

Figure 3.1: ELM block diagram for n = 8

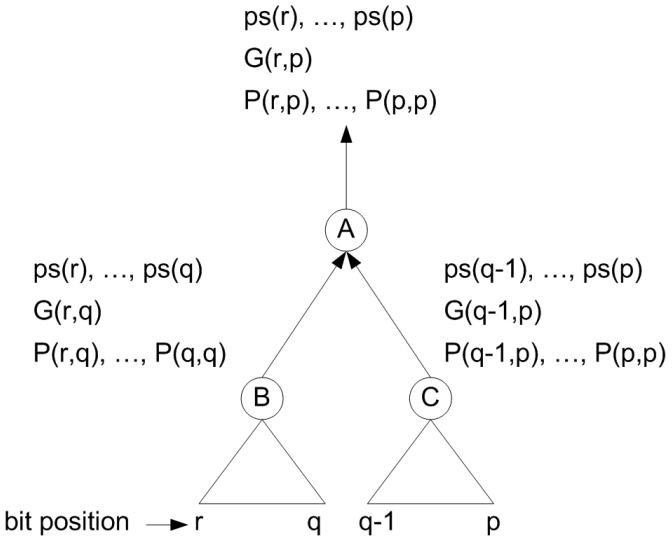

Figure 3.2: Internal Node ELM Calculations

The key to the algorithm is the computation which occurs at internal nodes, which is graphically described in Figure 3.2. The edge from node C to node A passes up the partial sums for its subtree, the generalized generate for its entire subtree and the generalized propagates for each bit of its subtree. Node B passes similar information, for its subtree, to node A. Computations at node A combine the information passed from node B and C to produce ps_i, $G(r,p)$, and $P(i,p)$ for its subtree. Note that node A requires does not have to do any processing to produce ps_p through ps_{q-1}, nor $P(p,p)$ through $P(q-1,p)$ and these signals may be passed up to the node at the next higher level directly. The remaining bits are computed according with Equation (3.1), (3.2), (3.3).

$$G_A(r,p) = G_B(r,q) + P_B(r,q)G_C(q-1,p) \tag{3.1}$$

$$P_A(q+j,p) = P_B(q+j,q)P_C(q-1,p) \tag{3.2}$$

$$ps^A_{q+j} = ps^B_{q+j} \oplus P({}^Bq+j-1,q)G^C(q-1,p) \tag{3.3}$$

The rightmost node of each level is required to neither pass up propagate signal nor partial sums. The propagate signals present in these nodes are not used any longer in the higher levels of the tree. In fact, its partial sums are the actual sum bits, because the subtree includes the least significant bits of the addition. This implies that the carry coming from the rightmost node is the actual carry-in to bit i and not merely a partial carry. The complete design for the considered 8-bit adder can be implemented as follows:

Stage 0:

${}_0ps_i = p_{i,i} = a_i \oplus b_i, \qquad g_{i,i} = a_i b_i, \quad$ where $0 \leq i \leq 7$.

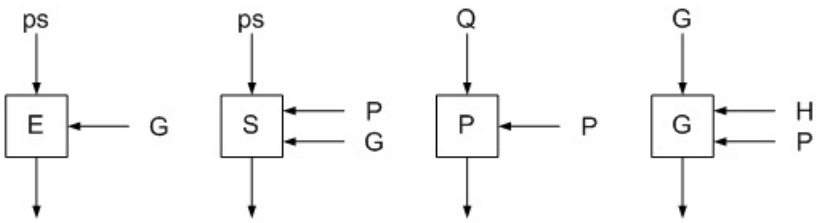

Figure 3.3: ELM Adder Cell Inputs

Stage 1:

$_1ps_i = p_{i,i} \oplus g_{i-1,i-1}$, where $i = 1, 3, 5, 7$.
$g_{i,i-1} = g_i + p_i g_{i-1}$,
$p_{i,i-1} = p_i p_{i-1}$, where $i = 3, 5, 7$.

Stage 2:

$_2ps_2 =_0 ps_2 \oplus g_{1,0}$, $_2ps_3 =_1 ps_3 \oplus p_{2,2}g_{1,0}$, $g_{3,0} = g_{3,2} + p_{3,2}g_{1,0}$,
$_2ps_6 =_0 ps_6 \oplus g_{5,4}$, $_2ps_7 =_1 ps_7 \oplus p_{6,6}g_{5,4}$, $g_{7,4} = g_{7,6} + p_{7,6}g_{5,4}$,
$p_{6,4} = p_{6,6}p_{5,4}$ $p_{7,4} = p_{7,6}p_{5,4}$

Stage 3:

$_3ps_4 =_0 ps_4 \oplus g_{3,0}$, $_3ps_5 =_1 ps_5 \oplus p_{4,4}g_{3,0}$,
$_3ps_6 =_2 ps_6 \oplus p_{5,4}g_{3,0}$, $_3ps_7 =_2 ps_7 + p_{6,4}g_{3,0}$,
$c_{out} = g_{7,0} = g_{7,4} \oplus p_{7,4}g_{3,0}$

In view of the previous discution, the implementation of any ELM adder can be done by using the following cells: The E cell which performs the function $ps \oplus G$, the S cell which performs the function of $ps \oplus PG$, the P cell which computes PQ, and the G cell which computes $G + PH$. The E and S cell functions to update a partial sum, P updates the propagate signal block, $P(i, j)$, and G computes new generate signal block, $G(i, j)$. Figure 3.3 present previously mentioned the cells and their inputs.

3.2 ELM Modular Adder

By using Equation (2.4), modular addition can be implemented by performing concurrent addition of $x+y$ and $x+y+t$ and selecting the correct result based on the carry-out signal produced by the three-operand addition. Three blocks are required to compute RNS addition: (1) the Simplified Carry Save Adder (SCSA) module, (2) the ELMMA tree, and (3) the MUX module [11] (Fig. 3.4).

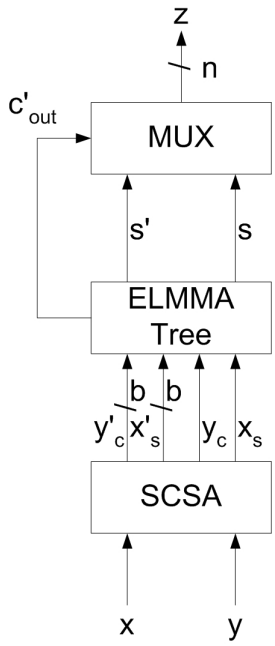

Figure 3.4: Block Representation of ELMMA Adder

3.2.1 The SCSA Module

The SCSA module is used to convert the three operand addition $x+y+t$ to two operands so that ELM tree architecture can be subsequently used to perform fast modular addition. The structure is a simplified version of the conventional CSA, where the extent of the simplification is highly dependent upon the binary realization of t. The simplification of CSA is given in the following description.

Let s_i and c_{i+1} respectively denote the sum and carry outputs form the i^{th} Full Adder (FA) in the CSA, then

$$s_i = x_i \oplus y_i \oplus t_i \tag{3.4}$$

$$c_{i+1} = x_i y_i \oplus x_i t_i \oplus y_i t_i \tag{3.5}$$

where x_i, y_i, and t_i represent the input bits to the FA and $\oplus$ represents the XOR operator. If the input bit t_i assumes the value 0, then writing s_i as $x_{s,i}$ and c_{i+1} as $y_{c,i+1}$, Equation (3.4) and (3.5) become:

$$x_{s,i} = x_i \oplus y_i \tag{3.6}$$

$$y_{c,i+1} = x_i y_i \tag{3.7}$$

Similiarly, if t_i is 1, then Equation (3.4) and (3.5) can be written as:

$$x'_{s,i} = \overline{x_i \oplus y_i} \tag{3.8}$$

$$y'_{c,i+1} = x_i + y_i \tag{3.9}$$

From Equation (3.6) and (3.7) it can be seen that for the case $t_i = 0$, the sum and carry bits for the sum $x + y + t$ are identical to the sum and carry bits one would obtain for the sum $x + y$. Note that for a modulus m, where the binary representation of t has b ones, the signal bus x'_s and y'_c contains b bits ($b \leq n-1, m \neq 2^n$), as depicted in Figure 3.4. A prime symbol is given to all signals involved in the computation of $x + y + t$, to easily distinguish them from the sum of two operand $(x + y)$. In addition, as it can be noticed that for n bit moduli, the Most Significant Bit (MSB) of t, i.e., t_{n-1}, is always zero, hence $y_{c,n}$ forms the partial output carry information for the addition $x + y$. This is because the subtraction with the additive inverse which has bit one at bit position $n - 1$ will decrease the number of utilized bit by one.

3.2.2 The ELMMA Tree

This module is the core for ELMMA. To describe the structure of the tree, an example is presented in Figure 3.5. The idea behind the structure is to compute both sums $x + y$ and $x + y + t$ in parallel and then using the carry resulting from the sum $x + y + t$ to determine the correct sum. The ELMMA tree thus consists of two overlapped ELM tree, where the extent of overlapping depends on the binary form of t. Let label the tree of $x + y$ as tree A and of $x + y + t$ as tree B. To give a clear separation between signal coming from tree A and B, the both trees are drawn using different line pattern (see Figure 3.5).

Tree A is completely identified by conjoining the thinly-dashed nodes/branches of the ELMMA tree with the solid-lined nodes/branches. Note that the solid-lined nodes and branches comprise of hardware shared between both addition tree. Hence, node 0, 1, 2, 3, 4, 5, and 6 form the ELM tree that computes the sum $x + y$. The processing done in the nodes at each level is identical to the computation of ELM's signals, with the exception that node 6 does not compute the carry-out. Tree B is constructed in a similar manner to the combination of the dashed nodes/branches and the shared nodes/branches. This means that this tree consists of nodes 0', 1', 2, 3, 4', 5, and 6'. The input bits to tree B differ from those in tree A at the same position due to applying different equation to compute generate and propagate/partial sum at the lowest level of tree.

The carry from the addition of $x + y + t$, c'_{out}, is then found by taking the logical OR of the carry produced from the SCSA module, $y_{c,5}$, and the generate signal covering bit position 1 to 4 which is obtained from the calculation of three operand addition. In general, for an n bit modulus, c'_{out} is produced from the logic OR operation between generate signal coming out from SCSA at bit position $n-1$ ($y_{c,n}$) and the carry generated from tree B in the ELMMA tree over bit position 1 to $n - 1$. Both of these signals are mutually exclusive for $m \neq 2^n$, where $n = \lceil \log_2 m \rceil$.

The width of the tree is equal to $n - 1$, resulting in the tree of $\lceil \log_2(n-1) \rceil$ depth. This means that even with the SCSA module, the depth of the modular adder struc-

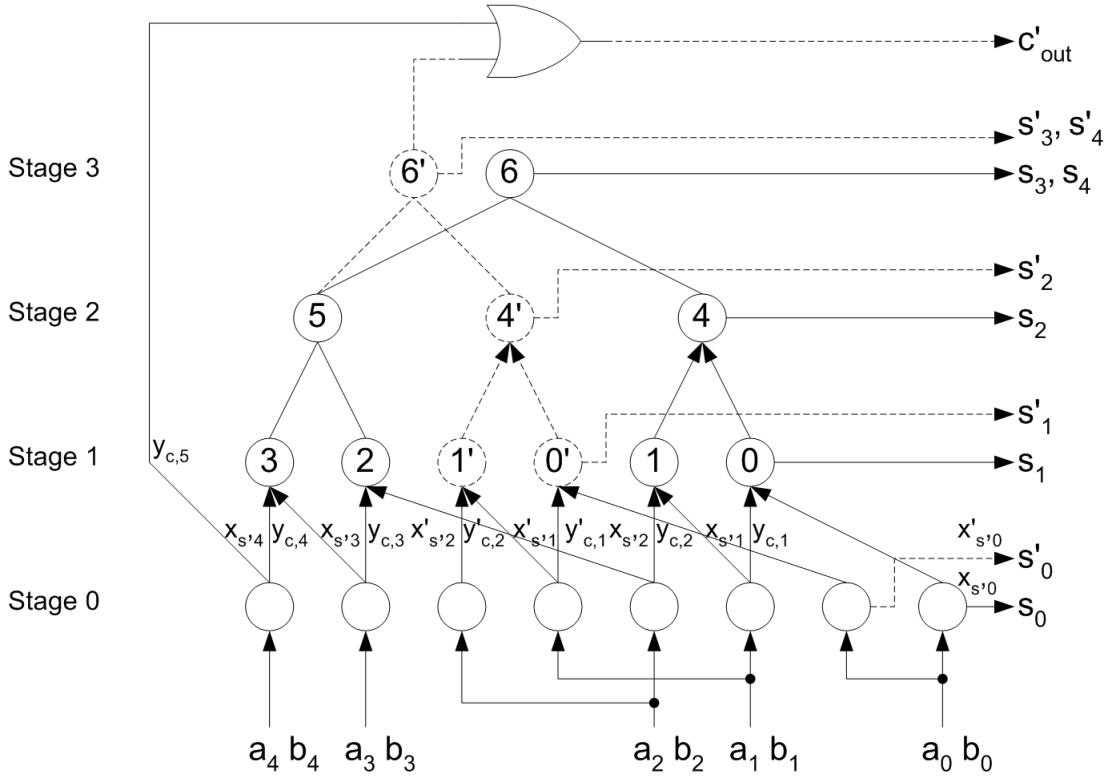

Figure 3.5: ELMMA Tree Block Diagram for m=29.

ture(excluding the MUX unit) is equal to that of an n bit ELM binary adder, a property that makes it a very fast modular adder implementations.

3.2.3 The MUX Module

This module consists of a number of 2-to-1 multiplexerss that are used to select the sum bits generated by the two additions performed in the ELMMA tree. The c'_{out} signal drives the multiplexer select port for all the multiplexers and it therefore has a maximum fan-out of n.

Connecting those three blocks as shown in Figure 3.4 results in the construction of ELMMA structure. To be able to present detailed ELMMA design, while keeping it comprehendable, we assumed in the following as an example of modular addition for the case of m = 29 (see Figure 3.5).

Stage 0 (SCSA):

$${}_0ps_i = p_{i,i} = a_i \oplus b_i, \qquad g_{i,i} = a_i b_i, \qquad \text{where } 0 \le i \le 4.$$

$${}_0ps'_0 = p'_{0,0} = \overline{a_0 \oplus b_0}, \quad g'_{0,0} = a_0 + b_0$$
$${}_0ps'_1 = p'_{1,1} = \overline{a_1 \oplus b_1}, \quad g'_{1,1} = a_1 + b_1$$

Stage 1:

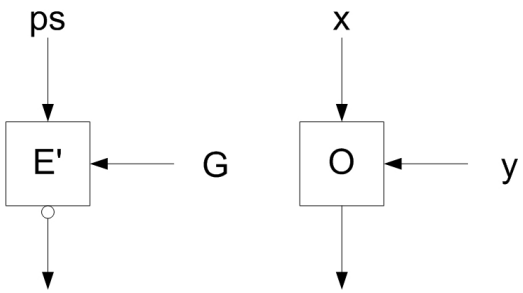

Figure 3.6: Cells used to compute the sum and carry when $t_i = 1$.

$${}_1ps_i = {}_0ps_i \oplus g_{i-1,i-1}, \quad g_{i,i} = p_{i,i}g_{i-1,i-1}, \quad \text{where } 1 \leq i \leq 4.$$

$${}_1ps'_1 = {}_0ps'_1 \oplus g'_{0,0}, \qquad g'_{1,1} = p'_{1,1}g'_{0,0}$$
$${}_1ps'_2 = {}_0ps_2 \oplus g'_{1,1}, \qquad g'_{2,2} = p_{2,2}g'_{1,1}$$

Stage 2:

$${}_2ps_2 = {}_1ps_2 \oplus g_{1,1}, \quad g_{2,1} = g_{2,2} + p_{2,2}g_{1,1},$$
$${}_2ps_4 = {}_1ps_4 \oplus g_{3,3}, \quad g_{4,3} = g_{4,4} + p_{4,4}g_{3,3}, \quad p_{4,3} = p_{4,4}p_{3,3}$$

$${}_2ps'_2 = {}_1ps_2 \oplus g'_{1,1}, \quad g'_{2,1} = g_{2,2} + p_{2,2}g'_{1,0},$$

Stage 3:

$${}_3ps_3 = {}_1ps_3 \oplus g_{2,1}, \qquad {}_3ps_4 = {}_2ps_4 \oplus p_{3,3}g_{2,1}$$

$${}_3ps'_3 = {}_1ps_3 \oplus g'_{2,1}, \qquad {}_3ps'_4 = {}_2ps_4 \oplus p_{3,3}g'_{2,1}$$
$$c'_{out} = g'_{4,1} = g_{4,3} + p_{4,3}g'_{2,1}$$

Referring to all the signals required to compute the two concurrent additions, ELMMA can be implemented by using all existing cells required by ELM and another additional cells, i.e., the E' and the O cell, to perform the 3-to-2 operand compression in the SCSA stage. Figure 3.6 presents the inputs to the additional cell required by ELMMA.

3.3 Conclusion

In this chapter we first discussed ELM which can be considered as an enhancement of Ladner Fischer Parallel Prefix Adder (LFPPA). The enhancement is obtained by incorporating partial sum calculation along the generation of intermediate signal. The modification applied in ELM leads to reduction of one computation stage when compared with LFPPA. Subsequently, we presented the ELM Modular Adder (ELMMA), a state of the art modular addition scheme which builds upon ELM. ELMMA is a computation anticipation based approach which is result in a fast adder with less power utilization due to the fact that resource sharing is enabled for the adders computing the two concurrent additions required for the modular addition are performed. In the next chapter we

introduce our proposal to further improve ELMMA performance measured in terms of area, delay an power consumption and present some experimental results.

Enhanced ELM Modular Addition

4

The basic idea behind our proposal is to eliminate the initial reduction stage required by the state of the art ELMMA approach. This may potentially result in area and delay savings under the conditions that this removal is not heavily affecting the complexity of the ELM trees. In the chapter we first introduce our basic scheme and than we present it in more details for some moduli of practical interest. Finally, we present three practical implementations and compare their performance with the one of equivalent ELMMA implementations.

4.1 Eliminating the Simplified Carry Save Adder

In the ELMMA algorithm described in the previous chapter, the Simplified Carry Save Adder (SCSA) is used to perform 3-to-2 operand compression in order to fit the sum of three operand as input for the ELM tree. As one might observe, the SCSA is employed to the addition of two operand to serve the purpose of taking the advantage of resource sharing when inputs originating from two and three operand addition enter the ELM module. Two cases can be distingushed here with respect the computation latency and signal complexity, based on the form of ELMMA's additive inverse, t. For the case of $t_i = 0$, the use of SCSA block can be avoided because the resutl of three operand addition is exactly the same to the two operand addition. This leads to area saving and latency reduction of one Full Adder (FA) when the SCSA module is removed from ELMMA scheme. On the other hand, the presence of the SCSA module in the case of $t_i = 1$ is certainly required since the signals produced from the operand reduction of two and three operand are different. Certain equations can be obviously developed to either link the intermediate signal of three operand addition with signals produced by two operands or compute the intermediate signal of three operand addition without utilizing the SCSA block. However, simple signal equation cannot be maintained for every case and excessive new equations leads to the performance drawback. Therefore, this thesis focuses the improvement only to the choice of restricted moduli, which is dominantly influenced to the bit one appearance in t.

As mentioned in previous chapter, the E' and O cells are used to compress the sum of three operand. In the case of SCSA block removal, the computation of three operand addition is performed in the ELMA tree, to replace signal computation performed in SCSA module. As shown in Table 4.1, the mutual exclusive relation between propagate and generate signals cannot be preserved anymore when the input $a = b = 1$. This condition leads to the generation of Ambiguous Carry Ripple (ACR) when the assoiated bit position receives carry-in generated from previous position. Detailed explanation to this will be presented in next subsection.

ACR occurs when a generate signal affects more than one digit. In Figure 4.1, it is

Table 4.1: Propagate and Generate Signal in the Absence of SCSA

a	b	p	g	p'	g'
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	1	1	1	1

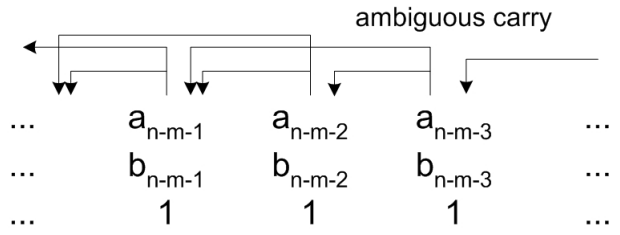

Figure 4.1: The Ripple of Ambiguous Carry

shown that when propagate and generate signals are equal to 1, the carry generated at certain digit position may influence one subsequent digit or the digit after the next one. The worse ACR even could take place if t contains a series of consecutive one. In this case, the ambiguous carry could be generated in every digit position, except at the Least Significat Bit (LSB). Consequently, the signal complexity increases significantly and no benefit could be obtained in this case. Nevertheless, an improvement obtained from the SCSA saving becomes obvious if the ambiguous carry can be minimized in order to keep the signal complexity low. In this thesis, we achieve this by applying certain restricted moduli to the ELMMA scheme.

4.2 Proposed Scheme For Certain Restricted Moduli

A significant reduction of ACR occurs if the additive inverse, t, is not allowed to have bit with value of 1 in consecutive positions and the position of that bit is closed to the Most Significant Bit (MSB). By assuming the carry-in of the modulo addition is equal to 0, the additive inverse with bit 1 discovered at the LSB will not have any impact to the emergence of the ambiguous carry signal. Applying this restriction of moduli selection, the additive inverse is in the form of 010...1, 010...0, and 001...1 for modulus in the form of $2^n - 2^{n-2}$, $2^n - (2^{n-2} + 1)$, and $2^n - (2^{n-3} + 1)$, respectively. Thus, the moduli of interest include: $2^n - 2^{n-2}$, $2^n - (2^{n-2} + 1)$, and $2^n - (2^{n-3} + 1)$. Although the selection of the moduli depends on the strategy to minimize the ACR, one modulus selection falls to the one which is reported in [10]. Furthermore, as one can observe in Table 4.2, modulus in the form of $2^n - (2^{n-2} + 1)$ is relatively prime to $2^n - (2^{n-3} + 1)$ for n within the span of $\{3, 16\}$. Another possibility of proposed moduli utilization is to incorporate it with the famous modulus in the form of 2^n in order to present a set containing three relatively pairwise moduli.

Table 4.2: Number Factorization for the Corresponding Modulo

n	$2^n-(2^{n-2}+1)$	Fact.	$2^n-(2^{n-2})$	Fact.	$2^n-(2^{n-3}+1)$	Fact.
3	5	5	6	2,3	6	2,3
4	11	11	12	2,3	13	13
5	23	23	24	2,3	27	3
6	47	47	48	2,3	55	5,11
7	95	5,19	96	2,3	111	3,37
8	191	191	192	2,3	223	223
9	383	383	384	2,3	447	3149
10	767	13,59	768	2,3	895	5,179
11	1535	5,307	1536	2,3	1791	3,199
12	3071	3071	3072	2,3	3583	3583
13	6143	6143	6144	2,3	7167	3,2389
14	12287	11,1117	12288	2,3	14335	5,47,61
15	24575	5,983	24576	2,3	28671	3,19,503
16	49151	23,2137	49152	2,3	57343	11,13,401

4.2.1 Moduli 2^n-2^{n-2} and $2^n-(2^{n-2}+1)$

For modulus 2^n-2^{n-2}, bit 1 is discovered at position $n-2$ while for modulus $2^n-(2^{n-2}+1)$, bit 1 is discovered at positions n-2 and 1. For both moduli, resource sharing will be obtained for signal computation from bit position $n-3$ down to 2 in the best case. Since the computation signal of modulus $2^n-(2^{n-2}+1)$ is a superset of modulus 2^n-2^{n-2}, thus a complete signal generation is presented only for the modulus $2^n-(2^{n-2}+1)$.

Although SCSA is no longer used in this scheme, almost all signal equations remain intact when compared with ELMMA's equations. The absence of SCSA only affects the signal equation for position $n-2$ and greater. To clearly explain the developed signal modification, we utilized $n=16$. Fig. 4.2 delineates all signal generation for modulo 49151. As shown in the picture, bold-printed signals are associated with the computation of three operand addition, whereas the remaining signals used in the tree of three operand addition come from the signal network of two operand addition.

To retain the modulo addition produced correctly, new signal equation is introduced for propagate signal of bit 15 to 14. One can observe that partial sum, propagate, and generate signal at bit position 14 and the block containing bit position 14 remain the same, thus it does not require any modification, since the mutual exclusive within bit position 13,0 never violate. The formulation of mutually exclusive relation of the p and g is expressed as follows:

$$p_i + g_i = p_i \oplus p_i, \tag{4.1}$$

where $0 \leq i \leq 13$. Equation (4.1) ensures that all generate signals can be transmitted properly to each bit position in the range of {14,0} during the construction of carry network in ELM tree. Hence, new signal formulation will only take place to the case of updating the partial sum at bit position 15 and propagating the generate signal over block {15,14}. Specifically to the case of $n=2^m$, no additional equation is required to

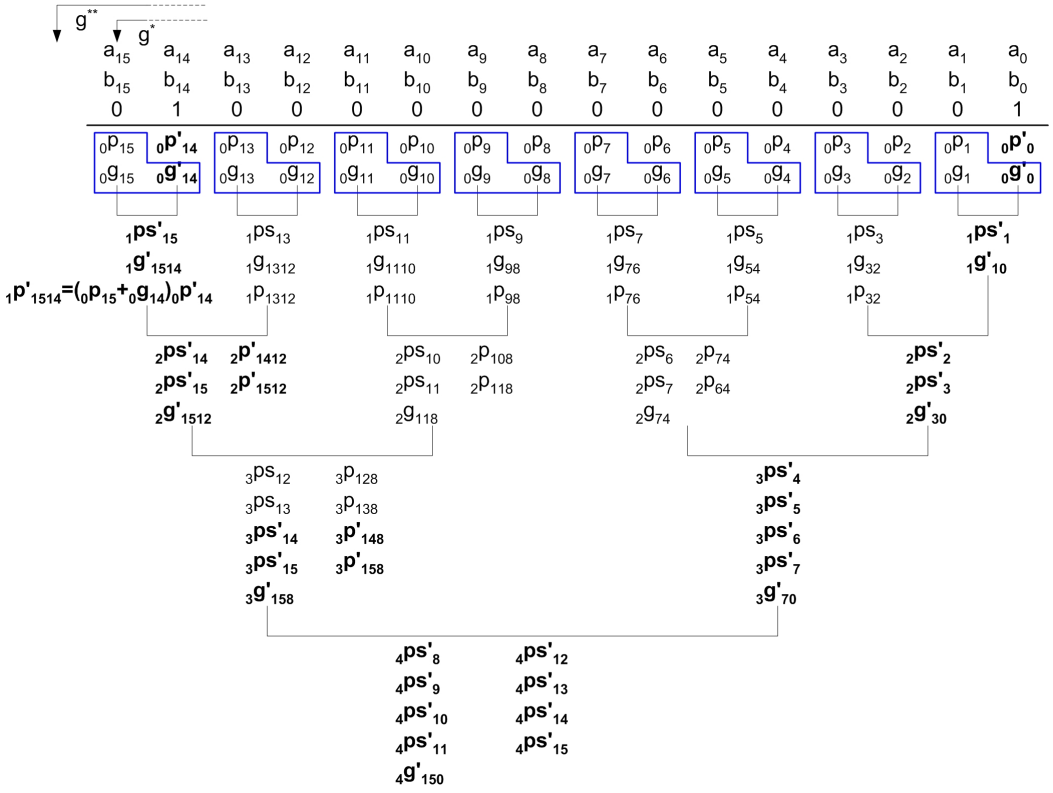

Figure 4.2: Proposed Signal generation for modulo 49151

be derived, since the partial signal of bit position 15 always receives a correct generate signal coming out from bit position 14 at the Stage 1. For general case, partial sum of bit position 15 has to be updated first to guarantee that the accumulation of parallel prefix is performed correctly. Thus, the equation which has to applied in general case is expressed as:

$$ps_{n-1} = p_{n-1} \oplus g_{n-1} \tag{4.2}$$

The modification of new propagate signal for block of bit 15 and 14 is denoted as:

$$_1p'_{1514} = (_0p'_{15} + \ _0g_{14})_0p'_{14} \tag{4.3}$$

Equation (4.3) implies that propagate signal at bit position 15 does not solely depends on its own input, but it is also influenced by the coming carry-in generated at bit position 14 of two operand addition. The modification aims to allow bit position 14 propagates generate signal over bit 15 when it receives carry-in when it also enables to propagate carry-in and generates carry-out at the same time ($_0p'_{14} = \ _0g'_{14} = 1$). Another signal which needs an attention is the partial sum at bit position 15 in stage 0 ($_0ps'_{15}$). In this stage, it is assumed that partial sum of bit 15 always receives carry-in produced by bit position 14. Then the correction is made when bit position 14 propagates carry-in. The latter condition holds true since bit position 15 (or each bit position, in general

case) receives carry-in with the value of 1 only once because of the mutual exclusive relation between propagate and generate signal (when a position generates carry-out, the propagate signal at the bit position or the block of propagate signal containing the bit position is always zero because of the mutual exclusive relation between propagate and generate signal in the case of the sum of two operand).

4.2.2 Modulus $2^n - (2^{n-3} + 1)$

The proposed ELMMA for modulus $2^n - (2^{n-3} + 1)$ is derived by applying local 3-to-2 operand reduction. Consequently, the delay of the proposed ELMMA increases. From Fig. 4.3, it can be observed that more intermediate signals must be computed at Stage 0 and Stage 1, resulting in higher area cost when compared with existing ELMMA at this point. The signal equations are derived by applying the following:

1. At Stage 0, all propagate and generate signals are computed. No signal equation modification is found here.

2. At Stage 1, partial sum of position 14 is updated to let it store a correct value after it receives carry-in coming from previous block. The propagate signal is denoted as: ${}_1ps'_{14} = {}_0p_{14} \oplus {}_0g'_{13}$. A new signal definition, ${}_1h'_{1514}$, is introduced, aiming to hold generate signal affecting bit position 15. The ${}_1h'_{1514}$ is denoted as the product of ${}_0h_{14}$ and ${}_0g'_{13}$.

3. At Stage 2, partial sum at bit position 14 is updated when generate signal is produced from bit position 13 and 12 (${}_2ps_{14} = {}_1ps'_{14} \oplus {}_1g'_{1312}$). Because a carry save addition is performed to bit position 14 and 15 at Stage 1, the calculation of propagate, generate, and partial sum signal which contains value of bit position 15 and 14 are defined as:
$$
\begin{aligned}
{}_2p'_{1412} &= {}_1ps'_{14} \oplus {}_1ps'_{14}\ {}_1g'_{1312} \oplus {}_1h_{1514} \\
{}_2g_{1512} &= {}_1g_{1514} + {}_1ps_{15}\ {}_1h'_{1514} \\
{}_2h'_{1514} &= {}_1ps_{15}\ {}_1ps'_{14}\ {}_1g'_{1312} \\
{}_2p'_{1412} &= {}_1ps'_{14}\ {}_1p'_{1312}\ {}_1g'_{1312} \\
{}_2p'_{1512} &= {}_1ps_{15}\ {}_1ps_{14}\ {}_1g'_{1312}
\end{aligned}
$$

4. In Stage 3 and 4, all signals are computed with the same equations as we have in the existing ELMMA algorithm. The carry-out of proposed ELMMA is obtained as the OR result of ${}_2h'_{1514}$ and ${}_4g'_{150}$.

4.3 Experimental Results

Two sets of designs, state of the art and enhanced ELMMA, have been developed in VHDL for three cases: $2^n - (2^{n-2} + 1)$, $2^n - 2^{n-2}$, and $2^n - (2^{n-3} + 1)$ with n = 16. After careful simulations and debug we synthesized all the designs, using the Cadence Encounter RTL Compiler for ASIC Designs at 90 nm CMOS technology. The performance of all designs measured in terms of area, delay and power has been estimated.

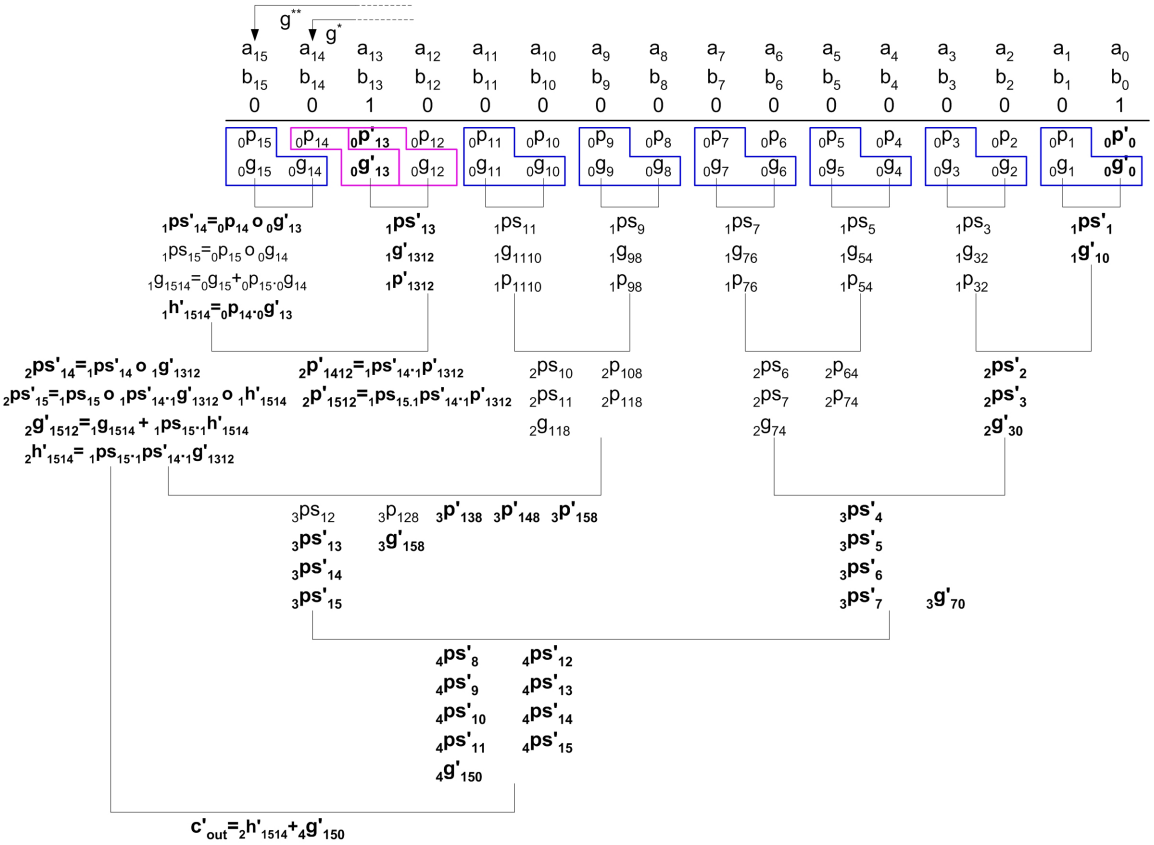

Figure 4.3: Signal Generation on the Scheme of Proposed Design for m = 57343

Table 4.3 summarizes the synthesis results for the moduli $2^n-(2^{n-2}+1)$, 2^n-2^{n-2}, and $2^n-(2^{n-3}+1)$ for the case of n = 16, , thus modulo 49151, 49152, and 57343, respectively. Our results indicate that the SCSA removal applied in the proposed ELMMA improves the speed of computation while occupying smaller area and consuming less power. Based on the simple metrics with rescpect to Delay(D), Area(A), and Power(P), the enhanced ELMMA for modulo 49151 has relatively balance improvement in its three aspects, spanning between 13%-15% improvement. In the remaining moduli forms, the chart indicates the signifiicant improvement in area saving, followed by the power efficiency. This holds true, since the target of improvement is to remove the utilization of SCSA, which significantly affect the total area required for the enhanced ELMMA. To further asses the implication of our proposal, we also compared the two type of designs in terms of compound metrics as Power x Delay (PD), Area x Delay (AD), Area x Power (AP), and Area x Delay^2 (DDA). The improvement in percentage of the enhanced ELMMA over the state of the art one is presented in Figure 4.5. The chart indicates that the savings of each metric is larger than 20%, with the average savings of 23%, 25%, and 26% in DP, AP, and DDA, respectively. The detail synthesis result for all three cases is provided as appendix.

Table 4.3: Synthesis Result for n=16

m	Delay (ps)		Area (Cell Area)		Power(μW)	
	Proposed	ELMMA	Proposed	ELMMA	Proposed	ELMMA
49151	1050	1227	857	986	50061	59170
49152	913	943	640	767	34239	42937
57343	1132	1256	895	1001	51485	59695

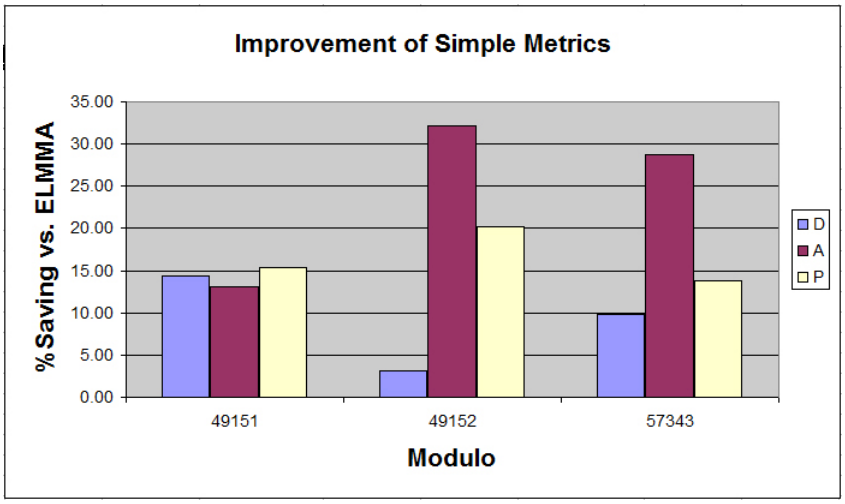

Figure 4.4: Enhanced vs Standard ELMMA Improvement in Terms of Simple Metrics.

4.4 Conclusion

Moduli restriction is applied on the state of the art modulo adder, ELMMA, resulting to the enhanced ELMMA design. The strategy of selecting the moduli is presented,

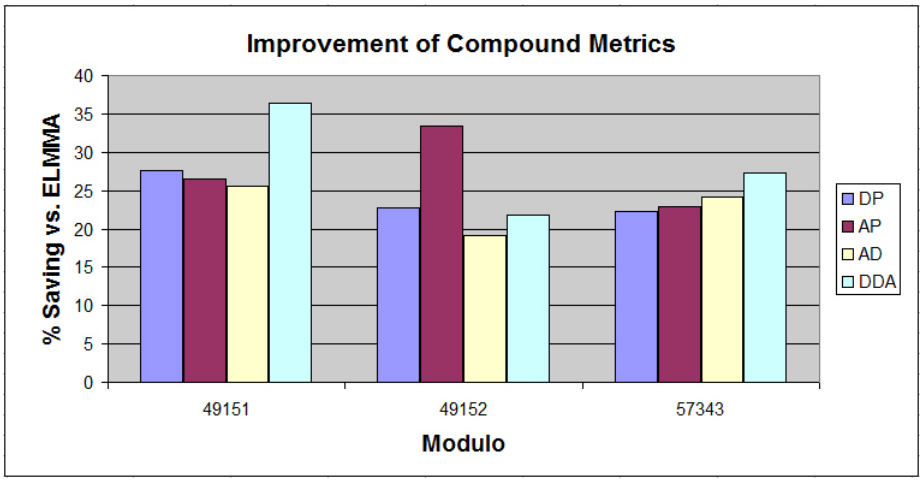

Figure 4.5: Enhanced vs Standard ELMMA Improvement in Terms of Compound Metrics.

coming up with the moduli selection in the form of $(2^n - (2^{n-2} + 1))$, $(2^n - 2^{n-2})$, and $(2^n - (2^{n-3} + 1))$. The simplest modification is found in the case of $(2^n - (2^{n-2} + 1))$ and $(2^n - 2^{n-2})$, where one modified equation is found at the bit position 15. Modulus in the form of $2^n - (2^{n-3} + 1)$ have more complicated signals due to the need to pass on the information of ambiguous carry ripple which could occur from position 2^{n-3}. For the case of modulus 49151, the proposed ELMMA requires 13% smaller area, 14% faster, and 15% more power efficient when compared with existing ELMMA design. In other modulus case, proposed ELMMA is more cost efficient especially in power and area, while the improvement of the speed is slightly better than existing one. From the compound metrics result, the enhanced ELMMA outperform the existing one with 23%, 25%, 25%, and 26% of average of DP, AD, AP, and DDA saving.

5 Conclusions

5.1 Summary

Chapter 2 discussed fundamental information about the basic notions and several important properties of RNS which gives the base for the succeeding chapters. First, the definition of Weighted Number System (WNS) and general characteristic is presented. Then the explanation of WNS limitation is described, focusing to the speed limitation due carry propagation. Next, RNS is introduced as one of the answer to eliminate carry propagation and thus speed up the computation. A brief explanation of RNS advantages and disadvantages are presented, leading to the conclusion that the use of RNS in special purpose application and the need to make modular based operation more efficient when operating in large digit number. The chapter was closed with the introduction of parallel prefix addition as the underlying algorithm applied in state of the art modular adder, ELMMA.

Chapter 3 presented an overview of ELM, a parallel prefix which is claimed to have higher performance than other parallel prefix adder. Signal generation involved in the ELM computation is also provided, which requires one less stage to complete computation when compared with the Ladner Fischer adder. Subsequently, we presented the ELM Modular Adder (ELMMA), a state of the art modular addition scheme which builds upon ELM. ELMMA is a computation anticipation based approach which is result in a fast adder with less power utilization due to the fact that resource sharing is enabled for the adders computing the two concurrent additions required for the modular addition are performed.

A new enhanced ELM modular adder was presented in Chapter 4. The discussion opened with the idea behind the enhanced ELMMA, that is to eliminate the simplified carry save adder module. An improvement can be obtained when moduli restriction is applied to ELMMA, to minimize the generation of ambiguous carry ripple. Applying the strategy to select the moduli, the performance of enhanced ELMMA adder in the form of moduli 2^n-2^{n-2}, $2^n-(2^{n-2}+1)$, and $2^n-(2^{n-3}+1)$ are investigated. As indicated by the synthesis tool, the largest on average improvement of enhanced ELMMA can be seen on the area saving, followed by power consumption, and then delay. For the case of modulus 49151, the proposed ELMMA requires 13% smaller area, 14% faster, and 15% more power efficient when compared with existing ELMMA design. In other modulus case, proposed ELMMA is more cost efficient especially in power and area, while the improvement of the speed is slightly better than existing one. The comparison of compound metrics is also provided, to give an indication the superiority of the enhanced ELMMA over state of the art ELMMA in when comparison is performed to the combination of (delay, area), (delay, power), and (area, power). From the compound metrics result, the enhanced ELMMA outperform the existing one with 23%, 25%, 25%, and 26% of average DP, AD,

AP, and DDA saving.

5.2 Major Contributions

In this thesis, we presented a number of novel modulo adders. Our overall achievements can be summarized by the following:

- We identify the moduli in the form of $(2^n - 2^{n-2})$, $(2^n - (2^{n-2} + 1))$, and $(2^n - (2^{n-3} + 1))$ as relevant candidates for the modulo adder implementations. The moduli selection is based on the scenario of minimizing ambiguous carry ripple and the the utilization of the moduli in practice.
- We estimate the performance of state of the art and the enhanced ELMMA. We simulated, debug, and synthesized the designs using Cadence Encounter RTL Compiler for ASIC Designs for 90 nm CMOS technology and the results indicate
  - For the moduli $(2^n-(2^{n-2}+1))$, our scheme is 14% faster, 13% lesser area cost, and consumes 15% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 27%, 26%, 25%, 35%, respectively, when compared to the existing ELMMA.
  - For the moduli $(2^n - 2^{n-2})$, our scheme is 3% faster, 32% lesser area cost, and consumes 20% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 22%, 33%, 19%, 24%, respectively, when compared to the existing ELMMA.
  - For the moduli $(2^n-(2^{n-3}+1))$, our scheme is 9% faster, 28% lesser area cost, and consumes 13% lesser power when compared with the ELMMA algorithm. With respect to compound metrics, the enhanced ELMMA improves the DP, AP, AD, and DDA about 22%, 22%, 24%, 27%, respectively, when compared to the existing ELMMA.

5.3 Future Directions

In this section, we present some future directions that could further improve RNS arithmetic.

- Given that for the moduli $2^n - (2^{n-2} + 1)$, $2^n - 2^{n-2}$, and $2^n - (2^{n-3} + 1)$, efficient modulo adders have been presented, it will be interesting to find out if efficient modulo multipliers can be designed using these moduli.
- Moduli $2^n - 2^{n-2}$ and $2^n - (2^{n-2} - 1)$ can be denoted as 3.2^n and $3.2^n + 1$, respectively. The new moduli forms imply that the modulo addition of $2^n - (2^{n-2} - 1)$ can be obtained from the modulo addition of $2^n - 2^{n-2}$ by restricting zero to be the valid input for the addtion, in order to fully map all number representation of modulo $2^n - (2^{n-2} - 1)$ to modulo $2^n - 2^{n-2}$ [2]. Diminished one representation

can be used to support the restriction and new ELMMA design can be developed based on the representation of diminished one representation. The same approach can be applied to modulo $(2^n - (2^{n-3} + 1))$.

Bibliography

[1] R. P. Brent and H. T. Kung, *A regular layout for parallel adders*, IEEE Transactions on Computers (1982).

[2] Costas Efstathiou Haridimos T. Vergos and Dimitris Nikolos, *Diminished-One Modulo* 2^n+1 *Adder Design*, Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on (2002).

[3] S. Knowles, *A family of adders*, IEEE Transactions on Computers (2002).

[4] P. Kogge and H. Stone, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, IEEE Transaction on Computers (1973).

[5] Israel Koren, *Computer arithmetic algorithm*, 2nd ed., A K Peters, Ltd., University of Massachussets, Amherst, 2002.

[6] R. E. Ladner and M. J. Fischer, *Parallel prefix computation*, Journal of the ACM (1980).

[7] Mi Lu, *Arithmetic and logic in computer systems*, vol. 1, John Wiley & Sons, Inc., 2004.

[8] Amos Omondi and Benyamin Premkumar, *Residue number systems, theory and implementation*, Imperial College Press, Yonsei University, South Korea and Nanyang Technological University, Singapore, 2007.

[9] Behrooz Parhami, *Computer arithmetic: Algorithms and hardware designs*, Oxford University Press, Department of Electrical and Computer Engineering, 2002.

[10] M. Benaissa R. A. Patel and S. Boussakta, *Fast Modulo* $2^n - (2^{n-2}+1)$ *Addition: A New Class of Adder for RNS*, IEEE (2006).

[11] N. Powell R. A. Patel, M. Benaissa and S. Boussakta, *ELMMA A New Low Power High-Speed Adder for RNS*, IEEE (2004).

[12] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*, McGraw-Hill, New York, 1967.

[13] R. M. Owens T. P. Kelliher and T.-T. Hwang, *ELM - A Fast Addition Algorithm Discovered by a Program*, IEEE Transactions on Computers (1992).

Synthesis Result for Modulo 49151

A

Table A.1: Area of the existing ELMMA for modulo 49151

Instance	Cells	Cell Area
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13p	2	8
l4ps13	2	8
l4ps12p	2	8
l4ps12	2	8
l4ps11p	2	8
l4ps11	2	8
l4ps10p	2	8
l4ps10	2	8
l3ps8p	2	8
l3ps8	2	8
l3ps7p	2	8
l3ps7	2	8
l3ps6p	2	8
l3ps6	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l2ps8	2	8
l2ps4p	2	8
l2ps4	2	8
l2ps12	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
p_g_n0_to_n15_level0_gen[9].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[8].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[7].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[6].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[5].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[4].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[2].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[1].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[15].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[14].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[13].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[12].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[11].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[10].p_n1_to_n15_level0	1	7
l4ps9p	1	7
l4ps9	1	7
l3ps5p	1	7
l3ps5	1	7
l3ps13	1	7
l2ps7	1	7
l2ps3p	1	7
l2ps3	1	7
l2ps15p	1	7
l2ps15	1	7
l2ps11	1	7
l1ps8	1	7

Continued on Next Page...

Table A.1 – Continued

Instance	Cells	Cell Area
l1ps6	1	7
l1ps4	1	7
l1ps2p	1	7
l1ps2	1	7
l1ps14p	1	7
l1ps14	1	7
l1ps12	1	7
l1ps10	1	7
l0p1p	1	7
l0p15p	1	7
l0p14p	1	7
MUX_bit9	1	6
MUX_bit8	1	6
MUX_bit7	1	6
MUX_bit6	1	6
MUX_bit5	1	6
MUX_bit4	1	6
MUX_bit3	1	6
MUX_bit2	1	6
MUX_bit15	1	6
MUX_bit14	1	6
MUX_bit13	1	6
MUX_bit12	1	6
MUX_bit11	1	6
MUX_bit10	1	6
MUX_bit1	1	6
MUX_bit0	1	6
xsp_bit14	1	6
xsp_bit0	1	6
l4g151p	1	5
l3g81p	1	5
l3g81	1	5
l3g159p	1	5
l2g85	1	5
l2g41p	1	5
l2g41	1	5
l2g1513p	1	5
l2g129	1	5
l1g87	1	5
l1g65	1	5
l1g43	1	5
l1g21p	1	5
l1g21	1	5
l1g1413p	1	5
l1g1413	1	5
l1g1211	1	5
l1g109	1	5
ycp_bit15	1	4
ycp_bit1	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
p_g_n0_to_n15_level0_gen[9].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[8].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[7].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[6].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[5].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[4].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[3].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[2].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[1].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[14].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[13].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[12].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[11].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[10].g_n1_to_n15_level0	1	4
l3p159p	1	4
l3p149p	1	4
l3p149	1	4
l3p139	1	4
l2p85	1	4

Continued on Next Page...

Table A.1 – Continued

Instance	Cells	Cell Area
l2p75	1	4
l2p1513p	1	4
l2p129	1	4
l2p119	1	4
l1p87	1	4
l1p65	1	4
l1p43	1	4
l1p1413p	1	4
l1p1413	1	4
l1p1211	1	4
l1p109	1	4
l0g1p	1	4
l0g15p	1	4
l0g14p	1	4
cout_prime	1	4
Total	**194**	**986**

Table A.2: Area of the proposed ELMMA for modulo 49151

Instance	Cells	Cell Area
l4ps9p	2	8
l4ps9	2	8
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13p	2	8
l4ps13	2	8
l4ps12p	2	8
l4ps12	2	8
l4ps11p	2	8
l4ps11	2	8
l4ps10p	2	8
l4ps10	2	8
l3ps7p	2	8
l3ps7	2	8
l3ps6p	2	8
l3ps6	2	8
l3ps5p	2	8
l3ps5	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l3ps13	2	8
l2ps7	2	8
l2ps3p	2	8
l2ps3	2	8
l2ps15p	2	8
l2ps15	2	8
l2ps11	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
l4ps8p	1	7
l4ps8	1	7
l3ps4p	1	7
l3ps4	1	7
l3ps12	1	7
l2ps6	1	7
l2ps2p	1	7
l2ps2	1	7
l2ps14p	1	7
l2ps14	1	7
l2ps10	1	7
l1ps9	1	7
l1ps7	1	7
l1ps5	1	7

Continued on Next Page...

Table A.2 – Continued

Instance	Cells	Cell Area
l1ps3	1	7
l1ps1p	1	7
l1ps15p	1	7
l1ps15	1	7
l1ps13	1	7
l1ps11	1	7
l1ps1	1	7
MUX_bit9	1	6
MUX_bit8	1	6
MUX_bit7	1	6
MUX_bit6	1	6
MUX_bit5	1	6
MUX_bit4	1	6
MUX_bit3	1	6
MUX_bit2	1	6
MUX_bit15	1	6
MUX_bit14	1	6
MUX_bit13	1	6
MUX_bit12	1	6
MUX_bit11	1	6
MUX_bit10	1	6
MUX_bit1	1	6
MUX_bit0	1	6
xsp_bit14	1	6
xsp_bit0	1	6
l4g151p	1	5
l3g70p	1	5
l3g70	1	5
l3g158p	1	5
l2g74	1	5
l2g30p	1	5
l2g30	1	5
l2g1512p	1	5
l2g118	1	5
l1p1514p	1	5
l1g98	1	5
l1g76	1	5
l1g54	1	5
l1g32	1	5
l1g1514p	1	5
l1g1312	1	5
l1g1110	1	5
l1g10p	1	5
l1g10	1	5
ycp_bit14	1	4
ycp_bit0	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
l3p158p	1	4
l3p148p	1	4
l3p148	1	4
l3p138	1	4
l3p128	1	4
l2p74	1	4
l2p64	1	4
l2p1512p	1	4
l2p1412p	1	4
l2p1412	1	4
l2p118	1	4
l2p108	1	4
l1p98	1	4
l1p76	1	4
l1p54	1	4
l1p32	1	4
l1p1312	1	4
l1p1110	1	4
Total	**172**	**857**

Table A.3: Timing of the existing ELMMA for modulo 49151

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[3]	2	0	0
xs_yc_bit0_15_gen[3].xs_scsa/G			
g10/A2		0	0
g10/Z	2	121	121
xs_yc_bit0_15_gen[3].xs_scsa/Eout			
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0/ps			
g10/A2		0	121
g10/Z	5	171	292
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0/Eout			
l1p43/Q			
g8/A1		0	292
g8/Z	2	81	373
l1p43/Pout			
l2g41p/P			
g14/A2		0	373
g14/Z	5	98	471
l2g41p/Gout			
l3g81p/H			
g14/A1		0	471
g14/Z	8	134	605
l3g81p/Gout			
l4g151p/H			
g14/A1		0	605
g14/Z	1	76	681
l4g151p/Gout			
cout_prime/x			
g2/A1		0	681
g2/Z	16	390	1071
cout_prime/Oout			
MUX_bit0/sel			
g23/S		0	1071
g23/Z	1	156	1227
MUX_bit0/pout			
elmmaout[0]		0	1227

Table A.4: Timing of the proposed ELMMA for modulo 49151

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[2]	2	0	0
xs_yc_bit0_15_gen[2].xs_scsa/G			
g10/A2		0	0
g10/Z	5	158	158
xs_yc_bit0_15_gen[2].xs_scsa/Eout			
l1p32/Q			
g8/A1		0	158
g8/Z	2	81	239
l1p32/Pout			
l2g30p/P			
g14/A2		0	239
g14/Z	5	98	337
l2g30p/Gout			
l3g70p/H			
g14/A1		0	337
g14/Z	9	142	479
l3g70p/Gout			
l4g151p/H			
g14/A1		0	479
g14/Z	16	415	894
l4g151p/Gout			
MUX_bit0/sel			
g23/S		0	894
g23/Z	1	156	1050
MUX_bit0/pout			
elmmaout[0]		0	1050

Table A.5: Total power of the existing ELMMA for modulo 49151

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l0p15p	1	0.937	626.662	627.599
l0p1p	1	0.932	389.899	390.832
xs_yc_bit0..en[2].xs_scsa	1	0.922	302.453	303.374
xs_yc_bit0..en[6].xs_scsa	1	0.922	264.646	265.568
xs_yc_bit0..en[3].xs_scsa	1	0.922	302.437	303.358

Continued on Next Page. . .

Table A.5 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..en[4].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..en[5].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..n[11].xs_scsa	1	0.921	264.630	265.552
xs_yc_bit0..n[12].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[14].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[15].xs_scsa	1	0.921	351.249	352.171
xs_yc_bit0..en[1].xs_scsa	1	0.921	392.924	393.845
xs_yc_bit0..en[8].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..en[9].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[10].xs_scsa	1	0.921	302.432	303.354
xs_yc_bit0..en[0].xs_scsa	1	0.921	250.226	251.148
xs_yc_bit0..en[7].xs_scsa	1	0.921	302.414	303.335
xs_yc_bit0..n[13].xs_scsa	1	0.921	264.610	265.531
l1ps2p	1	0.921	509.040	509.961
p_g_n0_to_..to_n15_level0	1	0.920	319.973	320.892
p_g_n0_to_..to_n15_level0	1	0.919	636.713	637.632
p_g_n0_to_..to_n15_level0	1	0.919	494.619	495.537
p_g_n0_to_..to_n15_level0	1	0.918	416.148	417.066
p_g_n0_to_..to_n15_level0	1	0.918	581.401	582.319
p_g_n0_to_..to_n15_level0	1	0.917	564.845	565.762
p_g_n0_to_..to_n15_level0	1	0.917	454.774	455.691
p_g_n0_to_..to_n15_level0	1	0.916	532.371	533.286
l0p14p	1	0.916	536.635	537.551
p_g_n0_to_..to_n15_level0	1	0.915	630.972	631.887
p_g_n0_to_..to_n15_level0	1	0.915	448.413	449.328
p_g_n0_to_..to_n15_level0	1	0.915	799.270	800.185
p_g_n0_to_..to_n15_level0	1	0.915	689.099	690.014
p_g_n0_to_..to_n15_level0	1	0.914	570.969	571.884
l1ps8	1	0.914	443.922	444.836
p_g_n0_to_..to_n15_level0	1	0.913	624.459	625.372
p_g_n0_to_..to_n15_level0	1	0.913	482.933	483.846
l1ps12	1	0.913	485.558	486.471
l1ps14p	1	0.913	561.215	562.127
l1ps14	1	0.913	560.930	561.843
l1ps2	1	0.912	393.890	394.802
l2ps3p	1	0.912	492.460	493.372
l1ps6	1	0.912	602.245	603.157
l1ps10	1	0.912	671.280	672.192
l1ps4	1	0.912	573.796	574.708
l3ps5p	1	0.911	420.452	421.364
l3ps5	1	0.911	382.090	383.002
l3ps13	1	0.908	767.154	768.063
l2ps15	1	0.908	513.019	513.927
l2ps3	1	0.908	474.239	475.147
l2ps15p	1	0.907	543.010	543.917
l4ps9	1	0.906	443.203	444.110
l4ps9p	1	0.906	449.880	450.786
l2ps11	1	0.904	608.946	609.851
l2ps7	1	0.898	639.272	640.170
MUX_bit15	1	0.633	375.492	376.125
MUX_bit0	1	0.633	354.280	354.913
MUX_bit10	1	0.632	335.973	336.606
MUX_bit14	1	0.632	483.496	484.129
MUX_bit6	1	0.632	302.021	302.653
MUX_bit13	1	0.632	357.362	357.993
MUX_bit3	1	0.631	338.929	339.561
MUX_bit11	1	0.631	301.579	302.210
MUX_bit4	1	0.631	320.303	320.933
l0g15p	1	0.625	215.172	215.797
MUX_bit12	1	0.625	338.482	339.107
MUX_bit2	1	0.623	319.853	320.475
l0g1p	1	0.618	293.128	293.746
MUX_bit8	1	0.618	318.197	318.815
l1p1211	1	0.617	388.063	388.680
l1p87	1	0.615	387.956	388.572
MUX_bit9	1	0.614	317.111	317.725
MUX_bit1	1	0.612	262.208	262.820
MUX_bit5	1	0.610	297.901	298.511
MUX_bit7	1	0.609	315.639	316.248
l1p109	1	0.599	418.861	419.460
l1p65	1	0.598	459.550	460.148
xs_yc_bit0..en[3].yc_scsa	1	0.597	114.636	115.232
xs_yc_bit0..en[4].yc_scsa	1	0.597	229.271	229.868
xs_yc_bit0..en[5].yc_scsa	1	0.597	152.847	153.444
xs_yc_bit0..en[2].yc_scsa	1	0.597	114.629	115.225
xs_yc_bit0..en[6].yc_scsa	1	0.597	152.838	153.435
xs_yc_bit0..n[11].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[12].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[14].yc_scsa	1	0.597	97.057	97.654
xs_yc_bit0..n[15].yc_scsa	1	0.597	108.708	109.305
xs_yc_bit0..en[1].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[8].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[9].yc_scsa	1	0.597	191.056	191.652

Continued on Next Page...

Table A.5 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..n[10].yc_scsa	1	0.596	152.841	153.438
xs_yc_bit0..en[0].yc_scsa	1	0.596	152.847	153.444
xs_yc_bit0..en[7].yc_scsa	1	0.596	76.424	77.020
l1p1413p	1	0.596	336.051	336.648
xs_yc_bit0..n[13].yc_scsa	1	0.596	214.989	215.586
l1p1413	1	0.595	192.700	193.294
l2p75	1	0.593	247.742	248.336
l1p43	1	0.592	224.604	225.196
l2p119	1	0.590	247.515	248.105
p_g_n0_to_..to_n15_level0	1	0.589	164.396	164.985
p_g_n0_to_..to_n15_level0	1	0.587	110.905	111.492
p_g_n0_to_..to_n15_level0	1	0.586	184.765	185.351
p_g_n0_to_..to_n15_level0	1	0.586	134.902	135.487
p_g_n0_to_..to_n15_level0	1	0.585	53.922	54.508
p_g_n0_to_..to_n15_level0	1	0.583	148.300	148.883
p_g_n0_to_..to_n15_level0	1	0.575	74.140	74.715
p_g_n0_to_..to_n15_level0	1	0.574	135.176	135.749
p_g_n0_to_..to_n15_level0	1	0.573	81.117	81.690
l0g14p	1	0.573	136.662	137.235
p_g_n0_to_..to_n15_level0	1	0.572	208.745	209.317
p_g_n0_to_..to_n15_level0	1	0.572	223.317	223.889
l2p85	1	0.571	223.439	224.009
l2p129	1	0.569	364.331	364.900
p_g_n0_to_..to_n15_level0	1	0.567	135.745	136.312
l3p139	1	0.565	61.742	62.307
l2p1513p	1	0.563	97.075	97.638
p_g_n0_to_..to_n15_level0	1	0.559	54.013	54.573
p_g_n0_to_..to_n15_level0	1	0.559	74.035	74.594
l3p149p	1	0.535	0.000	0.535
l3p149	1	0.531	51.982	52.513
l3p159p	1	0.509	0.000	0.509
l1g1211	1	0.503	159.007	159.509
l1g21p	1	0.487	335.696	336.183
l2g129	1	0.482	476.388	476.870
l1g87	1	0.482	111.758	112.240
l1g65	1	0.478	168.887	169.364
l1g21	1	0.474	276.574	277.048
l1g1413p	1	0.474	267.934	268.408
l1g1413	1	0.469	188.933	189.403
l1g109	1	0.464	209.159	209.623
l2g85	1	0.463	231.785	232.248
l1g43	1	0.448	178.102	178.550
l2g41p	1	0.443	388.114	388.558
l2g41	1	0.438	346.732	347.170
l3g81p	1	0.434	455.881	456.315
l3g81	1	0.434	409.964	410.397
l2g1513p	1	0.379	188.993	189.372
cout_prime	1	0.368	1402.140	1402.508
l3g159p	1	0.353	193.670	194.023
l4g151p	1	0.349	189.031	189.380
ycp_bit1	1	0.317	126.004	126.322
ycp_bit15	1	0.317	157.512	157.829
l3ps6p	2	0.033	393.704	393.736
l3ps6	2	0.033	354.461	354.494
l4ps10	2	0.032	438.901	438.933
l4ps10p	2	0.032	444.948	444.980
l2ps8	2	0.032	603.842	603.874
l2ps12	2	0.032	656.288	656.321
l2ps4p	2	0.032	437.005	437.037
l3ps14p	2	0.032	583.328	583.360
l3ps14	2	0.032	576.186	576.218
l2ps4	2	0.032	395.873	395.905
l3ps7p	2	0.032	402.658	402.690
l3ps7	2	0.032	396.793	396.824
l3ps15p	2	0.031	561.787	561.818
l3ps8p	2	0.031	373.940	373.971
l3ps8	2	0.031	368.305	368.336
l4ps11	2	0.031	370.395	370.425
l4ps11p	2	0.031	375.767	375.798
l3ps15	2	0.031	484.437	484.468
l4ps12	2	0.031	401.777	401.807
l4ps12p	2	0.031	407.737	407.767
l4ps13	2	0.030	417.292	417.322
l4ps13p	2	0.030	423.858	423.888
l4ps14p	2	0.030	435.096	435.126
l4ps14	2	0.030	425.986	426.016
l4ps15	2	0.030	379.134	379.164
l4ps15p	2	0.030	423.208	423.238
xsp_bit14	1	0.023	205.231	205.254
xsp_bit0	1	0.023	157.523	157.546
Total	**194**	**98.057**	**59072.305**	**59170.361**

Table A.6: Total power of the proposed ELMMA for modulo 49151

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l1ps15p	1	0.937	478.994	479.932
l1ps1p	1	0.932	389.901	390.833
l2ps2p	1	0.929	502.811	503.740
l3ps4p	1	0.923	469.380	470.303
l3ps4	1	0.922	462.832	463.754
xs_yc_bit0..en[2].xs_scsa	1	0.922	493.720	494.642
xs_yc_bit0..en[6].xs_scsa	1	0.922	367.267	368.188
xs_yc_bit0..en[3].xs_scsa	1	0.922	351.274	352.195
xs_yc_bit0..en[4].xs_scsa	1	0.922	493.702	494.623
xs_yc_bit0..en[5].xs_scsa	1	0.922	351.274	352.195
xs_yc_bit0..n[11].xs_scsa	1	0.921	307.363	308.284
xs_yc_bit0..n[12].xs_scsa	1	0.921	419.714	420.636
xs_yc_bit0..n[14].xs_scsa	1	0.921	378.040	378.961
xs_yc_bit0..n[15].xs_scsa	1	0.921	401.801	402.723
xs_yc_bit0..en[1].xs_scsa	1	0.921	401.801	402.723
xs_yc_bit0..en[8].xs_scsa	1	0.921	493.700	494.621
xs_yc_bit0..en[9].xs_scsa	1	0.921	351.272	352.193
xs_yc_bit0..n[10].xs_scsa	1	0.921	419.712	420.633
xs_yc_bit0..en[0].xs_scsa	1	0.921	250.226	251.148
xs_yc_bit0..en[7].xs_scsa	1	0.921	351.251	352.173
xs_yc_bit0..n[13].xs_scsa	1	0.921	307.343	308.264
l3ps12	1	0.921	734.366	735.287
l1ps1	1	0.920	319.987	320.906
l1ps7	1	0.919	443.585	444.504
l4ps8	1	0.919	462.083	463.001
l4ps8p	1	0.919	468.894	469.813
l1ps15	1	0.918	412.072	412.990
l1ps11	1	0.918	404.421	405.339
l1ps13	1	0.915	370.769	371.685
l1ps5	1	0.915	597.411	598.326
l2ps14	1	0.915	531.030	531.945
l2ps14p	1	0.915	553.406	554.321
l1ps9	1	0.915	518.495	519.410
l1ps3	1	0.913	469.443	470.357
l2ps6	1	0.912	581.630	582.542
l2ps2	1	0.911	353.057	353.968
l2ps10	1	0.911	621.672	622.583
MUX_bit15	1	0.633	375.323	375.956
MUX_bit0	1	0.633	353.986	354.619
MUX_bit10	1	0.632	335.828	336.461
MUX_bit14	1	0.632	483.255	483.887
MUX_bit6	1	0.632	301.870	302.502
MUX_bit13	1	0.632	357.203	357.834
MUX_bit3	1	0.631	338.762	339.393
MUX_bit11	1	0.631	301.426	302.057
MUX_bit4	1	0.631	320.156	320.787
MUX_bit12	1	0.625	338.333	338.959
MUX_bit2	1	0.623	319.548	320.171
MUX_bit8	1	0.618	318.086	318.704
MUX_bit9	1	0.614	316.944	317.558
l1p98	1	0.614	334.391	335.005
MUX_bit1	1	0.612	262.079	262.692
MUX_bit5	1	0.610	297.779	298.389
MUX_bit7	1	0.609	315.497	316.107
l1p54	1	0.606	334.422	335.028
l1p32	1	0.606	321.707	322.314
l1p1110	1	0.603	193.704	194.307
l1p1312	1	0.600	472.311	472.911
xs_yc_bit0..en[3].yc_scsa	1	0.597	82.728	83.324
xs_yc_bit0..en[4].yc_scsa	1	0.597	227.052	227.648
xs_yc_bit0..en[5].yc_scsa	1	0.597	110.304	110.900
xs_yc_bit0..en[2].yc_scsa	1	0.597	113.519	114.116
xs_yc_bit0..en[6].yc_scsa	1	0.597	151.359	151.955
xs_yc_bit0..n[11].yc_scsa	1	0.597	82.725	83.322
xs_yc_bit0..n[12].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..n[14].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..n[15].yc_scsa	1	0.597	110.301	110.897
xs_yc_bit0..en[1].yc_scsa	1	0.597	100.640	101.236
xs_yc_bit0..en[8].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..en[9].yc_scsa	1	0.597	137.876	138.472
xs_yc_bit0..n[10].yc_scsa	1	0.596	151.362	151.958
xs_yc_bit0..en[0].yc_scsa	1	0.596	151.368	151.964
xs_yc_bit0..en[7].yc_scsa	1	0.596	55.152	55.748
xs_yc_bit0..n[13].yc_scsa	1	0.596	110.301	110.897
l1p76	1	0.595	258.365	258.959
l2p108	1	0.586	217.966	218.552
l2p1412	1	0.579	222.119	222.698
l2p1412p	1	0.577	160.926	161.504
l2p118	1	0.575	286.562	287.137
l3p128	1	0.573	123.605	124.178
l2p64	1	0.571	217.917	218.488
l2p1512p	1	0.562	96.771	97.333

Continued on Next Page...

Table A.6 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l2p74	1	0.555	161.156	161.710
l3p138	1	0.534	31.152	31.686
l3p148	1	0.528	26.261	26.789
l3p148p	1	0.509	0.000	0.509
l3p158p	1	0.507	0.000	0.507
l1g1514p	1	0.493	195.210	195.703
l1g10p	1	0.485	262.731	263.216
l1g10	1	0.467	168.105	168.572
l1g1110	1	0.465	135.720	136.185
l1g76	1	0.464	110.549	111.013
l1g98	1	0.459	207.391	207.850
l1p1514p	1	0.452	264.185	264.637
l1g54	1	0.450	204.833	205.283
l1g1312	1	0.447	358.074	358.521
l1g32	1	0.432	145.301	145.734
l2g118	1	0.428	470.892	471.321
l2g30p	1	0.425	342.814	343.239
l2g30	1	0.416	346.290	346.706
l2g74	1	0.414	233.348	233.761
l3g70p	1	0.407	491.549	491.956
l3g70	1	0.406	445.488	445.895
l2g1512p	1	0.332	166.424	166.756
l3g158p	1	0.320	169.828	170.148
l4g151p	1	0.318	1434.894	1435.212
ycp_bit0	1	0.317	124.499	124.817
ycp_bit14	1	0.317	155.630	155.947
l2ps3p	2	0.034	462.598	462.632
l4ps9	2	0.034	431.179	431.212
l4ps9p	2	0.034	436.848	436.882
l3ps5p	2	0.033	420.570	420.604
l3ps5	2	0.033	381.634	381.667
l3ps13	2	0.033	734.187	734.220
l2ps3	2	0.033	440.870	440.903
l3ps6p	2	0.033	394.448	394.480
l3ps6	2	0.033	355.190	355.222
l2ps15p	2	0.032	521.257	521.290
l4ps10	2	0.032	439.376	439.409
l4ps10p	2	0.032	445.451	445.483
l2ps15	2	0.032	479.853	479.885
l2ps11	2	0.032	592.349	592.381
l3ps7p	2	0.032	399.255	399.287
l2ps7	2	0.032	623.866	623.898
l3ps14p	2	0.032	582.444	582.476
l3ps7	2	0.032	393.605	393.637
l3ps14	2	0.032	575.695	575.727
l3ps15	2	0.032	472.227	472.259
l4ps11	2	0.031	367.335	367.366
l4ps11p	2	0.031	372.562	372.593
l4ps12	2	0.031	404.828	404.859
l4ps12p	2	0.031	410.999	411.030
l4ps13	2	0.031	413.272	413.303
l4ps13p	2	0.031	419.620	419.650
l3ps15p	2	0.031	548.845	548.876
l4ps14p	2	0.031	435.052	435.083
l4ps15p	2	0.031	423.207	423.237
l4ps14	2	0.031	425.937	425.967
l4ps15	2	0.031	379.084	379.115
xsp_bit14	1	0.023	325.562	325.585
xsp_bit0	1	0.023	157.523	157.546
Total	**172**	**73.621**	**49987.489**	**50061.11**

Synthesis Result for Modulo 49152

B

Table B.1: Area of the existing ELMMA for modulo 49152

Instance	Cells	Cell Area
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13	2	8
l4ps12	2	8
l4ps11	2	8
l4ps10	2	8
l3ps8	2	8
l3ps7	2	8
l3ps6	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l2ps8	2	8
l2ps4	2	8
l2ps12	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
p_g_n0_to_n15_level0_gen[9].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[8].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[7].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[6].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[5].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[4].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[2].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[1].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[15].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[14].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[13].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[12].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[11].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[10].p_n1_to_n15_level0	1	7
l4ps9	1	7
l3ps5	1	7
l3ps13	1	7
l2ps7	1	7
l2ps3	1	7
l2ps15p	1	7
l2ps15	1	7
l2ps11	1	7
l1ps8	1	7
l1ps6	1	7
l1ps4	1	7
l1ps2	1	7
l1ps14p	1	7
l1ps14	1	7
l1ps12	1	7
l1ps10	1	7
l0p15p	1	7
l0p14p	1	7
MUX_bit15	1	6
MUX_bit14	1	6
xsp_bit14	1	6

Continued on Next Page...

Table B.1 – Continued

Instance	Cells	Cell Area
l4g151p	1	5
l3g81	1	5
l3g159p	1	5
l2g85	1	5
l2g41	1	5
l2g1513p	1	5
l2g129	1	5
l1g87	1	5
l1g65	1	5
l1g43	1	5
l1g21	1	5
l1g1413p	1	5
l1g1413	1	5
l1g1211	1	5
l1g109	1	5
ycp_bit15	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
p_g_n0_to_n15_level0_gen[9].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[8].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[7].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[6].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[5].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[4].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[3].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[2].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[1].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[14].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[13].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[12].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[11].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[10].g_n1_to_n15_level0	1	4
l3p159p	1	4
l3p149p	1	4
l3p149	1	4
l3p139	1	4
l2p85	1	4
l2p75	1	4
l2p1513p	1	4
l2p129	1	4
l2p119	1	4
l1p87	1	4
l1p65	1	4
l1p43	1	4
l1p1413p	1	4
l1p1413	1	4
l1p1211	1	4
l1p109	1	4
l0g15p	1	4
l0g14p	1	4
cout_prime	1	4
Total	153	767

Table B.2: Area of the proposed ELMMA for modulo 49152

Instance	Cells	Cell Area
l4ps9	2	8
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13	2	8
l4ps12	2	8
l4ps11	2	8
l4ps10	2	8
l3ps7	2	8
l3ps6	2	8

Continued on Next Page...

Table B.2 – Continued

Instance	Cells	Cell Area
l3ps5	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l3ps13	2	8
l2ps7	2	8
l2ps3	2	8
l2ps15p	2	8
l2ps15	2	8
l2ps11	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
l4ps8	1	7
l3ps4	1	7
l3ps12	1	7
l2ps6	1	7
l2ps2	1	7
l2ps14p	1	7
l2ps14	1	7
l2ps10	1	7
l1ps9	1	7
l1ps7	1	7
l1ps5	1	7
l1ps3	1	7
l1ps15p	1	7
l1ps15	1	7
l1ps13	1	7
l1ps11	1	7
l1ps1	1	7
MUX_bit15	1	6
MUX_bit14	1	6
xsp_bit14	1	6
l4g151p	1	5
l3g70	1	5
l3g158p	1	5
l2g74	1	5
l2g30	1	5
l2g1512p	1	5
l2g118	1	5
l1p1514p	1	5
l1g98	1	5
l1g76	1	5
l1g54	1	5
l1g32	1	5
l1g1514p	1	5
l1g1312	1	5
l1g1110	1	5
l1g10	1	5
ycp_bit14	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
l3p158p	1	4
l3p148p	1	4
l3p148	1	4
l3p138	1	4

Continued on Next Page...

Table B.2 – Continued

Instance	Cells	Cell Area
l3p128	1	4
l2p74	1	4
l2p64	1	4
l2p1512p	1	4
l2p1412p	1	4
l2p1412	1	4
l2p118	1	4
l2p108	1	4
l1p98	1	4
l1p76	1	4
l1p54	1	4
l1p32	1	4
l1p1312	1	4
l1p1110	1	4
Total	**131**	**640**

Table B.3: Timing of the existing ELMMA for modulo 49152

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[10]	2	0	0
xs_yc_bit0_15_gen[10].xs_scsa/G			
g10/A2		0	0
g10/Z	2	121	121
xs_yc_bit0_15_gen[10].xs_scsa/Eout			
p_g_n0_to_n15_level0_gen[10].p_n1_to_n15_level0/ps			
g10/A2		0	121
g10/Z	3	137	259
p_g_n0_to_n15_level0_gen[10].p_n1_to_n15_level0/Eout			
l1g109/P			
g14/A1		0	259
g14/Z	3	88	347
l1g109/Gout			
l2g129/H			
g14/A1		0	347
g14/Z	6	112	458
l2g129/Gout			
l3ps14p/G			
g31/A2		0	458
g31/ZN	2	87	546
g30/A2		0	546
g30/ZN	2	155	701
l3ps14p/Sout			
l4ps14p/ps			
g30/A1		0	701
g30/ZN	1	152	853
l4ps14p/Sout			
MUX_bit14/pin1			
g23/I1		0	853
g23/Z	1	90	943
MUX_bit14/pout			
elmmaout[14]		0	943

Table B.4: Timing of the proposed ELMMA for modulo 49152

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[14]	5	0	0
xsp_bit14/y			
g12/B1		0	0
g12/ZN	4	223	223
xsp_bit14/notEout			
l2ps15p/P			
g31/A1		0	223
g31/ZN	2	140	363
g30/A2		0	363
g30/ZN	2	165	528
l2ps15p/Sout			
l3ps15p/ps			
g30/A2		0	528
g30/ZN	2	153	681
l3ps15p/Sout			
l4ps15p/ps			
g30/A2		0	681
g30/ZN	1	142	823
l4ps15p/Sout			
MUX_bit15/pin1			

Continued on Next Page...

Table B.4 – Continued

Pin	Fanout Load	Delay (ps)	Arrival (ps)
g23/I1		0	823
g23/Z	1	90	913
MUX_bit15/pout			
elmmaout[15]		0	913

Table B.5: Total power of the existing ELMMA for modulo 49152

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l0p15p	1	0.937	626.662	627.599
xs_yc_bit0..en[2].xs_scsa	1	0.922	302.453	303.374
xs_yc_bit0..en[6].xs_scsa	1	0.922	264.646	265.568
xs_yc_bit0..en[3].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..en[4].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..en[5].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..n[11].xs_scsa	1	0.921	264.630	265.552
xs_yc_bit0..n[12].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[14].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[15].xs_scsa	1	0.921	351.249	352.171
xs_yc_bit0..en[1].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..en[8].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..en[9].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[10].xs_scsa	1	0.921	302.432	303.354
xs_yc_bit0..en[0].xs_scsa	1	0.921	210.457	211.379
xs_yc_bit0..en[7].xs_scsa	1	0.921	302.414	303.335
xs_yc_bit0..n[13].xs_scsa	1	0.921	264.610	265.531
p_g_n0_to_..to_n15_level0	1	0.920	269.196	270.115
p_g_n0_to_..to_n15_level0	1	0.919	636.713	637.632
p_g_n0_to_..to_n15_level0	1	0.919	494.493	495.412
p_g_n0_to_..to_n15_level0	1	0.918	416.148	417.066
p_g_n0_to_..to_n15_level0	1	0.918	581.401	582.319
p_g_n0_to_..to_n15_level0	1	0.917	437.353	438.270
p_g_n0_to_..to_n15_level0	1	0.917	454.774	455.691
p_g_n0_to_..to_n15_level0	1	0.916	532.371	533.286
l0p14p	1	0.916	536.635	537.551
p_g_n0_to_..to_n15_level0	1	0.915	630.972	631.887
p_g_n0_to_..to_n15_level0	1	0.915	448.413	449.328
p_g_n0_to_..to_n15_level0	1	0.915	611.595	612.510
p_g_n0_to_..to_n15_level0	1	0.915	530.424	531.339
p_g_n0_to_..to_n15_level0	1	0.914	570.820	571.734
l1ps8	1	0.914	443.922	444.836
p_g_n0_to_..to_n15_level0	1	0.913	480.280	481.193
p_g_n0_to_..to_n15_level0	1	0.913	482.809	483.722
l1ps12	1	0.913	485.558	486.471
l1ps14p	1	0.913	561.215	562.127
l1ps14	1	0.913	560.930	561.843
l1ps2	1	0.912	333.489	334.401
l1ps6	1	0.912	475.367	476.279
l1ps10	1	0.912	535.093	536.005
l1ps4	1	0.912	457.060	457.971
l3ps5	1	0.911	311.689	312.600
l3ps13	1	0.908	600.099	601.008
l2ps15	1	0.908	513.019	513.927
l2ps3	1	0.908	388.211	389.119
l2ps15p	1	0.907	543.010	543.917
l4ps9	1	0.906	364.254	365.160
l2ps11	1	0.904	471.215	472.120
l2ps7	1	0.898	491.754	492.652
MUX_bit15	1	0.633	341.627	342.260
MUX_bit14	1	0.632	435.302	435.934
l0g15p	1	0.625	215.172	215.797
l1p1211	1	0.617	388.063	388.680
l1p87	1	0.615	387.956	388.572
l1p109	1	0.602	367.072	367.674
l1p65	1	0.600	405.060	405.659
xs_yc_bit0..en[3].yc_scsa	1	0.597	114.636	115.232
xs_yc_bit0..en[4].yc_scsa	1	0.597	229.271	229.868
xs_yc_bit0..en[5].yc_scsa	1	0.597	152.847	153.444
xs_yc_bit0..en[2].yc_scsa	1	0.597	114.629	115.225
xs_yc_bit0..en[6].yc_scsa	1	0.597	152.838	153.435
xs_yc_bit0..n[11].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[12].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[14].yc_scsa	1	0.597	97.057	97.654
xs_yc_bit0..n[15].yc_scsa	1	0.597	108.708	109.305
xs_yc_bit0..en[1].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[8].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[9].yc_scsa	1	0.597	191.056	191.652
xs_yc_bit0..n[10].yc_scsa	1	0.596	152.841	153.438
xs_yc_bit0..en[0].yc_scsa	1	0.596	152.847	153.444
xs_yc_bit0..en[7].yc_scsa	1	0.596	76.424	77.020
l1p1413p	1	0.596	336.051	336.648

Continued on Next Page...

Table B.5 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..n[13].yc_scsa	1	0.596	214.989	215.586
l1p1413	1	0.595	192.700	193.294
l2p75	1	0.593	208.097	208.691
l1p43	1	0.592	191.962	192.554
l2p119	1	0.590	207.832	208.422
p_g_n0_to_..to_n15_level0	1	0.589	135.566	136.155
p_g_n0_to_..to_n15_level0	1	0.587	110.905	111.492
p_g_n0_to_..to_n15_level0	1	0.586	184.765	185.351
p_g_n0_to_..to_n15_level0	1	0.586	134.902	135.487
p_g_n0_to_..to_n15_level0	1	0.585	53.922	54.508
p_g_n0_to_..to_n15_level0	1	0.583	148.300	148.883
p_g_n0_to_..to_n15_level0	1	0.575	74.140	74.715
p_g_n0_to_..to_n15_level0	1	0.574	135.176	135.749
p_g_n0_to_..to_n15_level0	1	0.573	81.117	81.690
l0g14p	1	0.573	136.662	137.235
p_g_n0_to_..to_n15_level0	1	0.572	208.745	209.317
p_g_n0_to_..to_n15_level0	1	0.572	223.317	223.889
l2p85	1	0.571	185.628	186.199
l2p129	1	0.569	330.813	331.383
p_g_n0_to_..to_n15_level0	1	0.567	135.745	136.312
l3p139	1	0.565	61.740	62.305
l2p1513p	1	0.563	97.075	97.638
p_g_n0_to_..to_n15_level0	1	0.559	54.013	54.573
p_g_n0_to_..to_n15_level0	1	0.559	74.035	74.594
l3p149p	1	0.535	0.000	0.535
l3p149	1	0.531	51.917	52.448
l3p159p	1	0.509	0.000	0.509
l1g1211	1	0.503	159.007	159.509
l2g129	1	0.482	476.388	476.870
l1g87	1	0.482	111.758	112.240
l1g65	1	0.478	168.884	169.362
l1g21	1	0.474	271.229	271.703
l1g1413p	1	0.474	267.934	268.408
l1g1413	1	0.469	188.933	189.403
l1g109	1	0.464	209.157	209.621
l2g85	1	0.463	185.179	185.642
l1g43	1	0.448	142.838	143.286
l2g41	1	0.437	347.879	348.316
l3g81	1	0.434	526.287	526.720
l2g1513p	1	0.379	188.993	189.372
cout_prime	1	0.368	269.360	269.728
l3g159p	1	0.353	193.670	194.023
l4g151p	1	0.349	189.031	189.380
ycp_bit15	1	0.317	157.512	157.829
l3ps6	2	0.033	290.214	290.247
l4ps10	2	0.032	359.397	359.430
l2ps8	2	0.032	448.561	448.593
l2ps12	2	0.032	502.774	502.807
l3ps14p	2	0.032	583.328	583.360
l3ps14	2	0.032	576.186	576.218
l2ps4	2	0.032	321.680	321.712
l3ps7	2	0.032	322.561	322.592
l3ps15p	2	0.031	561.787	561.818
l3ps8	2	0.031	284.447	284.478
l4ps11	2	0.031	301.080	301.110
l3ps15	2	0.031	484.437	484.468
l4ps12	2	0.031	320.969	321.000
l4ps13	2	0.030	333.608	333.638
l4ps14p	2	0.030	435.069	435.099
l4ps14	2	0.030	425.949	425.979
l4ps15	2	0.030	379.097	379.127
l4ps15p	2	0.030	423.208	423.238
xsp_bit14	1	0.023	205.231	205.254
Total	**153**	**82.169**	**42854.753**	**42936.922**

Table B.6: Total power of the proposed ELMMA for modulo 49152

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l1ps15p	1	0.937	478.994	479.932
l3ps4	1	0.922	383.032	383.954
xs_yc_bit0..en[2].xs_scsa	1	0.922	378.058	378.980
xs_yc_bit0..en[6].xs_scsa	1	0.922	367.267	368.188
xs_yc_bit0..en[3].xs_scsa	1	0.922	351.177	352.099
xs_yc_bit0..en[4].xs_scsa	1	0.922	378.042	378.964
xs_yc_bit0..en[5].xs_scsa	1	0.922	351.177	352.099
xs_yc_bit0..n[11].xs_scsa	1	0.921	307.363	308.284
xs_yc_bit0..n[12].xs_scsa	1	0.921	419.714	420.636
xs_yc_bit0..n[14].xs_scsa	1	0.921	378.040	378.961
xs_yc_bit0..n[15].xs_scsa	1	0.921	401.801	402.723
xs_yc_bit0..en[1].xs_scsa	1	0.921	308.404	309.326

Continued on Next Page...

Table B.6 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..en[8].xs_scsa	1	0.921	378.040	378.961
xs_yc_bit0..en[9].xs_scsa	1	0.921	351.175	352.096
xs_yc_bit0..n[10].xs_scsa	1	0.921	419.712	420.633
xs_yc_bit0..en[0].xs_scsa	1	0.921	210.457	211.379
xs_yc_bit0..en[7].xs_scsa	1	0.921	351.251	352.173
xs_yc_bit0..n[13].xs_scsa	1	0.921	307.343	308.264
l3ps12	1	0.921	576.760	577.681
l1ps1	1	0.920	269.200	270.120
l1ps7	1	0.919	443.585	444.504
l4ps8	1	0.919	384.778	385.697
l1ps15	1	0.918	412.072	412.990
l1ps11	1	0.918	404.421	405.339
l1ps13	1	0.915	370.769	371.685
l1ps5	1	0.915	470.117	471.032
l2ps14	1	0.915	531.030	531.945
l2ps14p	1	0.915	553.406	554.321
l1ps9	1	0.915	411.028	411.943
l1ps3	1	0.913	371.799	372.712
l2ps6	1	0.912	454.159	455.071
l2ps2	1	0.911	288.088	288.998
l2ps10	1	0.911	484.170	485.080
MUX_bit15	1	0.633	341.665	342.298
MUX_bit14	1	0.632	435.364	435.996
l1p98	1	0.615	294.404	295.019
l1p32	1	0.611	273.194	273.806
l1p54	1	0.608	294.371	294.979
l1p1110	1	0.603	193.704	194.307
l1p1312	1	0.600	472.311	472.911
xs_yc_bit0..en[3].yc_scsa	1	0.597	82.728	83.324
xs_yc_bit0..en[4].yc_scsa	1	0.597	227.052	227.648
xs_yc_bit0..en[5].yc_scsa	1	0.597	110.304	110.900
xs_yc_bit0..en[2].yc_scsa	1	0.597	113.519	114.116
xs_yc_bit0..en[6].yc_scsa	1	0.597	151.359	151.955
xs_yc_bit0..n[11].yc_scsa	1	0.597	82.725	83.322
xs_yc_bit0..n[12].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..n[14].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..n[15].yc_scsa	1	0.597	110.301	110.897
xs_yc_bit0..en[1].yc_scsa	1	0.597	82.725	83.322
xs_yc_bit0..en[8].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..en[9].yc_scsa	1	0.597	137.876	138.472
xs_yc_bit0..n[10].yc_scsa	1	0.596	151.362	151.958
xs_yc_bit0..en[0].yc_scsa	1	0.596	151.368	151.964
xs_yc_bit0..en[7].yc_scsa	1	0.596	55.152	55.748
xs_yc_bit0..n[13].yc_scsa	1	0.596	110.301	110.897
l1p76	1	0.595	258.365	258.959
l2p108	1	0.586	183.476	184.063
l2p1412	1	0.579	222.119	222.698
l2p1412p	1	0.577	160.926	161.504
l2p118	1	0.575	262.882	263.457
l3p128	1	0.573	103.925	104.499
l2p64	1	0.571	183.420	183.991
l2p1512p	1	0.562	96.771	97.333
l2p74	1	0.555	134.129	134.683
l3p138	1	0.534	31.147	31.682
l3p148	1	0.528	26.236	26.764
l3p148p	1	0.509	0.000	0.509
l3p158p	1	0.507	0.000	0.507
l1g1514p	1	0.493	195.210	195.703
l1g10	1	0.467	164.535	165.002
l1g1110	1	0.465	135.720	136.185
l1g76	1	0.464	110.549	111.013
l1g98	1	0.459	207.389	207.848
l1p1514p	1	0.452	264.185	264.637
l1g54	1	0.450	204.830	205.280
l1g1312	1	0.447	358.074	358.521
l1g32	1	0.432	115.892	116.324
l2g118	1	0.428	470.892	471.321
l2g30	1	0.417	345.575	345.993
l2g74	1	0.414	186.669	187.082
l3g70	1	0.406	561.898	562.304
l2g1512p	1	0.332	166.424	166.756
l3g158p	1	0.320	169.828	170.148
l4g151p	1	0.318	301.280	301.598
ycp_bit14	1	0.317	155.630	155.947
l4ps9	2	0.034	356.103	356.136
l3ps5	2	0.033	317.352	317.385
l3ps13	2	0.033	558.785	558.818
l2ps3	2	0.033	361.521	361.554
l3ps6	2	0.033	290.953	290.986
l2ps15p	2	0.032	521.257	521.290
l4ps10	2	0.032	359.997	360.030
l2ps15	2	0.032	479.853	479.885
l2ps11	2	0.032	458.081	458.113

Continued on Next Page. . .

Table B.6 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l2ps7	2	0.032	468.700	468.732
l3ps14p	2	0.032	569.002	569.034
l3ps14	2	0.032	562.308	562.340
l3ps7	2	0.032	313.929	313.961
l3ps15	2	0.032	460.119	460.151
l4ps11	2	0.031	296.615	296.647
l4ps12	2	0.031	325.914	325.945
l3ps15p	2	0.031	535.193	535.224
l4ps13	2	0.031	317.726	317.756
l4ps14p	2	0.031	417.497	417.528
l4ps15	2	0.030	364.222	364.252
l4ps15p	2	0.030	403.229	403.260
l4ps14	2	0.030	408.821	408.851
xsp_bit14	1	0.023	325.562	325.585
Total	**131**	**59.243**	**34179.260**	**34238.503**

Synthesis Result for Modulo 57343

C

Table C.1: Area of the existing ELMMA for modulo 57343

Instance	Cells	Cell Area
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13p	2	8
l4ps13	2	8
l4ps12p	2	8
l4ps12	2	8
l4ps11p	2	8
l4ps11	2	8
l4ps10p	2	8
l4ps10	2	8
l3ps8p	2	8
l3ps8	2	8
l3ps7p	2	8
l3ps7	2	8
l3ps6p	2	8
l3ps6	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l2ps8	2	8
l2ps4p	2	8
l2ps4	2	8
l2ps12	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
p_g_n0_to_n15_level0_gen[9].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[8].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[7].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[6].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[5].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[4].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[2].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[1].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[15].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[14].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[13].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[12].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[11].p_n1_to_n15_level0	1	7
p_g_n0_to_n15_level0_gen[10].p_n1_to_n15_level0	1	7
l4ps9p	1	7
l4ps9	1	7
l3ps5p	1	7
l3ps5	1	7
l3ps13p	1	7
l3ps13	1	7
l2ps7	1	7
l2ps3p	1	7
l2ps3	1	7
l2ps15p	1	7
l2ps15	1	7
l2ps11	1	7
l1ps8	1	7

Continued on Next Page...

Table C.1 – Continued

Instance	Cells	Cell Area
l1ps6	1	7
l1ps4	1	7
l1ps2p	1	7
l1ps2	1	7
l1ps14p	1	7
l1ps14	1	7
l1ps12	1	7
l1ps10	1	7
l0p1p	1	7
l0p14p	1	7
l0p13p	1	7
MUX_bit9	1	6
MUX_bit8	1	6
MUX_bit7	1	6
MUX_bit6	1	6
MUX_bit5	1	6
MUX_bit4	1	6
MUX_bit3	1	6
MUX_bit2	1	6
MUX_bit15	1	6
MUX_bit14	1	6
MUX_bit13	1	6
MUX_bit12	1	6
MUX_bit11	1	6
MUX_bit10	1	6
MUX_bit1	1	6
MUX_bit0	1	6
xsp_bit13	1	6
xsp_bit0	1	6
l4g151p	1	5
l3g81p	1	5
l3g81	1	5
l3g159p	1	5
l2g85	1	5
l2g41p	1	5
l2g41	1	5
l2g1513p	1	5
l2g129	1	5
l1g87	1	5
l1g65	1	5
l1g43	1	5
l1g21p	1	5
l1g21	1	5
l1g1413p	1	5
l1g1413	1	5
l1g1211	1	5
l1g109	1	5
ycp_bit14	1	4
ycp_bit1	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
p_g_n0_to_n15_level0_gen[9].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[8].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[7].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[6].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[5].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[4].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[3].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[2].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[1].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[15].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[14].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[13].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[12].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[11].g_n1_to_n15_level0	1	4
p_g_n0_to_n15_level0_gen[10].g_n1_to_n15_level0	1	4
l3p159p	1	4
l3p149p	1	4
l3p149	1	4
l3p139p	1	4

Continued on Next Page...

Table C.1 – Continued

Instance	Cells	Cell Area
l3p139	1	4
l2p85	1	4
l2p75	1	4
l2p1513p	1	4
l2p129	1	4
l2p119	1	4
l1p87	1	4
l1p65	1	4
l1p43	1	4
l1p1413p	1	4
l1p1413	1	4
l1p1211	1	4
l1p109	1	4
l0g1p	1	4
l0g14p	1	4
l0g13p	1	4
cout_prime	1	4
Total	**197**	**1001**

Table C.2: Area of the proposed ELMMA for modulo 57343

Instance	Cells	Cell Area
l4ps9p	2	8
l4ps9	2	8
l4ps15p	2	8
l4ps15	2	8
l4ps14p	2	8
l4ps14	2	8
l4ps13p	2	8
l4ps13	2	8
l4ps12p	2	8
l4ps12	2	8
l4ps11p	2	8
l4ps11	2	8
l4ps10p	2	8
l4ps10	2	8
l3ps7p	2	8
l3ps7	2	8
l3ps6p	2	8
l3ps6	2	8
l3ps5p	2	8
l3ps5	2	8
l3ps15p	2	8
l3ps15	2	8
l3ps14p	2	8
l3ps14	2	8
l3ps13p	2	8
l3ps13	2	8
l2ps7	2	8
l2ps3p	2	8
l2ps3	2	8
l2ps15	2	8
l2ps11	2	8
xs_yc_bit0_15_gen[9].xs_scsa	1	7
xs_yc_bit0_15_gen[8].xs_scsa	1	7
xs_yc_bit0_15_gen[7].xs_scsa	1	7
xs_yc_bit0_15_gen[6].xs_scsa	1	7
xs_yc_bit0_15_gen[5].xs_scsa	1	7
xs_yc_bit0_15_gen[4].xs_scsa	1	7
xs_yc_bit0_15_gen[3].xs_scsa	1	7
xs_yc_bit0_15_gen[2].xs_scsa	1	7
xs_yc_bit0_15_gen[1].xs_scsa	1	7
xs_yc_bit0_15_gen[15].xs_scsa	1	7
xs_yc_bit0_15_gen[14].xs_scsa	1	7
xs_yc_bit0_15_gen[13].xs_scsa	1	7
xs_yc_bit0_15_gen[12].xs_scsa	1	7
xs_yc_bit0_15_gen[11].xs_scsa	1	7
xs_yc_bit0_15_gen[10].xs_scsa	1	7
xs_yc_bit0_15_gen[0].xs_scsa	1	7
l4ps8p	1	7
l4ps8	1	7
l3ps4p	1	7
l3ps4	1	7
l3ps12	1	7
l2ps6	1	7
l2ps2p	1	7
l2ps2	1	7
l2ps14p	1	7
l2ps14	1	7
l2ps10	1	7
l1ps9	1	7

Continued on Next Page...

Table C.2 – Continued

Instance	Cells	Cell Area
l1ps7	1	7
l1ps5	1	7
l1ps3	1	7
l1ps1p	1	7
l1ps15	1	7
l1ps14p	1	7
l1ps13p	1	7
l1ps13	1	7
l1ps11	1	7
l1ps1	1	7
MUX_bit9	1	6
MUX_bit8	1	6
MUX_bit7	1	6
MUX_bit6	1	6
MUX_bit5	1	6
MUX_bit4	1	6
MUX_bit3	1	6
MUX_bit2	1	6
MUX_bit15	1	6
MUX_bit14	1	6
MUX_bit13	1	6
MUX_bit12	1	6
MUX_bit11	1	6
MUX_bit10	1	6
MUX_bit1	1	6
MUX_bit0	1	6
xsp_bit13	1	6
xsp_bit0	1	6
l4g150p	1	5
l3g70p	1	5
l3g70	1	5
l3g158p	1	5
l2g74	1	5
l2g30p	1	5
l2g30	1	5
l2g1512p	1	5
l2g118	1	5
l1g98	1	5
l1g76	1	5
l1g54	1	5
l1g32	1	5
l1g1514	1	5
l1g1312	1	5
l1g1110	1	5
l1g10p	1	5
l1g10	1	5
ycp_bit13	1	4
ycp_bit0	1	4
xs_yc_bit0_15_gen[9].yc_scsa	1	4
xs_yc_bit0_15_gen[8].yc_scsa	1	4
xs_yc_bit0_15_gen[7].yc_scsa	1	4
xs_yc_bit0_15_gen[6].yc_scsa	1	4
xs_yc_bit0_15_gen[5].yc_scsa	1	4
xs_yc_bit0_15_gen[4].yc_scsa	1	4
xs_yc_bit0_15_gen[3].yc_scsa	1	4
xs_yc_bit0_15_gen[2].yc_scsa	1	4
xs_yc_bit0_15_gen[1].yc_scsa	1	4
xs_yc_bit0_15_gen[15].yc_scsa	1	4
xs_yc_bit0_15_gen[14].yc_scsa	1	4
xs_yc_bit0_15_gen[13].yc_scsa	1	4
xs_yc_bit0_15_gen[12].yc_scsa	1	4
xs_yc_bit0_15_gen[11].yc_scsa	1	4
xs_yc_bit0_15_gen[10].yc_scsa	1	4
xs_yc_bit0_15_gen[0].yc_scsa	1	4
l3p158p	1	4
l3p148p	1	4
l3p148	1	4
l3p138p	1	4
l3p138	1	4
l3p128	1	4
l2p74	1	4
l2p64	1	4
l2p1412p	1	4
l2p1412	1	4
l2p118	1	4
l2p108	1	4
l1p98	1	4
l1p76	1	4
l1p54	1	4
l1p32	1	4
l1p1312p	1	4
l1p1312	1	4
l1p1110	1	4
l1h1514p	1	4

Continued on Next Page...

Table C.2 – Continued

Instance	Cells	Cell Area
l1g1312p	1	4
cout	1	4
Total	**181**	**895**

Table C.3: Timing of the existing ELMMA for modulo 57343

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[3]	2	0	0
xs_yc_bit0_15_gen[3].xs_scsa/G			
g10/A2		0	0
g10/Z	2	121	121
xs_yc_bit0_15_gen[3].xs_scsa/Eou			
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0/ps			
g10/A2		0	121
g10/Z	5	171	292
p_g_n0_to_n15_level0_gen[3].p_n1_to_n15_level0/Eout			
l1p43/Q			
g8/A1		0	292
g8/Z	2	81	373
l1p43/Pout			
l2g41/P			
g14/A2		0	373
g14/Z	5	98	471
l2g41/Gout			
l3g81/H			
g14/A1		0	471
g14/Z	11	158	629
l3g81/Gout			
l4g151p/H			
g14/A1		0	629
g14/Z	1	81	710
l4g151p/Gout			
cout_prime/x			
g2/A1		0	710
g2/Z	16	390	1100
cout_prime/Oout			
MUX_bit0/sel			
g23/S		0	1100
g23/Z	1	156	1256
MUX_bit0/pout			
elmmaout[0]		0	1256

Table C.4: Timing of the proposed ELMMA for modulo 57343

Pin	Fanout Load	Delay (ps)	Arrival (ps)
yin[2]	2	0	0
xs_yc_bit0_15_gen[2].xs_scsa/G			
g10/A2		0	0
g10/Z	5	158	158
xs_yc_bit0_15_gen[2].xs_scsa/Eout			
l1p32/Q			
g8/A1		0	158
g8/Z	2	81	239
l1p32/Pout			
l2g30/P			
g14/A2		0	239
g14/Z	5	98	337
l2g30/Gout			
l3g70/H			
g14/A1		0	337
g14/Z	12	166	503
l3g70/Gout			
l4g150p/H			
g14/A1		0	503
g14/Z	1	83	586
l4g150p/Gout			
cout/x			
g2/A1		0	586
g2/Z	16	390	976
cout/Oout			
MUX_bit0/sel			
g23/S		0	976
g23/Z	1	156	1132
MUX_bit0/pout			
elmmaout[0]		0	1132

Table C.5: Total power of the existing ELMMA for modulo 57343

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l0p1p	1	0.932	389.899	390.832
l0p14p	1	0.928	491.339	492.267
xs_yc_bit0..en[2].xs_scsa	1	0.922	302.453	303.374
xs_yc_bit0..en[6].xs_scsa	1	0.922	264.646	265.568
xs_yc_bit0..en[3].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..en[4].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..en[5].xs_scsa	1	0.922	302.437	303.358
xs_yc_bit0..n[11].xs_scsa	1	0.921	264.630	265.552
xs_yc_bit0..n[12].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[14].xs_scsa	1	0.921	392.924	393.845
xs_yc_bit0..n[15].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..en[1].xs_scsa	1	0.921	392.924	393.845
xs_yc_bit0..en[8].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..en[9].xs_scsa	1	0.921	302.435	303.356
xs_yc_bit0..n[10].xs_scsa	1	0.921	302.432	303.354
xs_yc_bit0..en[0].xs_scsa	1	0.921	250.226	251.148
xs_yc_bit0..en[7].xs_scsa	1	0.921	302.414	303.335
xs_yc_bit0..n[13].xs_scsa	1	0.921	264.610	265.531
l1ps2p	1	0.921	509.040	509.961
p_g_n0_to_..to_n15_level0	1	0.920	319.973	320.892
p_g_n0_to_..to_n15_level0	1	0.919	636.713	637.632
p_g_n0_to_..to_n15_level0	1	0.919	494.619	495.537
p_g_n0_to_..to_n15_level0	1	0.918	645.873	646.791
p_g_n0_to_..to_n15_level0	1	0.918	581.401	582.319
p_g_n0_to_..to_n15_level0	1	0.917	564.845	565.762
p_g_n0_to_..to_n15_level0	1	0.917	454.774	455.691
l2ps15p	1	0.917	456.404	457.321
p_g_n0_to_..to_n15_level0	1	0.916	533.024	533.940
p_g_n0_to_..to_n15_level0	1	0.915	535.125	536.041
p_g_n0_to_..to_n15_level0	1	0.915	448.413	449.328
l0p13p	1	0.915	538.454	539.369
p_g_n0_to_..to_n15_level0	1	0.915	799.270	800.185
p_g_n0_to_..to_n15_level0	1	0.915	689.099	690.014
p_g_n0_to_..to_n15_level0	1	0.914	570.969	571.884
l1ps8	1	0.914	443.922	444.836
p_g_n0_to_..to_n15_level0	1	0.913	624.459	625.372
p_g_n0_to_..to_n15_level0	1	0.913	482.933	483.846
l1ps12	1	0.913	485.558	486.471
l1ps14	1	0.913	561.006	561.918
l1ps14p	1	0.912	412.807	413.719
l1ps2	1	0.912	393.890	394.802
l2ps3p	1	0.912	492.460	493.372
l1ps6	1	0.912	602.245	603.157
l1ps10	1	0.912	671.280	672.192
l1ps4	1	0.912	573.796	574.708
l3ps5p	1	0.911	420.452	421.364
l3ps5	1	0.911	382.090	383.002
l3ps13	1	0.908	598.320	599.228
l2ps15	1	0.908	516.406	517.314
l3ps13p	1	0.908	598.452	599.360
l2ps3	1	0.908	474.239	475.147
l4ps9	1	0.906	445.897	446.803
l4ps9p	1	0.906	448.100	449.006
l2ps11	1	0.904	608.946	609.851
l2ps7	1	0.898	639.272	640.170
MUX_bit0	1	0.631	386.105	386.736
MUX_bit14	1	0.631	364.030	364.660
MUX_bit10	1	0.631	341.314	341.945
MUX_bit6	1	0.630	307.478	308.108
MUX_bit3	1	0.630	344.588	345.217
MUX_bit15	1	0.629	320.229	320.859
MUX_bit11	1	0.629	306.905	307.534
MUX_bit4	1	0.629	325.775	326.405
MUX_bit13	1	0.628	513.086	513.713
MUX_bit12	1	0.623	344.047	344.671
MUX_bit2	1	0.621	347.395	348.016
l0g14p	1	0.620	214.866	215.486
l0g1p	1	0.618	293.128	293.746
l1p1211	1	0.617	388.063	388.680
MUX_bit8	1	0.616	323.490	324.107
l1p87	1	0.615	387.956	388.572
MUX_bit9	1	0.612	322.281	322.893
MUX_bit1	1	0.610	289.579	290.189
MUX_bit5	1	0.608	303.058	303.666
MUX_bit7	1	0.608	320.706	321.313
l1p109	1	0.599	418.861	419.460
l1p65	1	0.598	459.550	460.148
xs_yc_bit0..en[3].yc_scsa	1	0.597	114.636	115.232
xs_yc_bit0..en[4].yc_scsa	1	0.597	229.271	229.868
xs_yc_bit0..en[5].yc_scsa	1	0.597	152.847	153.444
xs_yc_bit0..en[2].yc_scsa	1	0.597	114.629	115.225
xs_yc_bit0..en[6].yc_scsa	1	0.597	152.838	153.435

Continued on Next Page...

Table C.5 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..n[11].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[12].yc_scsa	1	0.597	161.242	161.839
xs_yc_bit0..n[14].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..n[15].yc_scsa	1	0.597	108.708	109.305
xs_yc_bit0..en[1].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[8].yc_scsa	1	0.597	114.633	115.230
xs_yc_bit0..en[9].yc_scsa	1	0.597	191.056	191.652
xs_yc_bit0..n[10].yc_scsa	1	0.596	152.841	153.438
xs_yc_bit0..en[0].yc_scsa	1	0.596	152.847	153.444
xs_yc_bit0..en[7].yc_scsa	1	0.596	76.424	77.020
xs_yc_bit0..n[13].yc_scsa	1	0.596	152.844	153.441
l1p1413	1	0.595	191.390	191.985
l2p75	1	0.593	247.742	248.336
l1p43	1	0.592	224.604	225.196
l2p119	1	0.590	247.515	248.105
p_g_n0_to_..to_n15_level0	1	0.589	164.396	164.985
l1p1413p	1	0.588	370.686	371.274
p_g_n0_to_..to_n15_level0	1	0.587	110.905	111.492
p_g_n0_to_..to_n15_level0	1	0.586	184.765	185.351
p_g_n0_to_..to_n15_level0	1	0.586	134.902	135.487
p_g_n0_to_..to_n15_level0	1	0.585	53.922	54.508
p_g_n0_to_..to_n15_level0	1	0.585	54.049	54.634
p_g_n0_to_..to_n15_level0	1	0.583	148.300	148.883
p_g_n0_to_..to_n15_level0	1	0.575	74.140	74.715
p_g_n0_to_..to_n15_level0	1	0.574	135.665	136.238
p_g_n0_to_..to_n15_level0	1	0.573	81.117	81.690
p_g_n0_to_..to_n15_level0	1	0.572	147.473	148.044
p_g_n0_to_..to_n15_level0	1	0.572 223.317	223.889	
l0g13p	1	0.571 111.212	111.783	
l2p85	1	0.571 223.439	224.009	
l2p129	1	0.569 400.214	400.784	
l3p139p	1	0.569 129.822	130.391	
p_g_n0_to_..to_n15_level0	1	0.567 135.745	136.312	
l3p139	1	0.565 51.946	52.512	
l2p1513p	1	0.563 193.385	193.948	
p_g_n0_to_..to_n15_level0	1	0.559 54.013	54.573	
p_g_n0_to_..to_n15_level0	1	0.559 74.035	74.594	
l3p149	1	0.531 52.059	52.590	
l3p149p	1	0.529 52.293	52.823	
l3p159p	1	0.508 0.000	0.508	
l1g1211	1	0.503 159.007	159.509	
l1g21p	1	0.487 335.696	336.183	
l2g129	1	0.482 535.091	535.573	
l1g87	1	0.482 111.758	112.240	
l1g65	1	0.478 168.887	169.364	
l1g21	1	0.474 276.574	277.048	
l1g1413	1	0.469 189.163	189.632	
l1g109	1	0.464 209.159	209.623	
l2g85	1	0.463 231.785	232.248	
l2g1513p	1	0.456 175.551	176.006	
l1g43	1	0.448 178.102	178.550	
l2g41p	1	0.443 388.114	388.558	
l2g41	1	0.438 346.732	347.170	
l3g81p	1	0.43 301.92	302.35	
l3g81	1	0.43 559.97	560.41	
l1g1413p	1	0.39 245.37	245.75	
cout_prime	1	0.37 602.61	602.98	
l3g159p	1	0.37 192.71	193.08	
l4g151p	1	0.36 189.01	189.37	
ycp_bit14	1	0.32 126.01	126.33	
ycp_bit1	1	0.32 126	126.32	
l3ps6p	2	0.03 393.7	393.74	
l3ps6	2	0.03 354.46	354.49	
l4ps10	2	0.03 438.21	438.24	
l4ps10p	2	0.03 445.25	445.28	
l2ps8	2	0.03 603.84	603.87	
l2ps12	2	0.03 656.29	656.32	
l2ps4p	2	0.03 437.01	437.04	
l3ps14p	2	0.03 495.19	495.22	
l3ps14	2	0.03 576.08	576.11	
l2ps4	2	0.03 395.87	395.91	
l3ps7p	2	0.03 402.66	402.69	
l3ps7	2	0.03 396.79	396.82	
l3ps8p	2	0.03 373.94	373.97	
l3ps8	2	0.03 368.31	368.34	
l4ps11	2	0.03 370.13	370.16	
l4ps11p	2	0.03 375.89	375.92	
l3ps15	2	0.03 484.33	484.36	
l4ps12	2	0.03 401.57	401.6	
l4ps12p	2	0.03 407.83	407.86	
l3ps15p	2	0.03 466.66	466.69	
l4ps13p	2	0.03 427.29	427.32	
l4ps13	2	0.03 419.79	419.82	

Continued on Next Page...

Table C.5 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l4ps14p	2	0.03	363.28	363.31
l4ps15p	2	0.03	362.82	362.85
l4ps14	2	0.03	425.94	425.97
l4ps15	2	0.03	379.08	379.11
xsp_bit0	1	0.02	157.52	157.55
xsp_bit13	1	0.02	179.46	179.48
Total	**197**	**100.078**	**59593.368**	**59693.446**

Table C.6: Total power of the proposed ELMMA for modulo 57343

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
l1ps1p	1	0.932	389.901	390.833
l2ps2p	1	0.929	502.811	503.740
l1ps14p	1	0.928	533.543	534.471
l3ps4p	1	0.923	469.380	470.303
l3ps4	1	0.922	462.832	463.754
xs_yc_bit0..en[2].xs_scsa	1	0.922	493.720	494.642
xs_yc_bit0..en[6].xs_scsa	1	0.922	367.267	368.188
xs_yc_bit0..en[3].xs_scsa	1	0.922	351.274	352.195
xs_yc_bit0..en[4].xs_scsa	1	0.922	493.702	494.623
xs_yc_bit0..en[5].xs_scsa	1	0.922	351.274	352.195
xs_yc_bit0..n[11].xs_scsa	1	0.921	307.363	308.284
xs_yc_bit0..n[12].xs_scsa	1	0.921	502.444	503.366
xs_yc_bit0..n[14].xs_scsa	1	0.921	467.342	468.263
xs_yc_bit0..n[15].xs_scsa	1	0.921	308.404	309.326
xs_yc_bit0..en[1].xs_scsa	1	0.921	401.801	402.723
xs_yc_bit0..en[8].xs_scsa	1	0.921	493.700	494.621
xs_yc_bit0..en[9].xs_scsa	1	0.921	351.272	352.193
xs_yc_bit0..n[10].xs_scsa	1	0.921	419.712	420.633
xs_yc_bit0..en[0].xs_scsa	1	0.921	250.226	251.148
xs_yc_bit0..en[7].xs_scsa	1	0.921	351.251	352.173
xs_yc_bit0..n[13].xs_scsa	1	0.921	307.343	308.264
l3ps12	1	0.921	737.968	738.889
l1ps1	1	0.920	319.987	320.906
l1ps7	1	0.919	443.585	444.504
l4ps8	1	0.919	465.529	466.448
l4ps8p	1	0.919	466.211	467.129
l1ps15	1	0.918	722.181	723.099
l1ps11	1	0.918	404.421	405.339
l1ps13	1	0.915	370.705	371.621
l1ps13p	1	0.915	376.862	377.778
l1ps5	1	0.915	597.411	598.326
l2ps14	1	0.915	532.878	533.792
l1ps9	1	0.915	518.495	519.410
l1ps3	1	0.913	469.443	470.357
l2ps14p	1	0.912	408.739	409.652
l2ps6	1	0.912	581.630	582.542
l2ps2	1	0.911	353.057	353.968
l2ps10	1	0.911	621.672	622.583
MUX_bit0	1	0.631	386.114	386.745
MUX_bit14	1	0.631	364.034	364.665
MUX_bit10	1	0.631	341.318	341.949
MUX_bit6	1	0.630	307.482	308.113
MUX_bit3	1	0.630	344.529	345.159
MUX_bit15	1	0.629	320.234	320.863
MUX_bit11	1	0.629	306.906	307.535
MUX_bit4	1	0.629	325.839	326.468
MUX_bit13	1	0.628	513.092	513.720
MUX_bit12	1	0.623	344.055	344.679
MUX_bit2	1	0.621	347.241	347.862
l1h1514p	1	0.620	290.314	290.933
MUX_bit8	1	0.616	323.475	324.092
l1p98	1	0.614	334.391	335.005
MUX_bit9	1	0.612	322.286	322.898
MUX_bit1	1	0.610	289.583	290.194
MUX_bit5	1	0.608	303.031	303.639
MUX_bit7	1	0.608	320.706	321.314
l1p54	1	0.606	334.422	335.028
l1p32	1	0.606	321.707	322.314
l1p1110	1	0.603	193.704	194.307
l1p1312p	1	0.600	261.544	262.144
l1p1312	1	0.600	335.653	336.252
xs_yc_bit0..en[3].yc_scsa	1	0.597	82.728	83.324
xs_yc_bit0..en[4].yc_scsa	1	0.597	227.052	227.648
xs_yc_bit0..en[5].yc_scsa	1	0.597	110.304	110.900
xs_yc_bit0..en[2].yc_scsa	1	0.597	113.519	114.116
xs_yc_bit0..en[6].yc_scsa	1	0.597	151.359	151.955
xs_yc_bit0..n[11].yc_scsa	1	0.597	82.725	83.322
xs_yc_bit0..n[12].yc_scsa	1	0.597	160.132	160.729
xs_yc_bit0..n[14].yc_scsa	1	0.597	113.524	114.120

Continued on Next Page...

Table C.6 – Continued

Instance	Cells	Leakage Power (nW)	Dyanmic Power (nW)	Total Power (nW)
xs_yc_bit0..n[15].yc_scsa	1	0.597	110.301	110.897
xs_yc_bit0..en[1].yc_scsa	1	0.597	100.640	101.236
xs_yc_bit0..en[8].yc_scsa	1	0.597	113.524	114.120
xs_yc_bit0..en[9].yc_scsa	1	0.597	137.876	138.472
xs_yc_bit0..n[10].yc_scsa	1	0.596	151.362	151.958
xs_yc_bit0..en[0].yc_scsa	1	0.596	151.368	151.964
xs_yc_bit0..en[7].yc_scsa	1	0.596	55.152	55.748
xs_yc_bit0..n[13].yc_scsa	1	0.596	110.301	110.897
l1p76	1	0.595	258.365	258.959
l2p108	1	0.586	217.966	218.552
l2p1412	1	0.579	222.301	222.880
l2p118	1	0.575	315.802	316.376
l3p128	1	0.573	123.749	124.322
l1g1312p	1	0.571	110.723	111.294
l2p64	1	0.571	217.917	218.488
l2p1412p	1	0.566	155.783	156.349
l2p74	1	0.555	161.156	161.710
l3p138p	1	0.542	78.306	78.848
l3p138	1	0.534	26.258	26.792
l3p148	1	0.528	26.285	26.813
l3p148p	1	0.522	52.827	53.349
l3p158p	1	0.508	0.000	0.508
l1g10p	1	0.485	262.731	263.216
l1g1514	1	0.470	114.367	114.837
l1g10	1	0.467	168.105	168.572
l1g1110	1	0.465	135.720	136.185
l1g76	1	0.464	110.549	111.013
l1g98	1	0.459	207.391	207.850
l1g54	1	0.450	204.833	205.283
l1g1312	1	0.447	225.943	226.390
l1g32	1	0.432	145.301	145.734
l2g118	1	0.428	506.695	507.124
l2g30p	1	0.425	342.814	343.239
l2g30	1	0.416	346.290	346.706
l2g74	1	0.414	233.348	233.761
l3g70p	1	0.407	338.910	339.318
l3g70	1	0.406	595.465	595.871
l2g1512p	1	0.406	204.220	204.626
cout	1	0.384	599.153	599.536
l3g158p	1	0.340	193.810	194.149
l4g150p	1	0.325 189.021	189.346	
ycp_bit13	1	0.318 126.009	126.327	
ycp_bit0	1	0.317 124.499	124.817	
l2ps3p	2	0.034 462.598	462.632	
l4ps9	2	0.034 430.329	430.363	
l4ps9p	2	0.034 437.273	437.306	
l3ps5p	2	0.033 420.570	420.604	
l3ps5	2	0.033 381.634	381.667	
l3ps13p	2	0.033 573.689	573.723	
l3ps13	2	0.033 571.178	571.211	
l2ps3	2	0.033 440.870	440.903	
l3ps6p	2	0.033 394.448	394.480	
l3ps6	2	0.033 355.190	355.222	
l4ps10	2	0.032 438.785	438.818	
l4ps10p	2	0.032 445.761	445.794	
l2ps15	2	0.032 488.053	488.085	
l2ps11	2	0.032 592.349	592.381	
l3ps7p	2	0.032 399.255	399.287	
l2ps7	2	0.032 623.866	623.898	
l3ps7	2	0.032 393.605	393.637	
l3ps14p	2	0.032 491.450	491.482	
l3ps14	2	0.032 576.359	576.391	
l3ps15	2	0.032 480.047	480.078	
l3ps15p	2	0.031 447.442	447.474	
l4ps11	2	0.031 367.134	367.165	
l4ps11p	2	0.031 372.669	372.700	
l4ps12	2	0.03 404.54	404.57	
l4ps12p	2	0.03 410.96	410.99	
l4ps13p	2	0.03 422.66	422.69	
l4ps13	2	0.03 415.07	415.11	
l4ps14p	2	0.03 363.27	363.3	
l4ps14	2	0.03 425.92	425.95	
l4ps15	2	0.03 379.06	379.09	
l4ps15p	2	0.03 348.75	348.78	
xsp_bit0	1	0.02 157.52	157.55	
xsp_bit13	1	0.02 215.92	215.94	
Total	**181**	**77.565**	**51407.319**	**51484.884**