A graph based approach for formal verification of Galois field multipliers

by

Anil Kumar Boga

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Data transfer between devices has increased rapidly with improvements in technology and the internet. To protect data from hackers, data is encrypted using methods of cryptography. To make the process of encryption faster, these circuits are often implemented in hardware. A bug in these circuits compromise the security of these systems. Cryptography circuits are becoming large and complex due to the increase in the importance of security and computation power available to the hacker. Hence verification of these circuits is of utmost importance. Time and resources taken by conventional methods increase exponentially with the size and complexity of circuits. Formal verification has the potential to improve the verification process by providing better than exponential complexity for some systems.

Conventional formal verification methods do not perform well on cryptography circuits as they are "XOR" rich circuits. Cryptography circuits often consist of Galois field circuits. Galois field circuits are also widely used in various fields like communication, security and signal processing. There are two main operations in Galois field namely addition and multiplication. Addition is simply bitwise "XOR" of operands. Multiplication is more complicated. Mastrovito, Montgomery and Karatsuba multipliers are optimized algorithms for multiplication. In this thesis, we developed novel methods for the formal verification of Galois field multipliers.

Equivalence verification of Galois field circuits becomes challenging as the size of inputs increases because the asymptotic worst-case complexity is exponential. The previous best-known method reduces the time and memory to some extent by using parallelism. In this thesis, a novel formal verification method is developed which provides a range of $4\times$–$256\times$ speedup when compared to the previously best-known method. We developed novel data structures and algorithms based on using the algebraic normal form as a canonical graph-based representation of Boolean functions. We have developed various normalization methods for our data structures. Experiments were performed on bit-level synthesized Mastrovito, Montgomery and Karatsuba multipliers.

KeyWords: cryptography, Galois field, formal verification, equivalence verification, algebraic normal form.

Acknowledgements

I would like to thank my supervisor, Dr. Mark Aagaard, for his support and guidance throughout my graduate studies. Through his guidance, I have learned a lot of things in formal verification and software engineering. I enjoyed working with him. I would like to thank Nusa Zidaric for the discussion on irreducible polynomials. I would also like to thank Bo Yang for helping me with synthesis tools.

Dedication

This is dedicated to my family and friends.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital circuits are found in products that we use every day. Following Moore's law [9], digital circuits are becoming more complex day by day. Verification of such complex digital circuits is very challenging. Even a small bug in the design of digital circuits can incur a huge extra cost. For example, a bug in the Intel Pentium floating point divide cost around 500 million USD (now in billions) in 1994 [1]. Signal processing, communication and security-related hardware systems involve cryptography circuits. Cryptography circuits are becoming large and complex due to the increase in the importance of security and high computation power available to the hacker. A bug in cryptography circuits [3], can result in leakage of security key which comprises the security of these systems. Hence to protect sensitive data from hackers, verification of cryptography circuits is of utmost importance.

Chip fabrication has different stages. It starts with a specification which is manually developed into Register Transfer Level (RTL) using hardware description languages like VHDL and Verilog. Synthesis tools process RTL to produce an optimized netlist and map to physical gates. The process in synthesis tools has many stages. As a design flows through different stages, we need to verify the design at each stage. Design can be verified by simulation-based methods or formal verification. Time and computing resources taken by simulation-based verification techniques will increase exponentially with an increase in the complexity of the design. Alternatively, Formal verification has the potential to improve the verification process by providing better than exponential complexity for some systems. Formal verification can be of two types. They are property checking and equivalence checking.

Property checking verifies whether the design satisfies a collection of properties, where each property describes part of the desired behaviour of system. Equivalence checking ver-

ifies whether two designs, namely the specification and implementation, are functionally equivalent. The implementation model is an optimized version of the specification model. Two prominent techniques, namely Boolean satisfiability (SAT) and binary decision diagrams (BDD) can be used for equivalence verification. BDDs are canonical diagrams that can be formed from the truth table of Boolean functions. If two functions have the same BDD then they are equivalent. SAT solvers try to find an input for which two functions are not equivalent. If no such input exists, then the two functions are equivalent.

Conventional formal verification does not perform well on cryptography circuits. Cryptography circuits handle sensitive data and to protect this data, cryptography circuits are designed to scramble the data in a way that makes it difficult for hackers to steal data. This feature of cryptography circuits makes them complex and "XOR" rich when compared to normal arithmetic circuits. Verification of "XOR" rich circuits is challenging for conventional formal verification tools when compared to circuits containing "AND" and "OR" gates.

Cryptography circuits often consist of Galois field circuits. Galois field circuits is also found in error checking [17] and signal coding [22]. There are two main operations in Galois field. They are addition and multiplication. Addition is simply bitwise "XOR" of operands. Multiplication is more complicated. Karatsuba, Montgomery and Mastrovito multipliers are optimized algorithms for multiplication. The current state-of-the-art method for formal verification of Galois field multipliers was proposed by Yu [25] and uses parallel extraction to reduce the time and memory to some extent.

Novel methods for the formal verification of Galois field multipliers are developed in this thesis. Our method involves the building of a graph from the netlist, obtaining an expression for the output bit by traversing the graph. Our methods provide a range of 4×–256× speedup when compared to Yu's [25] method. Any Boolean function can be represented by algebraic normal form. If two Boolean functions have the same algebraic normal form, then they are equivalent. In this thesis, we have developed novel algorithms and data structures for algebraic normal form, so two Boolean functions are equivalent if they have the same data structure.

1.1 Contributions

In this thesis, we have developed novel methods for the formal verification of Galois field multipliers. Our methods take the netlist of the specification and implementation as input and outputs whether the two models are equivalent. If the models are not equivalent, our

method gives the output bits which are not equivalent. A graph structure is developed which is built from the netlist and traversed in a depth-first manner to obtain the algebraic normal form expression of the output bit. We first developed a string-based approach in which expressions are represented as a list of cubes and a cube is represented as a string. Conventionally, integers are used to represent signals and list of integers represents a cube. But we are using strings over integers because strings can be easily used as a key in a hash map, whereas it is difficult to hash list of integers. Since strings are used to represent expressions, the same sub-expression may exist many times in different expressions. This results in consuming a lot of memory and increases the time to compute expressions. To overcome the drawbacks of the string-based approach we have developed a graph-based approach. In a graph-based approach, we developed novel data structures and algorithms based on using the algebraic normal form as a canonical graph-based representation of Boolean functions. We have developed various normalization methods for the our data structures. The current state-of-art method for formally verifying Galois field multipliers was proposed by Yu [25] which uses function extraction methods in a parallel fashion from outputs to primary inputs. In Yu's [25] methods, nodes which are common to output nodes are visited multiple times. In our methods, during traversal, all of the nodes are traversed only once which reduces time when compared to Yu's [25] method. The efficiency of the our methods is illustrated through experiments on bit-level synthesized Mastrovito, Montgomery and Karatsuba multipliers. The developed methods achieves a range of 4 – 256× speedup when compared to Yu's [25] method.

1.2 Thesis Organisation

The remainder of the thesis is organized as follows. Chapter 2 provides background information on various formal verification techniques and previous work related to the verification of finite field multipliers. Chapter 3 describes our string-based approach. Chapter 4 describes our graph-based approach. Chapter 5 presents the results of our algorithms and comparison to Yu's [25] method. Chapter 6 concludes the thesis and presents ideas for future work.

Chapter 2

Background and Related Work

This chapter gives an introduction to various topics which are useful to understand this thesis. The background section will give a brief overview of a finite field, finite field multipliers and formal verification methods. The related work section explores past work regarding formal verification methods for finite field multipliers. The significance of the common nodes section presents the motivation for this thesis.

2.1 Background

2.1.1 Finite Field Multipliers

The finite field with p elements $(0, 1, 2, ..., p-1)$ is denoted by $GF(p)$ where p is a prime number. Finite fields are also called Galois fields in honor of the founder of finite field theory, *Évariste Galois*. The arithmetic operation (addition, subtraction, multiplication) in the Galois field uses the usual operation on integers, followed by reduction modulo p. An extended Galois field of $GF(p)$ which is denoted by $GF(p^n)$ has p^n elements where p is a prime number. Elements in $GF(p^n)$ are polynomials of degree strictly less than n and coefficients of these polynomials are taken from the $GF(p)$ field. To construct and define properties of the extended field, the concept of the irreducible polynomial is required.

Definition 1 *A polynomial, $P(x)$, is irreducible over a field F if it cannot be expressed as the product of two or more polynomials whose coefficients are in F.*

The irreducible polynomial will be of degree n to reduce the polynomial to a degree less than n. There exist many irreducible polynomials in a field but finding one is not a trivial task.

Examples for valid irreducible polynomials in $GF(2^4)$

1. $x^4 + x + 1$
2. $x^4 + x^3 + 1$

Examples for invalid irreducible polynomial in $GF(2^4)$

1. $x^4 + x^2 + 1$: because it can be written as $(x^2 + x + 1)(x^2 + x + 1)$
2. $x^4 + x^3 + x^2$: because it can be written as $x^2(x^2 + x + 1)$

In $GF(p^n)$, if $p = 2$, then it is called a binary extension field and coefficients of the terms in the polynomial are either 0 or 1. For example, an element in $GF(2^4)$ is represented as follows

$$B(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0, a_i \in GF(2) \tag{2.1}$$

$B(x)$ can also be represented as a bit vector as (a_3, a_2, a_1, a_0).

Addition in extended Galois field is bitwise "XOR" of operands. It can be defined as

Definition 2 *Let $A(x)$, $B(x) \in GF(2^n)$. The addition of the two elements $A(x)$ and $B(x)$ is computed as*

$$C(x) = A(x) + B(x), \quad c_i = a_i + b_i \mod 2 \tag{2.2}$$

Multiplication in $GF(2^n)$ is a costly operation as it involves polynomial multiplication which is of $O(n^2)$ where n is the degree of the polynomial and followed by reduction using irreducible polynomial.

Definition 3 *Let $A(x)$, $B(x) \in GF(2^n)$ and let $P(x) \in GF(2^n)$ be an irreducible polynomial. Multiplication of the two elements $A(x)$ and $B(x)$ is computed as*

$$C(x) = A(x) \cdot B(x) \mod P(x) \tag{2.3}$$

Example 2.1

Let $A(x), B(x) \in GF(2^3)$ and $A(x) = x^2+x$, $B(x) = x^2+1$. Let the irreducible polynomial in $GF(2^3)$ be $P(x) = x^3 + x + 1$. Find $C(x) = A(x) \cdot B(x) \mod (P(x))$

$$\begin{aligned} C(x) &= A(x) \cdot B(x) \mod (P(x)) \\ &= (x^2 + x)(x^2 + 1) \mod (P(x)) \\ &= (x^4 + x^3 + x^2 + x) \mod (P(x)) \end{aligned} \tag{2.4}$$

$$\begin{array}{r|l} & x + 1 \\ \hline x^3 + x + 1 & x^4 + x^3 + x^2 + x \\ & -x^4 \quad\quad -x^2 - x \\ \hline & \quad x^3 \\ & -x^3 \quad\quad -x - 1 \\ \hline & \quad\quad -x - 1 \end{array}$$

The remainder is $x + 1$ because $-1 \mod 2 = 1$ and therefore $C(x) = x + 1$

Multiplication in a finite field involves two steps, namely polynomial multiplication and reduction using an irreducible polynomial. After polynomial multiplication, the resultant polynomial is of degree at most $2n - 2$ in the field of $GF(2^n)$. The time taken for the modulo operation depends on the irreducible polynomial. For example, sparse irreducible polynomial takes less time than dense. Karatsuba, Montgomery and Mastrovito are optimized algorithms for multiplication in $GF(2^n)$. In this thesis, a new method for the formal verification of these algorithms is developed.

Classic Multiplier

The classic multiplier computes the output in two steps: multiplication and reduction.

Multiplication

Let $D(x)$ be the product of two polynomials $A(x)$ and $B(x)$, where $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ then

$$D(x) = A(x) \cdot B(x) \tag{2.5}$$

Then $D(x)$ is a polynomial of degree $2n-2$, which can be written in the form of a matrix as

$$\begin{bmatrix} d_0 \\ d_1 \\ . \\ . \\ . \\ d_{2n-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & . & . & . & 0 \\ a_1 & a_0 & 0 & . & . & . & 0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ a_{n-1} & a_{n-2} & . & . & . & . & a_0 \\ 0 & a_{n-1} & . & . & . & . & a_1 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 0 & 0 & . & . & . & . & a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ . \\ . \\ . \\ b_{n-2} \\ b_{n-1} \end{bmatrix} \tag{2.6}$$

$D(x)$ can be represented as:

$$d_k = \begin{cases} \sum_{i=0}^{k} a_i b_{k-i} & \text{where } k = 0, ..., n-1 \\ \sum_{i=k}^{2n-2} a_{k-i+(n-1)} b_{i-(n-1)} & \text{where } k = n, ..., 2n-2 \end{cases}$$

From the above expression we can observe that to compute $D(x)$ requires n^2 "AND" gates and $(n-1)^2$ "XOR" gates.

Reduction

Let $C(x)$ be the reduced polynomial for $D(x)$ then

$$C(x) = D(x) \mod (P(x)) \tag{2.7}$$

Let R be a reduction matrix of size $(n, n-1)$, then $C(x)$ can be written as:

$$\begin{bmatrix} c_0 \\ c_1 \\ . \\ . \\ . \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & . & r_{0,0} & . & r_{0,n-2} \\ 0 & 1 & 0 & . & r_{1,0} & . & r_{1,n-2} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 0 & . & 1 & . & r_{n-1,0} & . & r_{n-1,n-2} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ . \\ . \\ . \\ d_{2n-3} \\ d_{2n-2} \end{bmatrix} \tag{2.8}$$

Elements of the reduction matrix can be computed from $P(x)$ as:

$$r_{j,i} = \begin{cases} p_j & \text{where } i = 0,\ j = 0, ..., n-1 \\ r_{j-1,i-1} + r_{n-1,i-1} r_{j,0} & \text{where } j = 0, ., n-1,\ i = 1, ..., n-2 \end{cases} \tag{2.9}$$

Algorithm 1 is the algorithm for the classic multiplier

Algorithm 1 Classic multiplier

Input: $A(x)$, $B(x)$ are two polynomials in $GF(2^n)$ and $P(x)$ is an irreducible polynomial
Output: return $C(x)$ which is product of $A(x)$ and $B(x)$

```
D = poly_multiplication(A,B)
R = reduction_matrix(P)
for j in 0 to n-1 do
    c(j) = d(j)
end for
for j in 0 to n-1 do
    for i in 0 to n-2 do
        c(j) = xor(c(j), and(R(j,i),d(n+i)))
    end for
end for
return c
```

The functions poly_multiplication and reduction_matrix return $D(x)$ and R respectively

Example 2.2

Taking the same parameters as in example 2.1, $D(x)$ can be calculated from equation 2.6:

$$D(x) = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{2.10}$$

R can be calculated from equation 2.9:

$$R = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{2.11}$$

Karatsuba Multiplier

Karatsuba multiplier is a recursive method to efficiently compute the multiplication of two finite field polynomials. It is known that to multiply two polynomials of degree $n-1$ requires n^2 multiplications and $(n-1)^2$ additions. But Karatsuba multiplier requires fewer multiplication and additions for $n = 2^t$ where t is an integer.

Let $A(x)$, $B(x)$ be two polynomials of degree $n-1$ in $GF(2^n)$. They can be represented as:

$$\begin{aligned} A(x) &= x^{n/2}(x^{n/2-1}a_{n-1} + ... + a_{n/2}) + (x^{n/2-1}a_{n/2-1} + ... + a_0) \\ &= x^{n/2}A_H + A_L \\ B(x) &= x^{n/2}B_H + B_L \end{aligned} \tag{2.12}$$

where
$A_H = x^{n/2-1}a_{n-1} + ... + a_{n/2}$,
$A_L = x^{n/2-1}a_{n/2-1} + ... + a_0$,
$B_H = x^{n/2-1}b_{n-1} + ... + b_{n/2}$,
$B_L = x^{n/2-1}b_{n/2-1} + ... + b_0$

Now $D(x)$ can be represented as:

$$\begin{aligned} D(x) &= A(x).B(x) \\ &= x^n A_H B_H + x^{n/2}(A_H B_L + A_L B_H) + A_L B_L \end{aligned} \tag{2.13}$$

Let

$$\begin{aligned} M_0(x) &= A_L(x)B_L(x) \\ M_1(x) &= (A_L(x) + A_H(x))(B_L(x) + B_H(x)) \\ M_2(x) &= A_H(x)B_H(x) \end{aligned} \tag{2.14}$$

Now $D(x)$ can be written in terms of $M_0(x)$, $M_1(x)$ and $M_2(x)$:

$$D(x) = x^n M_2(x) + x^{n/2}(M_1(x) + M_0(x) + M_2(x)) + M_0(x) \tag{2.15}$$

This algorithm is recursively applied to compute $M_0(x)$, $M_1(x)$ and $M_2(x)$ and it takes $\log n$ steps to compute $D(x)$. After obtaining $D(x)$ we can use the reduction matrix of a classic multiplier to obtain $C(x)$. We can observe from equation 2.13, we need 4 multiplications to compute $D(x)$, but equation 2.13 can be reduced to equation 2.15 and it requires only 3 multiplications of polynomials of size $n/2$. This reduces the total number of multiplications required to $n^{log_2^3} \approx n^{1.585}$ [23] when compared to n^2 multiplications in a classic multiplier.

Montgomery Multiplier

Montgomery multiplication was first proposed for integers but later extended to the finite field. Let $A(x)$ and $B(x)$ be two polynomials and let $P(x)$ be an irreducible polynomial in $GF(2^n)$. Instead of computing 2.16, Montgomery multiplier computes 2.17 where $R(x)$ is a polynomial such that $gcd(R(x), P(x)) = 1$.

$$C(x) = A(x) \cdot B(x) \mod (P(x)) \tag{2.16}$$

$$C(x) = A(x) \cdot B(x) \cdot R^{-1}(x) \mod (P(x)) \tag{2.17}$$

$C(x)$ can be computed in the following ways

Method 1:

If $A(x)$ and $B(x)$ are inputs, we can transform them into Montgomery form AR and BR which can be computed using a classic multiplier, then their Montgomery product will be ABR. Now again compute Montgomery product taking ABR, 1 as inputs, then the result will be AB which is required.

$$
\begin{aligned}
AR(x) &= A(x) \cdot R(x) \\
BR(x) &= B(x) \cdot R(x) \\
ABR(x) &= montg\ (AR(x), BR(x)) \\
&= AR(x) \cdot BR(x) \cdot R^{-1}(x) \\
&= A(x) \cdot R(x) \cdot B(x) \cdot R(x) \cdot R^{-1}(x) \\
&= A(x) \cdot B(x) \cdot R(x) \\
AB(x) &= montg\ (ABR(x), 1) \\
&= A(x) \cdot B(x) \cdot R(x) \cdot R^{-1}(x) \\
&= A(x) \cdot B(x)
\end{aligned}
\tag{2.18}
$$

Method 1 is used to compute expressions that contain many multiplications. The inputs are converted into the Montgomery form, then multiplications are done with the Montgomery algorithm and the final result is converted back into the conventional representation.

Method 2:

First obtain the Montgomery product of A, B which is ABR^{-1}. Now transform into the conventional representation by using a classic multiplier with inputs ABR^{-1} and R. Because we need to verify against a specification of a conventional finite field multiplication, we will be using this method in the thesis for experiments.

Following is the bit-level algorithm for the Montgomery product when $R(x) = x^n$

Algorithm 2 Montgomery multiplier

Input: $A(x)$, $B(x)$ are two polynomials in $GF(2^n)$ and $P(x)$ is an irreducible polynomial
Output: return $C(x) = A(x) \cdot B(x)x^{-n} \mod (P(x))$

1: $C(x) = 0$
2: **for** j in 0 to n-1 **do**
3: $\quad C(x) = C(x) + a_i b(x)$
4: $\quad C(x) = C(x) + c_0 p(x)$
5: $\quad C(x) = C(x)/x$
6: **end for**
7: return C

Mastrovito Multiplier

Multiplication of two finite field polynomials can be computed in two steps like classic multiplier but it can also be computed in a single step using the Mastrovito product matrix. Equation 2.16 can be modified as 2.19 where Z is the Mastrovito product matrix of size (n, n).

$$C(x) = Z \cdot B(x) \tag{2.19}$$

Z can be computed as:

$$z_{j,i} = \begin{cases} a_i & \text{where } j = 0, i = 0...n-1 \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i} a_{n-1-t} & \text{where } j, i = 0, ..n-1,\ j \neq 0 \end{cases}$$

where $u(x)$ is a step function that is 1 when $x > 0$ and 0 when $x < 0$. The Q matrix is required for computation of the Z matrix and the Q matrix can be computed from P as:

$$q_{i,j} = \begin{cases} q_{i-1,n-1} & \text{where } i = 1..n-2\ , j = 0 \\ q_{i-1,j-1} + q_{i-1,n-1} q_{0,j} & \text{where } j = 1, ...n-1, i = 0, ..n-2 \end{cases}$$

where $q_{0,j} = p_j$. We can observe that the Q matrix is very similar to the reduction matrix in a classic multiplier. In fact $R = Q^T$. The Mastrovito multiplier reduces the number of gates required when compared to the classic multiplier by combining the multiplication and reduction steps into one step. The Mastrovito multiplier of n bit inputs with an irreducible polynomial of the form $x^n + x^m + 1$ takes n^2 "AND" gates and $n^2 - 1$ "XOR" gates [20].

2.1.2 Algebraic Normal Form

Every Boolean function can be represented in terms of the sum of products in the following way

$$\begin{aligned} f(x_1, x_2, ...x_n) = a_0 \oplus \\ & a_1 x_1 \oplus a_2 x_2 \oplus ..... \oplus a_n x_n \oplus \\ & a_{1,2} x_1 x_2 \oplus ..... \oplus a_{n-1,n} x_{n-1} x_n \oplus \\ & .....\oplus \\ & a_{1,2,3,...n} x_1 ... x_n \end{aligned} \tag{2.20}$$

where $a_0, a_1, ...a_{1,2,....n} \in \{0,1\}$ are constant coefficients and
$x_1, ...x_n$ are Boolean variables.

This representation of f is called an algebraic normal form. Algebraic normal form expressions do not contain inverters and "OR" gates. They only contain "XOR" and "AND" gates. The algebraic normal form is canonical and so if two functions have the same algebraic normal form, then they are equivalent.

Examples of algebraic normal form

1. $x \oplus y \oplus xy$
2. $1 \oplus y \oplus xyz$

Examples which are not algebraic normal form

1. $x \oplus x \oplus xy$: is not in algebraic normal form because it simplifies to xy
2. $x \odot (y \oplus z)$: is not in algebraic normal form because it is not in the sum of products form.

Converting into algebraic normal form

Inverter and "OR" gates can be converted into "AND" and "XOR" using the following equations

$$\begin{aligned} \neg a &= 1 \oplus a \\ a \vee b &= a \oplus b \oplus ab \end{aligned} \tag{2.21}$$

Example

Following are examples for converting Boolean expression into algebraic normal form

1. This is an example of a Boolean expression containing "OR" gate and an inverter.

$$\begin{aligned} f &= x \vee (y \odot \neg z) \\ &= x \vee (y(1 \oplus z)) \\ &= x \vee (y \oplus yz) \\ &= x \oplus y \oplus yz \oplus x(y \oplus yz) \\ &= x \oplus y \oplus yz \oplus xy \oplus xyz \end{aligned} \tag{2.22}$$

2. This is an example of a Boolean expression containing an inverter.

$$\begin{aligned} f &= \neg(1 \oplus x \oplus y) \\ &= 1 \oplus 1 \oplus x \oplus y \\ &= x \oplus y \end{aligned} \tag{2.23}$$

3. "AND" can be represented as an operation over Boolean expressions and as a product of two expressions in algebraic normal form. Following is the example.

$$\begin{aligned} f &= (1 \oplus x) \odot (1 \oplus x \oplus y) \\ &= 1(1 \oplus x \oplus y) \oplus x(1 \oplus x \oplus y) \\ &= 1 \oplus x \oplus y \oplus x \oplus x \oplus xy \\ &= 1 \oplus x \oplus y \oplus xy \end{aligned} \tag{2.24}$$

2.1.3 Canonical Diagrams

BDD (Binary Decision Diagrams) which are canonical Directed Acyclic Graph (DAG) representations of Boolean functions were first proposed in [4]. A BDD can be obtained in the following way. From the truth table of a Boolean function, we can form a binary decision tree. From a binary decision tree we can form a binary decision diagram by combining equivalent nodes. From a binary decision diagram, we can form an ordered BDD by enforcing some order of Boolean variables. From an ordered BDD, we can form a reduced ordered BDD (ROBDD), which is commonly called BDD, by merging equivalent leaves and isomorphic nodes. A BDD is a canonical diagram and hence if two Boolean functions have the same BDD, then they are equivalent. Figure 2.1 is the BDD for $f = (a + b) \odot c$ and figure 2.2 is the ROBDD for f.

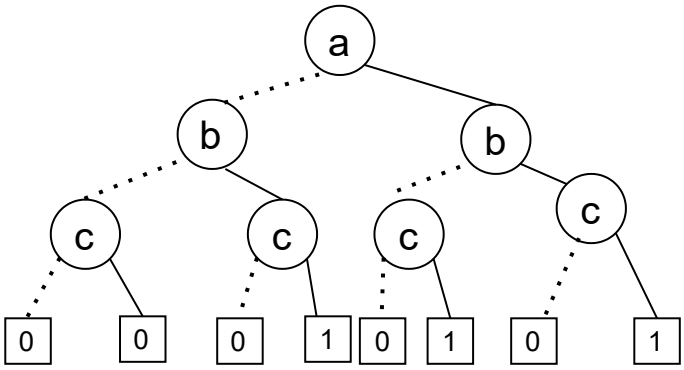


Figure 2.1: BDD for $f = (a + b) \odot c$

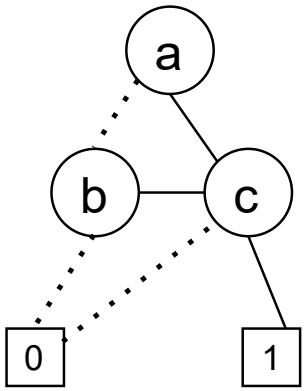


Figure 2.2: ROBDD for $f = (a + b) * c$

BDDs are extensively used in logic synthesis and verification of small circuits but their application to very large arithmetic circuits is limited as they suffer from high memory consumption.

In the graph-based approach which will be introduced in chapter 4, we have developed a data structure that represents algebraic normal form. It is a binary tree and canonical. If two functions point to the same root node of a tree, then they are equivalent. The ordering concept from BDD was taken and applied to our data structure. In the our data structure, let a, b be n bit primary inputs, the following order is considered $a_1 < a_2 < ..... < a_n < b_1 < .... < b_n$.

2.1.4 SAT Solvers

The Boolean satisfiability problem, which is abbreviated as SAT, is the problem of determining if there exists a situation for a given Boolean formula that evaluates to true. SAT is an NP-complete problem and many heuristics were proposed to solve this. Most of the heuristics were based on the Davis, Putnam, Logemann, Loveland (DPLL) [5] algorithm.

Verification of two arithmetic circuits can be modeled as a decision problem and thus SAT solvers can be used. Some of the SAT solvers are ABC [6] and Minisat [21]. SMT solvers are an extension to SAT solvers which can handle integer variables instead of Boolean variables and inequalities. Some SMT solvers which can be used for verification are Boolector [15], Z3 [14], CVC [7]. However, SAT and SMT solvers do not perform well for verification of large finite field arithmetic circuits, as demonstrated in [24], [11].

2.2 Related Work

V. Bakoev et al have investigated techniques for computing the algebraic normal forms of a set of Boolean functions with up to 24 variables [2, 16]. Their work is focused on batch processing a large number of functions with a small number of variables (e.g. 3800 functions with 24 variables to 3.9 million functions with 14 variables). In contrast, our work is aimed at computing the algebraic normal form of one or two functions with a large number of variables (up to 1142 in the current experiments).

Computer algebra-based methods were previously proposed for the verification of finite field arithmetic circuits. Lv et.al [11] first proposed computer algebra-based methods for formal verification of finite field multipliers. Let B be the set of polynomials for the implementation and let the specification polynomial be F. The implementation model satisfies the specification if F is reduced to zero by the Gröbner basis of B. The methods proposed in [11] are feasible up to 163-bit multipliers before time and memory limits are exceeded. The authors in [8] use Gaussian elimination instead of polynomial division to speed up the process.

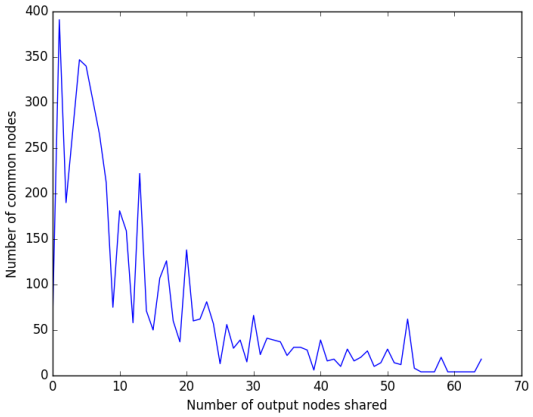
Gröbner basis reduction has been successful in verification of large integer multipliers, previously proposed techniques suffer from exponential blow-up of polynomials. Sayed-Ahmed et. al in [18] identified reasons for this problem and proposed a new method that uses structural information of circuits to remove terms that evaluate to zero before their blow-up. This approach works up to 128-bit multipliers. Mahzoon et al. in [12] identified that the exponential blow-up of polynomials is due to redundant monomials,

also known as vanishing monomials. They proposed a new method, called SCA-verifier PolyCleaner, which presents a new theory of the origin of vanishing monomials and how they can be handled to prevent explosion during backward rewriting. In [13], Mahzoon et al. presented further optimizations which identifies atomic blocks of arithmetic circuits using reverse engineering techniques and use these atomic blocks to identify all sources of vanishing monomials. This method works up to 1024-bit multipliers.

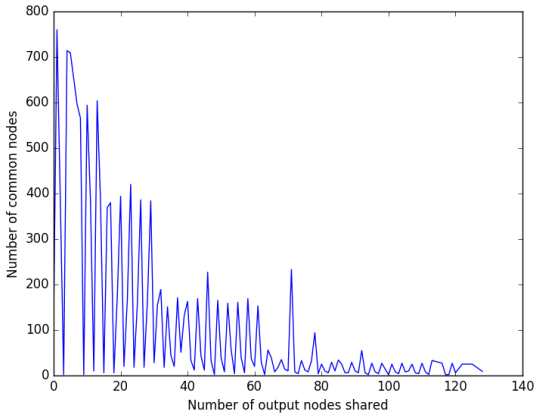
The recently proposed method by Yu [25] uses the function extraction method proposed in [24]. Yu's method exploits the inherent parallel structure of Galois field circuits and computes the expression for output bit in terms of primary inputs by backward rewriting (i.e. from outputs to primary inputs) in a parallel fashion using threads. The output bits are distributed between the threads equally. Backward rewriting involves substitution and elimination of terms. But in this process, nodes that are common to two or more output bits are computed multiple times. In the thesis, we developed a novel algorithm to overcome these issues and compared our results to Yu's [25] method.

2.3 Significance of Common Nodes

Let the path from output node n_1 to primary inputs be p_1 and let the path for n_2 be p_2. The nodes which are present in both paths p_1 and p_2 are called common nodes. An optimized circuit consists of large number of common nodes as a synthesis tool tries to reduce the number of gates. In finite field multipliers like Karatsuba, there are a large number of common nodes as it uses the same unit blocks to compute multiple outputs. We can observe from Figures 2.3a and 2.3b, the percentage of internal nodes that drive only one output is a small fraction of the total number of nodes. In Yu's [25] method, the expression for each output bit is computed by traversing from the output bits to the primary inputs in a parallel fashion using threads. When traversing in a parallel fashion, the expression for common nodes is computed multiple times. In our work, the traversal of graphs is designed such that each of the nodes is visited only once, which significantly reduces time.



(a) Karatsuba multiplier of 64 bit size



(b) Karatsuba multiplier of 128 bit size

Figure 2.3: Common nodes VS shared output nodes

Chapter 3

String Based Approach

In this chapter, we will introduce our first algorithm for the formal verification of Galois field multipliers. There are three main steps involved in the algorithm namely building a graph from the netlist, rewriting and verification. Our algorithm takes netlists of specification and implementation and builds a graph from it. To compute algebraic normal form expressions for output bits, the graph is traversed from outputs to primary inputs in depth-first traversal. We will introduce a data structure for expressions and various methods for normalizing expressions. We will also introduce two verification methods for the computed expressions. At last, we will identify drawbacks of the algorithm.

3.1 Graph

The graph formed from the netlist is a **DAG** (Directed Acyclic Graph) because the netlist of finite multiplier consists of combinational circuits only. The graph which is formed represents the circuit in the following way. Nodes in the circuit represent gates in the netlist and edges represent a signal. Since the node represents the gate in the circuit it should contain information about fanin (i.e. sources) and type of gate and fanout of gate which are destinations of the gate.

The graph can be built from the netlist in the following way. For every gate in netlist make a node. Edges between two nodes are formed when the output signal of a node matches with an input signal of another node. To identify this, store the output signal and node in a map. When making a new node check for the input signal in the map, if it is present, append the current node to the destinations of the node in the map. There may

be a situation where inputs are not currently defined but are defined later in the netlist. To handle this situation, store the input and node in another map, say an input map, when building a new node check if the output of the current node is present in the input map and if yes, form an edge between the current node and node in input map.

Figure 3.1 is an example graph that can be formed from the netlist. While building a graph, one can identify nodes whose outputs are primary output bits and these nodes can be used as source nodes while traversing the graph in depth-first manner.

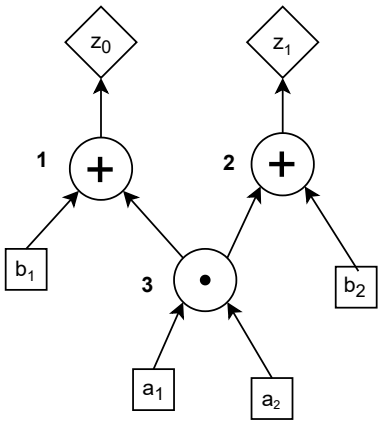


Figure 3.1: Graph

3.2 Expression

Rewriting uses normalization to convert circuit into expressions at every node in the graph. Expressions are used to represent the algebraic normal form. In the developed algorithm, expression is represented as list of strings. The expression data structure is constructed in the following way. A signal is represented by string and a cube which is product of two or more signals is represented by concatenating signals by '$*$'. Signals in each cube which are strings are sorted in ascending order. Let a, b be n bit primary inputs, following order is considered $a_1 < a_2 < ..... < a_n < b_1 < .... < b_n$.

Conventionally, integers are used to represent a signal, list of integers represent a cube and list of list of integers represent a expression. But using strings have a advantage as it is easier to hash strings than list of integers.

Examples of valid expressions

1. [“$a_1 * b_1$”, “a_2”, “$a_3 * b_1$”].
2. [“$a_2 * b_2$”, “$b_2 * b_3$”].

Examples of invalid expressions

1. [“$a_1 * a_1$”, “a_2”] : not valid because it can be simplified to [“a_1”, “a_2”].
2. [“a_2”, “a_2”, “b_2”]: not valid because the equivalent expression is $a_2 \oplus a_2 \oplus b_2$ and it can be simplified to b_2.
3. [“$b_1 * a_1 * b_2$”, “b_3”]: not valid because in cube “$b_1 * a_1 * b_2$” signal “a_1” should come before signal “b_1” as signals in a cube are sorted in ascending order.

3.3 Normalization

Rewriting uses normalization at every node to compute symbolic expression. The objective of rewriting is to traverse the graph to compute expression at every node. We traverse from output nodes to input nodes in a depth-first manner to compute an expression for output bits in terms of primary inputs. In this algorithm, we take output nodes as source nodes and recursively traverse nodes that are input to the current node and normalize them. To prevent nodes from visiting multiple times in depth-first traversal, after normalizing a node, its normalized expression is stored in a map. During recursion, if a node is present in the map, normalized expression is returned otherwise recursion continues. The following is the algorithm

Algorithm 3 Depth-first rewriting

Input: Gate-level netlist of $GF(2^m)$
Output: Output bit expression

```
1: Build the graph from netlist.
2: Extracting global expression of output nodes
3: for each output node do
4:    for each source node do
5:       inputs = depth-first-recursive(source node)
6:    end for
7:    Compute the normalized expression using normalization
8: end for
9: Depth-first recursive(node)
10: if node is in the map then
11:    Return the normalized expression which is already computed
12: else
13:    for each source node do
14:       Inputs = depth-first-recursive(source node)
15:    end for
16:    Compute the normalized expression for using normalization
17:    Store node and normalized expression in map
18:    Return normalized node
19: end if
```

Example: In figure 3.2, signals in square box are inputs and $\odot$ are node whose gate type is "AND" and $\bigoplus$ are nodes whose gate type is "XOR" and output signals are denoted by rhombus box. Figure 3.3 show steps in algorithm and nodes colored green are normalized. Table 3.1 explains operations that occur during recursion of depth-first rewriting algorithm.

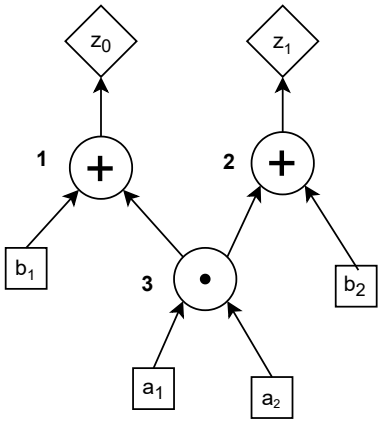
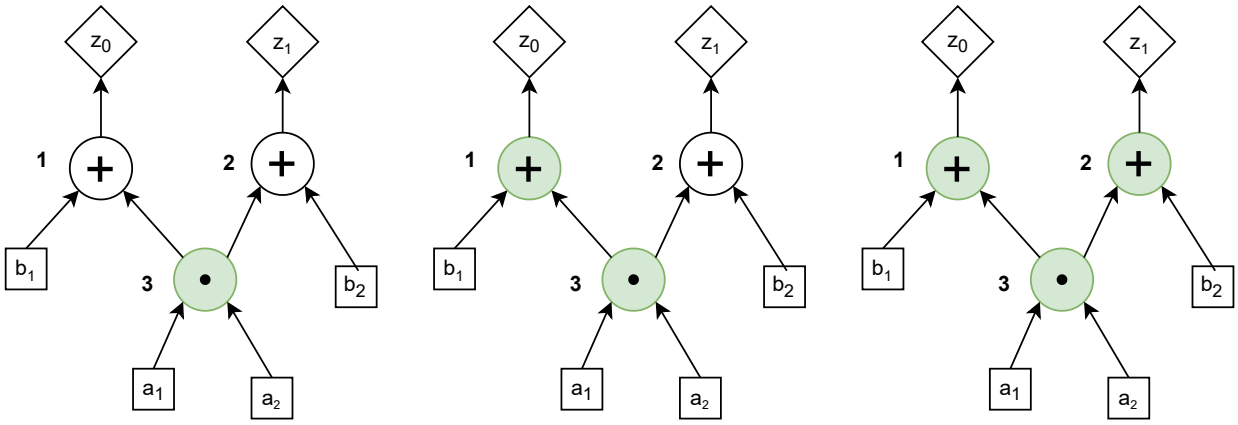


Figure 3.2: Graph



(a) (b) (c)

Figure 3.3: Depth first traversal steps

Table 3.1: Rewriting using Depth-first algorithm

Nodes on recursion stack	Operation
1	In figure 3.2, we can observe that one of sources of node 1 i.e node 3 is not normalized. So make a recursion call to node 3.
1,3	In figure 3.3a, we can observe sources of node 3 are primary inputs and so normalize node 3 and store normalized expression in a map and return normalized expression. We can observe node 3 is colored green 3.3a as it is normalized.
1	Now compute normalized expression for node 1 and node 1 is colored green in 3.3b and push node 2 on recursion stack.
2	In figure 3.3b, to compute normalized expression for node 2, make a recursion call to sources nodes. As one of input is primary node, make a recursion call to node 3.
2,3	In figure 3.3b, we can observe that node 3 is already normalized and return normalized expression stored in the map.
2	Now compute normalized expression for node 2 and node 2 is colored green in 3.3c as it is normalized now.

Normalization is a step in which symbolic expression for a node is computed. Normalization methods depend on the type of gate. Graph has "AND" , "XOR" gates only because "OR" and inverter gates are converted into "AND" and "XOR" gates using formula stated in chapter 2 while forming the node in graph. Subsequent sections will discuss various methods for normalization in detail.

3.3.1 XOR

The inputs to "XOR" normalization are two sources which are a list of cubes. The output will be a list of cubes that is the union of the two sources, except that any cube that appears in both sources is removed. The objective of normalizing the "XOR" gate is to remove cubes that are present in both the inputs of "XOR" gate. This can be achieved in the following ways depending on how sources to "XOR" is constructed. Different scenarios

of sources are individual cubes in sources are not sorted, cubes are sorted individually and cubes are present in a hash map instead of a list. Following are the algorithms in these scenarios

Approach 1

In this approach, it is assumed that cubes in both input lists are not sorted. Since cubes are not sorted, we can't use string equivalence checking to identify equal cubes. We can sort individual cubes and append them into a single list and sort the single list. Now, we can sort the single list and remove identical cubes since they appear together in the sorted list. Following is the algorithm

Algorithm 4

Input: Two inputs which are list of cubes
Output: Normalized "XOR" output of two inputs

1: Append one input to other and form a single list of cubes namely flist
2: Sort each cube in flist and sort the entire flist
3: **for** cube in flist **do**
4: Remove subsequent cubes which are equal from flist
5: **end for**
6: return flist

The time complexity for above algorithm can be analysed in following way. Let length of inputs list be n and m and length of largest cube be p. Step 1 takes $O(n)$ where n is size smallest list. In step 2, sorting each cube takes $O(plogp)$ and for sorting entire $p(n+m)log(n+m)$. For loop in line 3 takes $(n+m)$. Overall time complexity for entire algorithm is $O(n + plogp + p(n+m)log(n+m) + (n+m))$ which simplifies to $O(p(n+m)log(n+m))$. The space complexity of this approach is $O(n+m)$.

Example:

$input_1 = [$"$a_1 * b_2 * a_3 * a_4$",
"$a_1 * b_1$"$]$
$input_2 = [$"$a_4 * b_2 * a_1 * a_3$",
"$a_1 * b_2$"$]$
flist $= [$"$a_1 * b_2 * a_3 * a_4$",

"$a_1 * b_1$",
"$a_4 * b_2 * a_1 * a_3$",
"$a_1 * b_2$"]

Sort individual cubes flist = ["$a_1 * b_1$",
"$a_1 * b_2$",
"$a_1 * a_3 * a_4 * b_2$",
"$a_1 * a_3 * a_4 * b_2$"]

Remove cubes which are identical = ["$a_1 * b_1$", "$a_1 * b_2$"]

Approach 2

In this approach, it is assumed that cubes in both inputs are sorted individually. Since cubes are sorted, we can use a hash map that has constant time access to identify equal cubes and form a resultant list of cubes. Following is the algorithm

Algorithm 5

Input: Two inputs which are list of cubes
Output: Normalized "XOR" output of two inputs

```
1: Initialize a hash map
2: for cube in input1 do
3:    insert into hash map
4: end for
5: for  cube in input2 do
6:    if  cube in hash map then
7:       remove cube from hash map
8:    else
9:       insert cube in hash map
10:   end if
11: end for
12: return keys of hash map
```

The time complexity for the above algorithm can be analyzed in the following way. Let the length of the inputs list be n and m and the length of the largest cube be p. For loop in line 2 takes $O(n)$ and for loop in line 5 takes $O(m)$ and insert and deletion operations of the hash map take $O(1)$. The overall time complexity for the entire algorithm is $O(m+n)$. The space complexity of this approach is $O(n+m)$.

Example:

$input_1 = [$"$a_1 * b_1$",
"$a_1 * a_3 * a_4 * b_2$"]
$input_2 = [$"$a_1 * b_2$",
"$a_1 * a_3 * a_4 * b_2$"]
Remove cubes which are identical using hash map and keys of hash map will be ("$a_1 * b_1$", "$a_1 * b_2$")

Approach 3

In this approach, it is assumed that cubes are sorted and in hash map instead of list. Since cubes are sorted and present in hash map, we can traverse keys of one input hash map and check key is present in another hash map or not. If it is present remove it else insert key into hash map. This approach takes $O(n+m)$ where n, m are lengths of input hash maps.

Example:

Let $input_1$ = hash map with keys ("$a_1 * b_1$",
"$a_1 * a_3 * a_4 * b_2$")
$input_2$ = hash map with keys ("$a_1 * b_2$",
"$a_1 * a_3 * a_4 * b_2$")
Keys of resultant hash map will be ("$a_1 * b_1$", "$a_1 * b_2$")

Analysis

Among all these approaches, approach 2 works faster then remaining. Approach 3 also has same time complexity as approach 3 but in practice approach 2 is faster then approach 3.

3.3.2 AND

Let inputs to "AND" gate are input1 and input2. The output will be a list of cubes which can be obtained by multiplying each cube in input1 by the other in input2 and append to the list. Now, simplify each cube by removing the same elements in each cube and retaining

one of them, and sorting each cube. Sorting of the cube is required as cubes are used as a key in the hash map to identify equal cubes which are similar to "XOR" normalization. Now, remove identical cubes using the hash map, and the keys of the hash map will be output. Bottleneck while performing above is multiplying two cubes. This can be achieved by following approaches

Approach 1

In this approach, first, obtain signals from each cube and append them to a list. Signals from each cube can be obtained by splitting the cube by delimiter $*$. Now sort the list and remove signals which are equal by traversing the list and obtaining only one copy of the signal. The final output will be concatenating signals in the list by $*$.

Example:

Let cube1: "$a_2 * b_3 * a_4 * a_5$" cube2: "$b_3 * a_5 * b_6 * a_6$"
Obtain signals from cube 1. let be list 1: ["a_2", "b_3", "a_4", "a_5"]
Obtain signals from cube 2. let be list 2: ["b_3", "a_5", "b_6", "a_6"]
Concatenate list 1, list 2 to form list: ["a_2", "b_3", "a_4", "a_5", "b_3", "a_5", "b_6", "a_6"]
Sort list: ["a_2", "a_4", "a_5", "a_5", "a_6", "b_3", "b_3", "b_6"]
Remove signals which are equal and retain one of them: ["a_2", "a_4", "a_5", "a_6", "b_3", "b_6"]
Form the cube by concatenating signals by $*$: ["$a_2 * a_4 * a_5 * a_6 * b_3 * b_6$"]

Approach 2

In this approach, same as the previous obtain signals from each cube and insert them into a hash map instead of a list. While inserting to the hash map, check if the signal is already present in the hash map or not. If it is present, do not insert it, else insert it into the hash map. Now, the output will be keys of the hash map in a sorted manner. Sorting is required because cubes are used as keys in the hash map in the approach of "XOR" normalization.

Example:

Let cube1: "$a_2 * b_3 * a_4 * a_5$" cube2: "$b_3 * a_5 * b_6 * a_6$"
Obtain signals from cube1. let be list1: ["a_2", "b_3", "a_4", "a_5"]

Obtain signals from cube2. let be list2: [“b_3”, “a_5”, “b_6”, “a_6”]
Insert signals into hash map and keys will be m: [“a_2”, “b_3”, “a_4”, “a_5”, “b_6”, “a_6”]
Sort m: [“a_2”, “a_4”, “a_5”, “a_6”, “b_3”, “b_6”]
Form the cube by concatenating signals by $*$: [“$a_2 * a_4 * a_5 * a_6 * b_3 * b_6$”]

Analysis

Among the above 2 approaches, approach 2 works faster than approach 1 because it involves sorting the whole list of signals from two cubes and traversing the list on the other side approach 2 takes advantage of the hash map to simplify and only uses sorting for keys of the hash map which will be lesser than the original list of signals which approach 1 uses. Following is the example for “AND” normalization.

Example

Let input1: [“$a_1 * a_4$”, “$a_2 * b_1$”] input2: [“$a_1 * a_2$”, “$a_4 * b_1$”]
List of cubes obtained by multiplying each cube by other is
list = [“$a_1 * a_2 * a_4$”, “$a_1 * a_4 * b_1$”, “$a_1 * a_2 * b_1$”, “$a_1 * a_4 * b_1$”]
Remove identical cubes using hash map. Hash map keys will be
Output = [“$a_1 * a_2 * a_4$”, “$a_1 * a_2 * b_1$”]

3.4 Verification

After rewriting is completed, a map containing the output bit expression is computed. The equivalence of output bit expression in terms of primary inputs of two models can be verified using the following approaches.

3.4.1 String based

Since the expression is a list of cubes that are strings, one can verify the equivalence of two expressions by comparing individual cubes. Since cubes are sorted, two expressions are equivalent if cubes one expression are present in another expression using a hash map and their expression lengths are equal.

The normalization algorithm produces algebraic normal form expressions and checking equality of developed data structure is equivalent to SAT verification.

Example
Let us demonstrate if two cubes point to same data structure developed, then there are equivalent. Let the cube1 be "$a_1 * a_1 * b_1 * b_2$" and cube2 be "$a_1 * b_1 * b_1 * b_2$". After applying "AND" normalization method for cube1, it normalizes to "$a_1 * b_1 * b_2$" and cube2 normalizes to "$a_1 * b_1 * b_2$". We can observe that cube1 and cube2 normalizes to same and therefore cube1, cube2 are equivalent. Therefore developed data structure is truly algebraic normal form for cubes. Now consider a following two expressions.

Expression 1 = ["$a_1 * b_1$",
"$a_2 * b_3$",
"$a_2 * b_2 * b_2$",
"$a_2 * a_2 * b_2$"]

Expression 2 = ["$a_2 * b_3$",
"$a_1 * b_1$",
"$a_2 * b_3 * b_3$",
"$a_2 * a_2 * b_3$"]

We can observe that both expressions can be normalized to ["$a_2 * b_3$", "$a_1 * b_1$"]. Therefore developed data structure is truly algebraic normal form for expressions also.

3.4.2 SAT Solver based

For SAT solver based equivalence verification, we can use Z3 [14] sat solver and form a equivalent Z3 [14] command out of expression and check its equivalence with other.

Example : Expression 1 z_1 = ["a_1", "a_2"]
Expression 2 z_2 = ["a_2", "a_1"]

Following are the Z3 commands to check equivalence of z_1, z_2

```
SetLogic( QF_UF())
DeclareConst(SSymbol(‘‘a1’’), Sort(Identifier(SSymbol(‘‘Bool’’) )))
DeclareConst(SSymbol(‘‘a2’’), Sort(Identifier(SSymbol(‘‘Bool’’) )))
DefineFun(FunDef(SSymbol(‘‘z1’’),List(), Sort(Identifier(SSymbol(‘‘Bool’’))),
```

```
FunctionApplication(QualifiedIdentifier(Identifier(SSymbol(‘‘xor’’))),
Seq(QualifiedIdentifier(Identifier(SSymbol(‘‘a1’’))),
QualifiedIdentifier(Identifier(SSymbol(‘‘a2’’)))))))

DefineFun(FunDef(SSymbol(‘‘z2’’),List(), Sort(Identifier(SSymbol(‘‘Bool’’))),
FunctionApplication(QualifiedIdentifier(Identifier(SSymbol(‘‘xor’’))),
Seq(QualifiedIdentifier(Identifier(SSymbol(‘‘a2’’))),
QualifiedIdentifier(Identifier(SSymbol(‘‘a1’’)))))))

Assert(FunctionApplication(QualifiedIdentifier(Identifier(SSymbol(‘‘not’’))),
Seq(FunctionApplication(QualifiedIdentifier(Identifier(SSymbol(‘‘=’’))),
Seq(QualifiedIdentifier(Identifier(SSymbol(‘‘z1’’))),
QualifiedIdentifier(Identifier(SSymbol(‘‘z2’’)))))))))

CheckSat()
Exit()
```

3.4.3 Analysis

From experiments, we observed that string based approach take less time than SAT solver. The results of experiments are presented in chapter 5.

3.5 Summary

There are various drawback for algorithm developed in this chapter. One of the drawback is, using list of strings data structure for expression. Since strings are used to represent expression, same sub-expression may exist many times in different expressions. This results in consuming a lot of memory and increases time to compute expressions as we are comparing two expressions multiple times during normalization. Another drawback is common nodes between two models are traversed twice.

In this chapter, we have introduced graph representation of netlist and depth-first rewriting algorithm. We have introduced a data structure for expressions that represents algebraic normal form. We have introduced various approaches for “AND”, “XOR” normalization and equivalence verification of output bit expressions obtained from rewriting. The results of the developed algorithm are presented in chapter 5. We have identified the

drawbacks of our algorithm. We will further develop an alternative algorithm in chapter 4 overcoming these drawbacks.

Chapter 4

Graph Based Approach

In this chapter, we will introduce an alternative method for the verification of GF multipliers. In the previous chapter, symbolic expressions are represented as a list of strings. With a string-based approach for expression, we can't identify a common substructure between two expressions and the same sub-expression can be present in multiple expressions. To overcome these issues, an alternative data structure based on graph is introduced for symbolic expression in this chapter and an alternative graph data structure which represents a circuit and an algorithm to extract symbolic expression for output bit is introduced. Alternative normalization and verification methods are developed as the underlying data structure for symbolic expression is changed.

4.1 Graph

The netlist can be transformed into a graph. One can traverse this graph and can compute a symbolic expression for output bits. The developed graph data structure consists of four data structures namely gate, node, and two hash tables. Every gate in netlist represents gate data structure in the following way. Gate has information about sources and type of circuit gate. There are different types of gates namely gates with two inputs, input gates for primary inputs, and constant gates to represent true or false. The node data structure contains a number that is assigned to each unique gate, the destinations of the gate and gate. Two nodes are equal if they have the same node number. Similarly, node 1 is less than node 2 if the node number of node 1 is less than node 2. Sources in the gate are represented in form of nodes and source 0 is always less than source 1. Node number is generated through a counter.

Graph also consist of two hash tables namely *gateToNode* and *signalToNode*. *gateToNode* is required to prevent duplication of gates in the graph. The node which the gate represents can be found using *gateToNode* where the gate is key and node is the value. This hashing of gates guarantees that identical gates in the circuit are all represented by the same node in the graph. There may be different signals which represent the same gate, and to prevent duplication of the same gate, since the node is unique to every gate, without creating a new gate, we can store signal information in *signalToNode* data structure where a signal is key and node is value.

Consider construction of graphs in 4.1 consecutively. Tables 4.2, 4.3 show *gateToNode*, *signalToNode* tables respectively after building graphs.

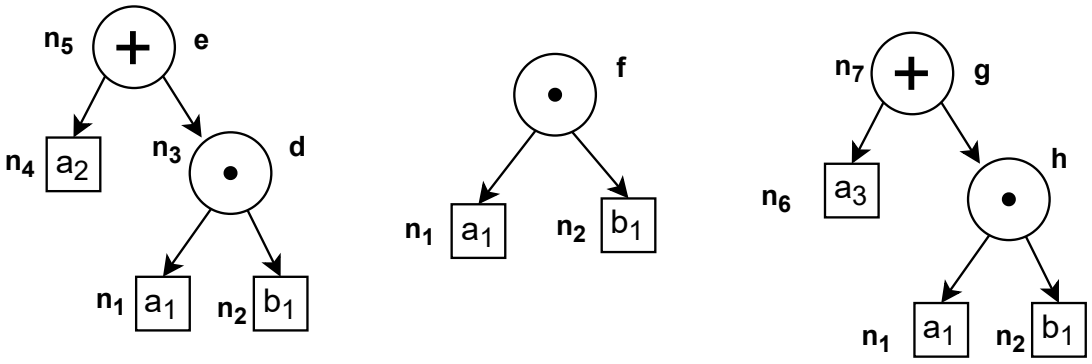


Figure 4.1: Graph

Table 4.1: Hashtables

Table 4.2: GateToNode

gate	node number	destinations
Input a_1	1	[3]
Input b_1	2	[3]
AND(1,2)	3	[5, 7]
Input a_2	4	[5]
XOR(4,3)	5	[ ]
Input a_3	6	[7]
XOR(6,3)	7	[ ]

Table 4.3: SignalTonode

signal	node
a_1	1
b_1	2
d	3
a_2	4
e	5
f	3
a_3	6
h	3

When constructing graph for middle figure in 4.1, we will first check whether a gate with sources 1, 2 and gate type "AND" is present in *gateToNode* table or not. Since it is already present, no new entry is inserted to *gateToNode* table and new signal f is mapped to node 3 in *signalToNode* table.

4.2 Algebraic Normal Form

Expression computed at every node in the graph is represented in normal form. The normal form is a binary tree. The left child is termed source 0 and the right child is termed source 1. The left child of node in binary tree is always a leaf or a cube. Leaf nodes are always primary inputs or constants. Therefore binary tree is always extended on right. All leaf nodes are sorted in ascending order from top to bottom. All the above features of binary tree makes it unique. The root node is either "XOR" or "AND".

Figure 4.2 shows valid normal forms

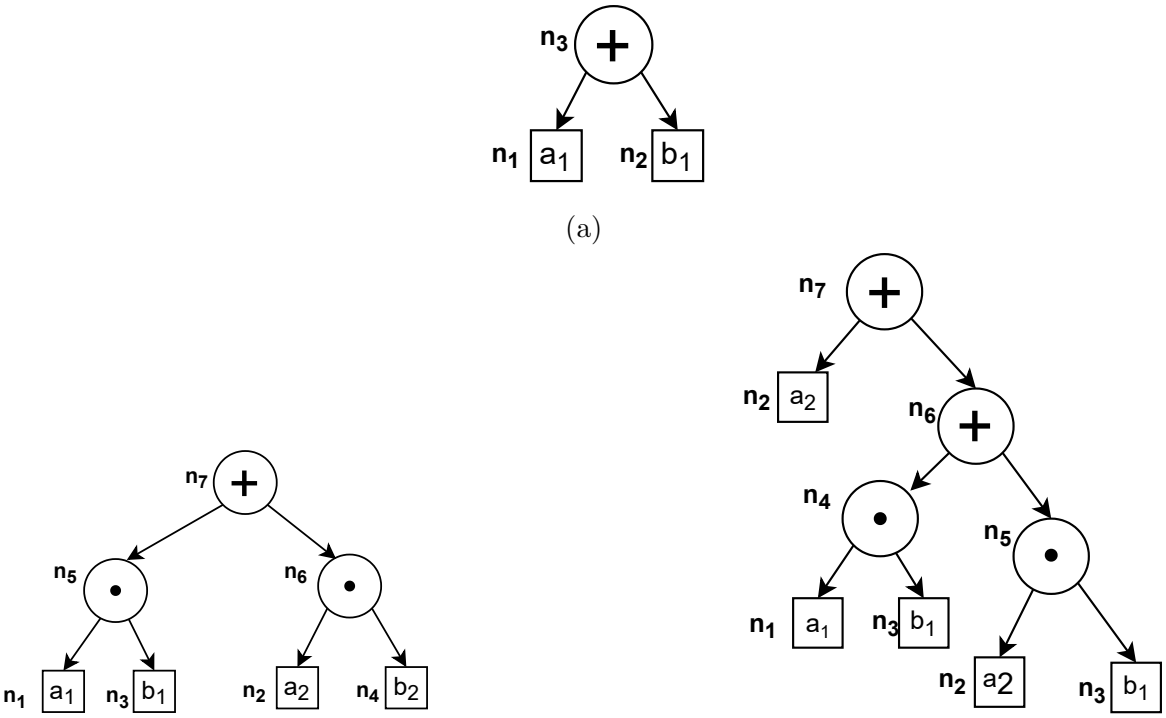
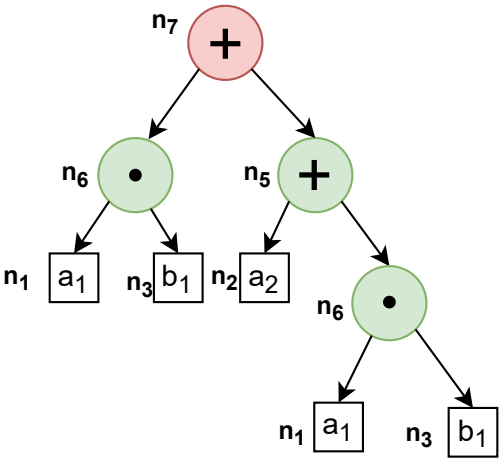
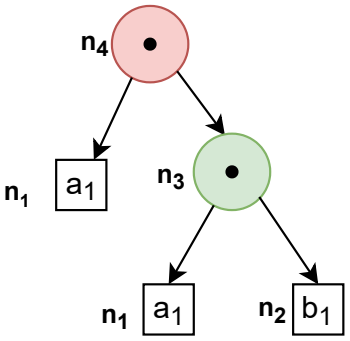


Figure 4.2: Valid Normal forms

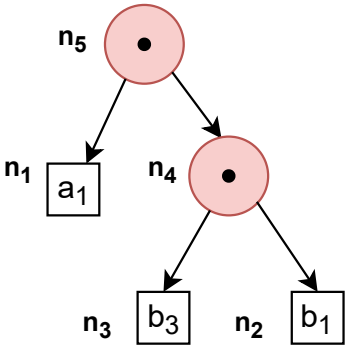
Nodes in green are valid nodes and nodes in red are invalid nodes. **Figure 4.3 shows invalid normal forms**



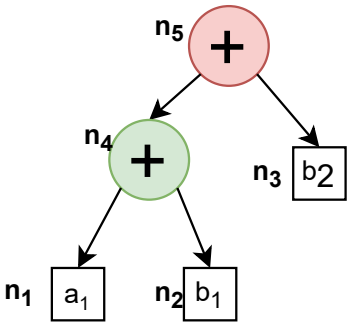
(a)



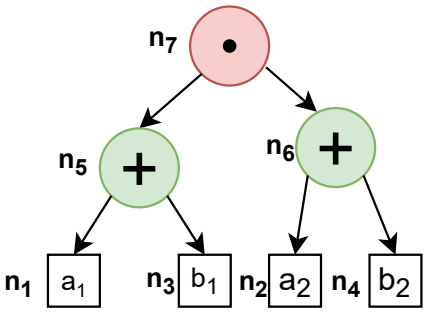
(b)



(c)



(d)



(e)

Figure 4.3: Invalid Normal forms

Following are the reasons for invalid normal forms

- 4.3a is invalid normal form because it has "XOR" as root node and cube n_6 as the source node for two root nodes and n_2, n_6 cubes are not sorted as n_2 should come before n_6.
- 4.3b is invalid normal form because it has "AND" as the root node and leaf node n_1 is source node for two root nodes.
- 4.3c is invalid normal form because, at node 4, its leaves are not sorted.
- 4.3d is invalid because source 0 of the root node which is "XOR" is the "XOR" node.
- 4.3e is invalid because the root node is "AND" and its sources nodes are "XOR".

The normal form has the following principles

- If the root node is "XOR", its source 0 can be "AND" or leaves, and the root node of source 1 is "XOR" or "AND" or leaf.
- If the root node is "AND", its source 0 is leaves and source 1 is either "AND" or leaves.
- If the root node is "XOR", there can't exist two identical cubes in the normal form.
- If the root node is "AND", there can't exist two identical leaves in the normal form.
- All the cubes from the root are sorted in ascending order.
- Since all cubes are sorted, the normal form is unique for a given symbolic expression.

It is assumed that a, b are n bit primary bits. Primary inputs are formed before any other nodes in order a_1, ... a_n,b_1,...b_n and therefore $a_1 < a_2 < ..... < a_n < b_1 < .... < b_n$.

4.3 Normalization

A graph is built for the specification, implementation and output nodes are identified. Symbolic expression for output bit is extracted by traversing graph in a depth-first manner. Graph traversal starts from output nodes and explored as far as possible recursively until traversal reaches the input nodes. When traversal reaches the input nodes, global

expression is computed and recursion call returns. To prevent traversing common nodes multiple times, whenever a node is normalized it is stored in a map from the current node to the normalized node and so while recursion, it is verified that the current node is present in the map or not. If it is present, the normalized node is returned else recursion continues. Hence all nodes are normalized only once.

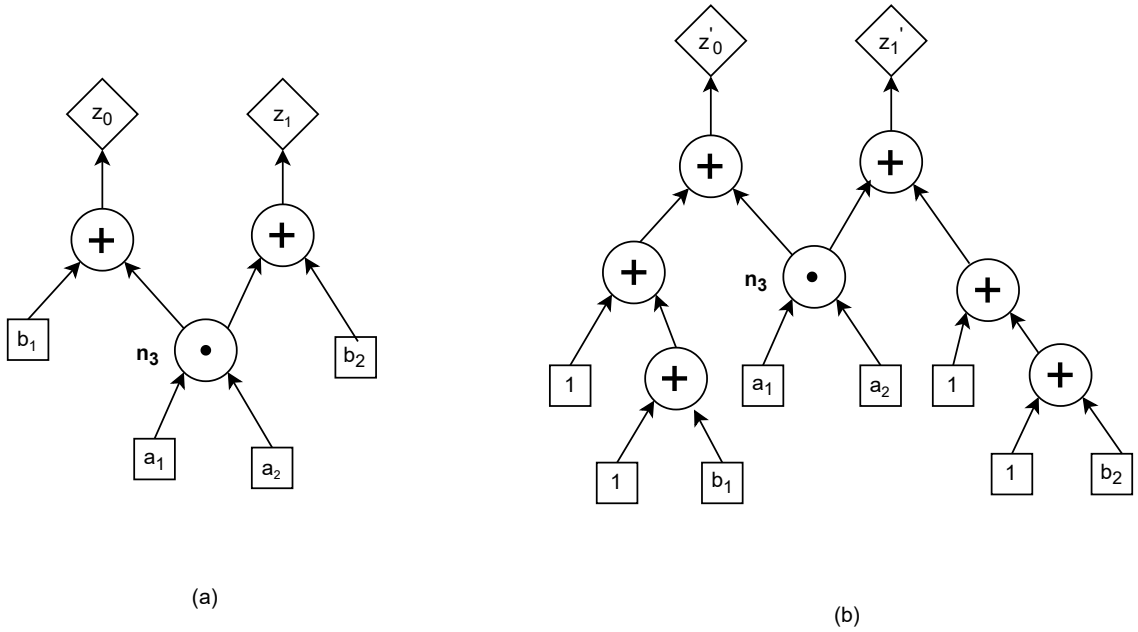


Figure 4.4: Specification and Implementation Graphs

In figure 4.4, sub-figure (a) is graph for specification and sub-figure (b) is graph for implementation model. Let z_0, z_1 are output bits for the specification model and $z_0^{'}$,$z_1^{'}$ be output bits for the implementation model. We can observe that n_3 is common to both models and common to output bits. There will be only one copy of n_3 and it is normalized only once during depth-first traversal.

Sections 4.3.1, 4.3.2 presents normalization methods for "XOR" and "AND" respectively.

4.3.1 XOR

The input argument to normalization function for an "XOR" is a node where both sources are normalized. The result of function is the union of cubes except that any cube that

appears in both sources is removed. We will present 4 approaches for normalizing "XOR" gate and each approach improves upon its predecessor.

Let input argument be figure 4.5. The number adjacent to leaves or cubes is the node number. The node in green color is normalized and the node in red is not normalized.

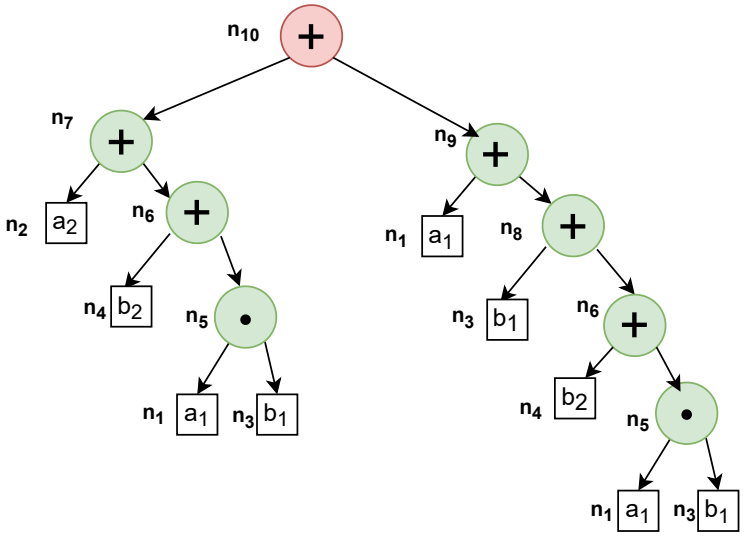


Figure 4.5: Input node for "XOR" normalisation

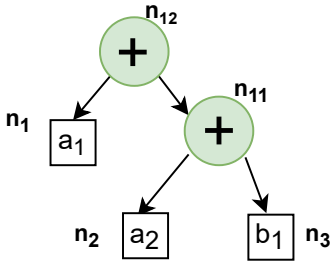


Figure 4.6: Output of "XOR" normalisation

Approach 1: Sorting

In this approach, obtain cubes from two sources in a list and concatenate them into a single list. Now, sort the cubes and remove cubes that are identical by iterating over the

list with the head and tail of the list. Now, build the resultant normal form with a list of cubes.

Example

Let cubes in source 0 be list1 = $[n_2, n_4, n_5]$
Let cubes in source1 list2 = $[n_1, n_3, n_4, n_5]$
Concatenate two list and sort them flist = $[n_1, n_2, n_3, n_4, n_4, n_5, n_5]$
Remove cubes which are equal flist = $[n_1, n_2, n_3]$
Build normal form with flist which result in figure 4.6

Approach 2: Merging

In above approach, instead of concatenating two list of cubes, since list of cubes are already sorted, use merge step of merge sort to remove cubes which are identical.

Approach 3: Recursive

In this approach, instead of traversing a tree to get the cubes and then sorting the cubes, we can recursively traverse the trees and remove identical cubes since cubes are sorted. While traversing, we can construct the resultant normal form. Following are the different scenarios that may occur during recursion. Let 'a', 'b' are two leaves and x, y are two cubes and it is assumed that 'a' is less than 'b'. Let simpXor be a recursive function.

Table 4.4: Recursive merge sort conditions and output

	source0	**source1**	**output**
1	a	a	false
2	a	b	a XOR b
3	a	(b XOR y)	a XOR (b XOR y)
4	a	(a XOR y)	y
5	(a XOR x)	a	x
6	(a XOR x)	b	a XOR (simpXor x b)
7	(a XOR x)	(b XOR y)	a XOR (simpXor x (b XOR y))
8	(a XOR x)	(a XOR y)	simpXor(x, y)

Example

Let input be figure 4.5
Following are the recursion steps

- In figure 4.7, the root node of source 0 which is n_7 is in form of $a_2 \oplus x$ and the root node of source 1 which is n_9 is in form of $a_1 \oplus y$. So it will fall under condition 7 of the table and output is $a_1 \oplus (\ simpxor(7,\ 8))$ where 7, 8 are root nodes of subtrees of x, y.
- Now for simpXor(7,8), with reference to figure 4.8, the root node 7 is in form of $a_2 \oplus 6$, and 8 is in form of $b_1 \oplus 6$ where 6 is a subtree of 7,8 nodes. So it will fall under condition 7 and output is $a_2 \oplus simpXor(6,\ 8)$.
- Now for simXor(6, 8), with reference to figure 4.9, the root node 6 is in form of $b_2 \oplus 5$ and 8 is in form of $b_1 \oplus 6$. So it will fall under condition 7 and output is $b_1 \oplus simpXor(6,\ 6)$.
- Now for simpXor(6, 6) , with reference to figure 4.10, both root nodes are the same so it will fall under condition 1 and output is false.
- Final result is $((a_1 \oplus a_2) \oplus b_1)$ whose normal form is figure 4.6.

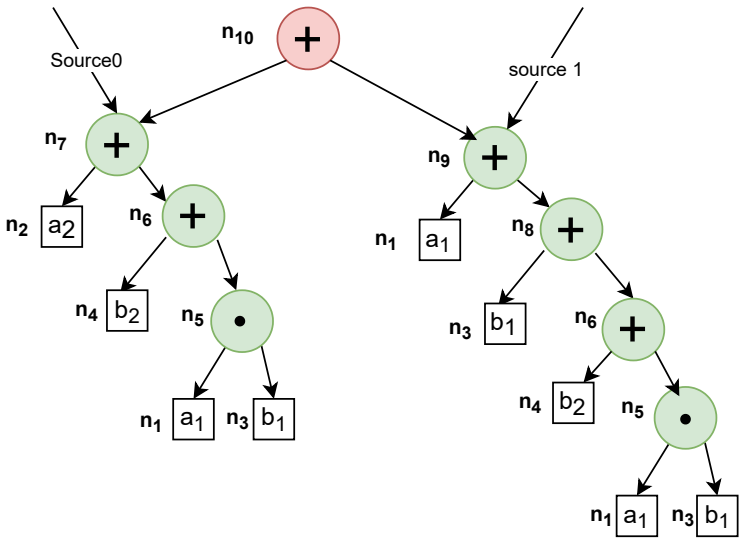


Figure 4.7: Step 1 of "XOR" recursion

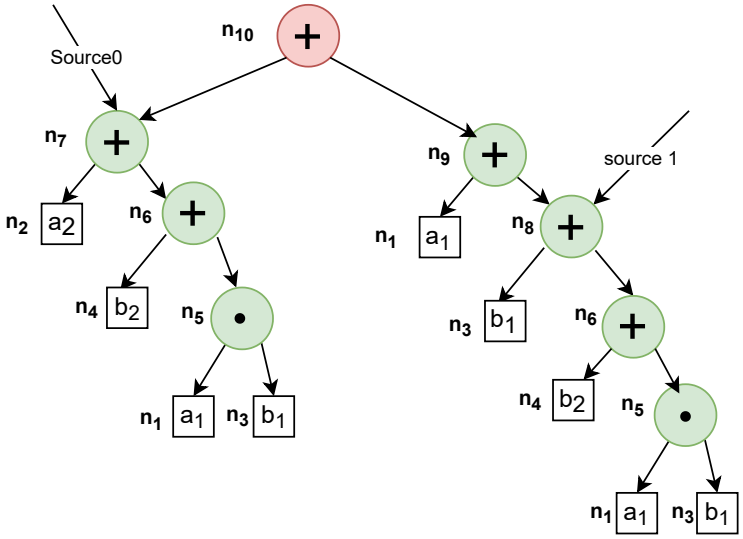


Figure 4.8: Step 2 of "XOR" recursion

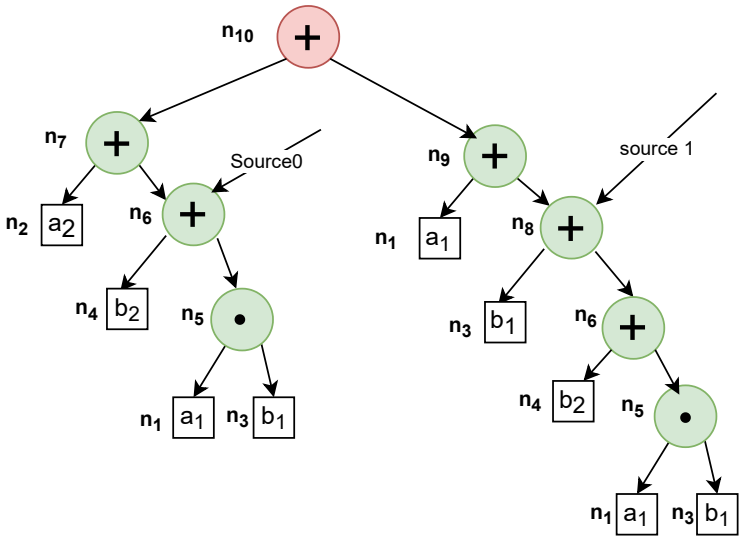


Figure 4.9: Step 3 of "XOR" recursion

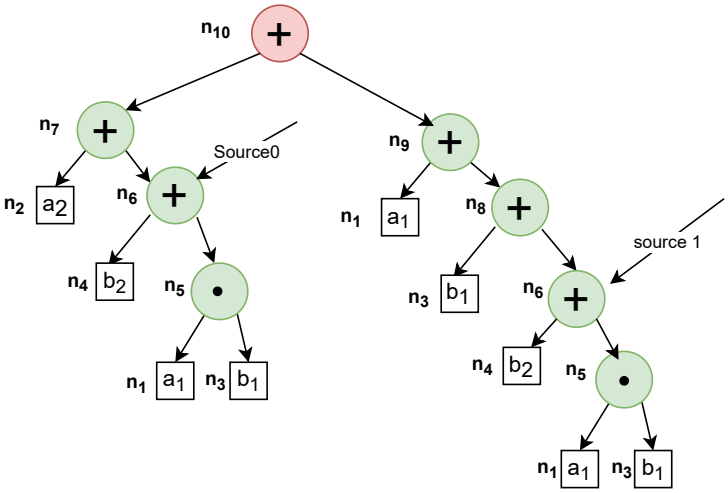


Figure 4.10: Step 4 of "XOR" recursion

Approach 4: Pseudo recursive

In approach 3, instead of recursion, one can traverse the trees in iterative manner, output at each step of recursion can be stored in a list and the resultant normal form can be formed from the list.

Example

Let input be 4.5
Following are the steps and list at each step

- Output in step 1 of recursion in approach 3 is a_1 list = $[a_1]$
- Output in step 2 of recursion in approach 3 is a_2 list = $[a_1, a_2]$
- Output in step 3 of recursion in approach 3 is b_1 list = $[a_1, a_2, b_1]$
- Now build normal form with the list which is figure 4.6

Analysis

Among all approaches, approach 4 takes less time than the remaining. In approaches 1 and 2, we need to traverse the sources to get cubes and rebuilt the normal form with the resultant cubes. Approach 3 builds the resultant normal form while traversing sources which saves time converting the normal form back and forth. But in approach 3, the recursion stack may run out of memory because the height of sources may be very high. Approach 4 mimics the recursion stack using the list and builds the resultant normal form with the list. Approach 4 saves time while traversing sources in an iterative manner and computing resultant cubes simultaneously.

4.3.2 AND

The input argument to normalization function for an "AND" is a node where both sources are normalized and gate of root node is of type "AND". Let children of root node be source 0 and source 1. The output is a union of cubes obtained from pairwise multiplying cubes from source 0 and source 1 in normal form. First, get the cubes from children and multiply every pair of cubes, and store them in a list. Now, build a normal form after getting "XOR" normalization of the list of cubes by applying approach 1 of "XOR" normalization. The bottleneck while performing the above operation is multiplying two cubes. Multiplying cubes take two cubes as inputs and output is a cube which is "AND" of two input cubes. To compute this take union of two cubes and sort the signals. Let cube 1 be 4.11 and cube 2 be 4.12 and the output of multiplying cube 1 and cube 2 is 4.13. It is assumed that 'a' is less than 'b'. We will present three approaches for multiplying two cubes and each approach improves upon its predecessor.

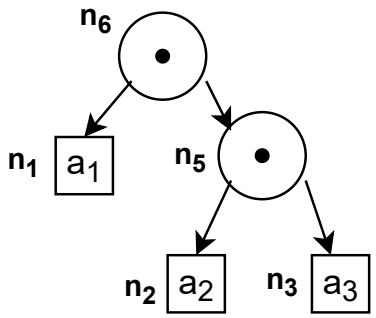


Figure 4.11: Input cube 1 for "AND" normalization

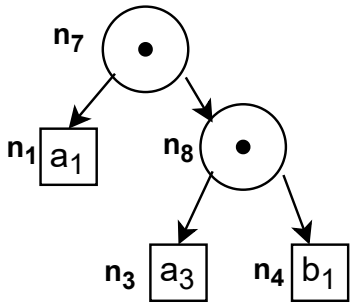


Figure 4.12: Input cube 2 for "AND" normalization

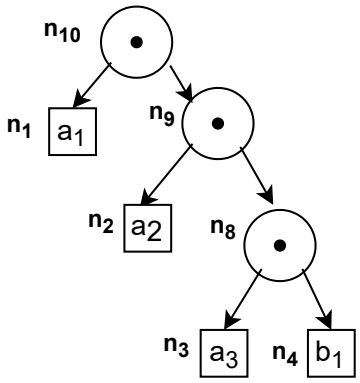


Figure 4.13: "AND" of cube 1 and cube 2

Approach 1: Merging

Union of two cubes can be computing by doing a merge sort of two cubes to remove duplicates. The leaves in individual cubes can obtained by traversing the cube in a list and do merge step of merge sort to remove duplicates and build a normal form with resultant list of cubes obtained after merge sort. Following is the algorithm

Example

Let leaves of cube1 (figure 4.11) list1 = $[n_1, n_2, n_3]$
Let leaves of cube2 (figure 4.12) list2 = $[n_1, n_3, n_4]$

After applying above algorithm flist = $[n_1, n_2, n_3, n_4]$
Normal form which is built from flist will be figure 4.13

Approach 2: Set

Alternatively in approach 1, one can use set to find union of two list of leaves and build a normal form after sorting elements in a set.

Example

Let leaves of cube1 (figure 4.11) list1 = $[n_1, n_2, n_3]$
Let leaves of cube2 (figure 4.12) list2 = $[n_1, n_3, n_4]$
After applying above algorithm set = $\{n_1, n_4, n_3, n_2\}$
Normal form which is built from set after sorting will be figure 4.13

Approach 3: Recursive

Merge sort over cubes can be performed by traversing two cube 1, cube 2 recursively. Following table explain different conditions that may occur during recursion. let a, b are two leaves and x, y are two cubes and it is assumed that 'a' is less than 'b'. Let mulcubes be the recursive function which computes "AND" of two cubes.

Let us discuss various situations for mulcubes function. We know that $a \odot a = a$, therefore $mulcubes(a,\ a) = a$. For $mulcubes(a,\ b)$, since leaves are different, result will be $a \odot$b.

Table 4.5: Recursive mergesort conditions "AND" output

	cube 1	**cube 2**	**output**
1	a	a	a
2	a	b	a AND b
3	a	(a AND x)	(a AND x)
4	a	(b AND x)	(a AND (b AND x))
5	(a AND x)	a	(a AND x)
6	(a AND x)	b	(a AND mulcubes(b, x))
7	(a AND x)	(a AND y)	(a AND mulcubes(x, y))
8	(a AND x)	(b AND y)	(a AND mulcubes(x, (b AND y)))

For $mulcubes(a,\ a \odot x)$, since a appears in both cubes, only one a will be present in result and result will be $a \odot x$. For $mulcubes(a,\ b \odot x)$ since a, b are different, both appears in result and result will be $a \odot b \odot x$. For $mulcubes(a \odot x,\ b)$, since a is less than b, a will be included in result and there will be recursive call $mulcubes(x,\ b)$. The final result will be $a \odot\ mulcubes(b,\ x)$. For $mulcubes(a \odot x,\ a \odot y)$, since a is present in both cubes, it will appear only once in result and since $x,\ y$ are different, there will be recursive call. The final result will be $a \odot\ mulcubes(x,\ y)$. For $mulcubes(a \odot x,\ b \odot y)$, since a is less than b, a appears before b in the result and there will be recursive call. The final result will be $a \odot mulcubes(x,\ (b \odot y))$.

Example

Consider multiplication of cube 1 (refer figure 4.11) and cube 2 (refer figure 4.12). Following are the recursion steps

- The root node of cube 1 which is n_6 is in form of $a_1\ \odot\ n_5$ and the root node of cube 2 which is n_7 is form of $a_1\ \odot\ n_8$ and so it will fall under condition 7 and output is $a_1\ \odot\ mulcubes(n_5,\ n_8)$. We can observe this in 4.14 where two pointers pointing to root nodes 6,7.
- Now in $mulcubes(n_5,\ n_8)$, n_5 is in form of $n_2\ \odot\ n_3$ and n_8 is in form of $n_3\ \odot\ n_4$ and so it falls under condition 8 and output is $n_2\ \odot\ mulcubes(n_3,\ n_8)$. We can observe this in 4.15 where pointers are pointing to nodes 5, 8.
- Now in $mulcubes(n_3,\ n_8)$, n_3 is leaf and n_8 is in form of $n_3\ \odot\ n_4$ and so it falls under condition 3 and output is n_8. we can observe this in 4.16 where pointers pointing to nodes 3, 8
- Final output will be figure 4.13

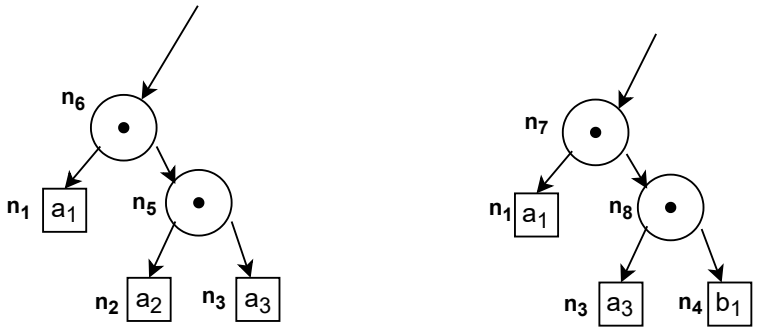


Figure 4.14: Step 1 of "AND" recursion

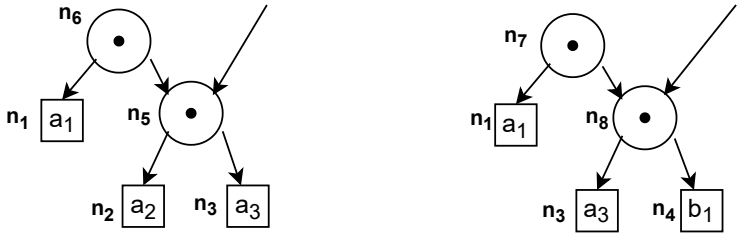


Figure 4.15: Step 2 of "AND" recursion

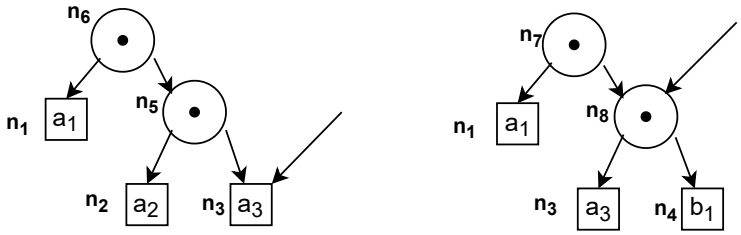


Figure 4.16: Step 3 of "AND" recursion

Analysis

Among the above approaches, approach 3 takes less time than the remaining. In approaches 1 and 2, we need to traverse sources to get leaves and rebuilt the normal form with resultant

leaves. But approach 3 builds resultant normal form while traversing sources recursively which saves time converting into normal form back and forth. Unlike approach 4 in "XOR" normalization, approach 3 doesn't suffer from recursion stack memory error since the height of cubes is small enough for the recursion stack.

4.4 Verification

We can verify equivalence of output bit expression in two ways namely node based and SAT solver based. Node based approach only depends on node to which expression is normalized. On other hand, SAT solver based approach uses SAT solver to verify equivalence of two expressions.

4.4.1 Node based

Since the normal form is unique i.e given an expression there exists only one normal form representation. Consider two output bits namely z_0, z_1 point to the same node, then the sources of the gate are the same. Let sources be n_1 and n_2 and by definition of normal form, n_1 is cube and n_2 may be a cube or sub-tree. So both output bit expressions contain n_1 cube and if n_2 is a cube then both the expression contain it else sources of n_2 and so on. Likewise, both expressions contain the same cubes and therefore they are equivalent. Therefore we can say that two expressions are semantically equal iff they are normalized to same node. Hence in this approach we just verify whether two expressions are normalized to same node or not.

The classic multiplier is used as specification and it is normalized by using algorithm developed. The normalized expression for each output bit is saved in a file and used as specification to verify implementation. we save time by not performing rewriting step for specification.

4.4.2 SAT Solver based

One can compute symbolic expression by traversing the normal form of the output node and feed two expressions into a SAT solver and SAT solver can tell whether they are equivalent or not.

4.4.3 Analysis

From experiments, we observed that the node-based approach takes less time than SAT solver based approach.

4.5 Summary

In this chapter, we have introduced alternative graph data structure and alternative algebraic normal form for expressions. These alternative data structure overcome the drawbacks of the string-based algorithm developed in chapter 3. We have also introduced alternative normalization methods and verification methods for developed data structures. Results of the developed algorithm are presented in chapter 5.

Chapter 5

Results

In this chapter, we present the results of our string-based approach and graph-based approach in comparison with Yu's [25] methods. Our graph based approach provide a range of 4×–256× speedup when compared to Yu's [25] method. We will present timing analysis for the verification for each multiplier. We will compare time required for various verification methods.

5.1 Experiment Setup

We synthesised netlist for classic, Mastrovito, Montgomery and Karatsuba from Synopsys Design Compiler version P-2019.03 and we have restricted the target ASIC library to use "AND", "XOR", "OR", "NOT" gates only.

We have implemented the algorithms stated in chapter 3 and chapter 4 in the scala programming language and experimented on the netlist of Mastrovito [20] and Montgomery [10], Karatsuba [23] multipliers of sizes 64, 128, 163, 233, 283, 571. Results are compared to that of the current state-of-art Yu's [25] algorithm. We ran our experiments on the computer whose configuration is Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz which is of 32GB RAM and cache size of 8192KB whereas Yu's [25] machine is of Intel(R) Xeon CPU E5-2420 v2 2.20 GHz × 12 with 32 GB memory. It's difficult to precisely compare the performance of our computer vs Yu's computer because ours has higher single-threaded performance (higher clock speed and newer architecture) and Yu's has more cores (24 threads for Yu vs 8 threads for us). But, the performance improvements of our algorithm vs Yu's range from 4× to 256×, which is much greater than any possible difference in raw computing power.

5.2 Comparison to Previous Work

Tables 5.1, 5.2 presents comparison of Mastrovito [20], Montgomery [10] multipliers between Yu's [25] work and graph based algorithm introduced in chapter 4. Table 5.3 present results of graph based algorithm for Karatsuba [23] multiplier. Yu's [25] didn't mention about Karatsuba [23] multiplier, so can't be compared. From figure 5.1, we can observe that our algorithm outperforms Yu's [25] algorithm with 30 threads for both multipliers. For the Mastrovito [20] multiplier of 283-bit, our algorithm gives 4× speedup for 30 threads and 14× speedup for one thread when compared to Yu's [25] algorithm. Yu's [25] algorithm can't verify 571-bit Mastrovito [20] multiplier due to memory issues but our graph-based algorithm can verify it. For Montgomery multiplier [10] of size 283, our algorithm gives 256× speedup for 30 threads, 556× speedup for one thread when compared to Yu's [25] algorithm. Yu [25] didn't mention about 571-bit Montgomery multiplier [10], but our graph based algorithm is able to verify it.

Table 5.1: Mastrovito Multiplier

MO stands for out-of-memory. T is the number of threads

Mastrovito	Yu [25] (s)					**This work (s)**
size	T=1	T=5	T=10	T=20	T=30	Runtime (s)
64	19	11	8	7	7	2.87
128	153	91	63	55	57	13.17
163	336	192	137	121	113	26.09
233	499	294	212	180	171	63.14
283	1580	890	606	550	530	112.64
571	13176	7980	5038	MO	MO	1488.31

Table 5.2: Montgomery Multiplier

N/A stands for not available. T is the number of threads

Montgomery	Yu [25] (s)					**This work (s)**
size	T=1	T=5	T=10	T=20	T=30	Runtime
64	80	45	31	28	27	2.25
128	335	209	121	151	110	6.65
163	2505	1616	1172	1095	1008	13.7
233	1240	722	565	457	480	24.87
283	32180	19745	17640	15300	14820	57.87
571	N/A	N/A	N/A	N/A	N/A	14022.72

Table 5.3: Karatsuba Multiplier

Size	Runtime (s)
64	1.69
128	3.98
163	5.92
233	5.24
283	9.84
571	44.73

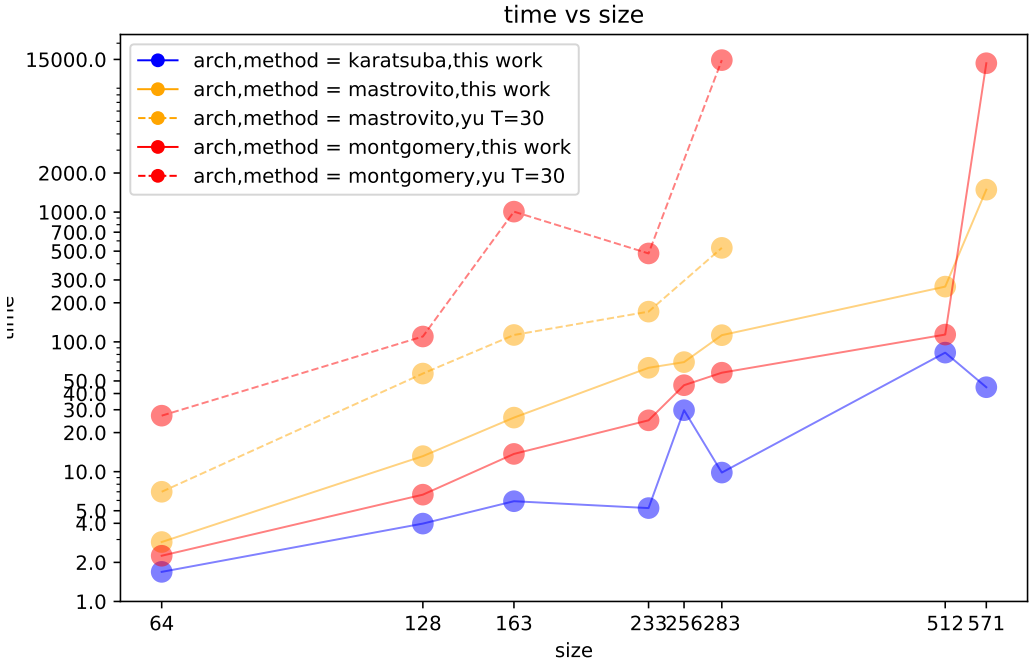


Figure 5.1: Yu vs this work

From tables 5.1, 5.2, we can observe that upto 283-bit Mastrovito takes more time than Montgomery but for 571-bit Montgomery takes 10× more time than Mastrovito. From table 5.3, we can observe that Karatsuba multiplier takes lesser time than Mastrovito and Montgomery multipliers. Tables 5.4, 5.5, 5.7 present time taken by graph-based algorithm in graph building which includes parsing normalized netlist of classic and implementation

model, rewriting and verification for Mastrovito, Montgomery, Karatsuba multipliers respectively. For all three multipliers, we can observe that when input size changes from 64 bit to 128 bit, time taken for graph building and rewriting is doubled but when input changes from 128 to 256 bit, time taken for graph building and rewriting became 10×. Again when input changes from 256 to 512 bit, time taken for graph building and rewriting is doubled. We can observe that with change in input size from 64 to 512 bit, time taken for verification for all three multipliers increases by 50%.

Table 5.4: Timing analysis for verification of Mastrovito multiplier

Size	Graph building (s)	Rewriting (s)	Verification (s)
64	1.26	0.23	0.019
128	4.02	1.1	0.019
256	47	25.51	0.025
512	109.41	154.74	0.028

Table 5.5: Timing analysis for verification of Montgomery multiplier

Size	Graph building (s)	Rewriting (s)	Verification (s)
64	1.02	0.32	0.024
128	3.03	1.68	0.022
256	26.27	21.59	0.028
512	69.09	44.13	0.030

Table 5.6: Timing analysis for verification of Karatsuba multiplier

Size	Graph building (s)	Rewriting (s)	Verification (s)
64	0.98	0.18	0.02
128	2.61	0.65	0.019
256	23.82	6.38	0.024
512	57.21	24.03	0.030

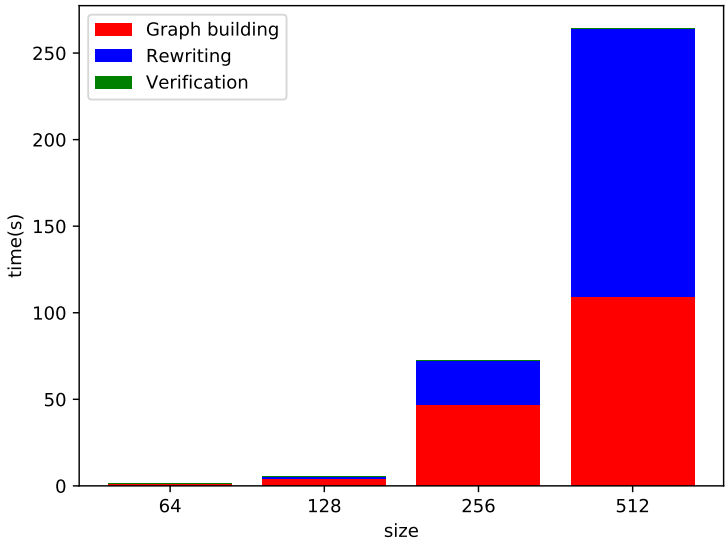


Figure 5.2: Timing analysis for verification of Mastrovito multiplier

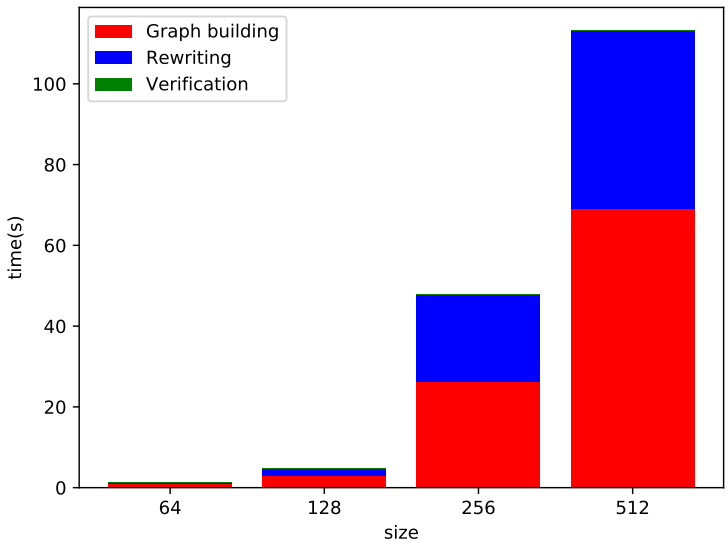


Figure 5.3: Timing analysis for verification of Montgomery multiplier

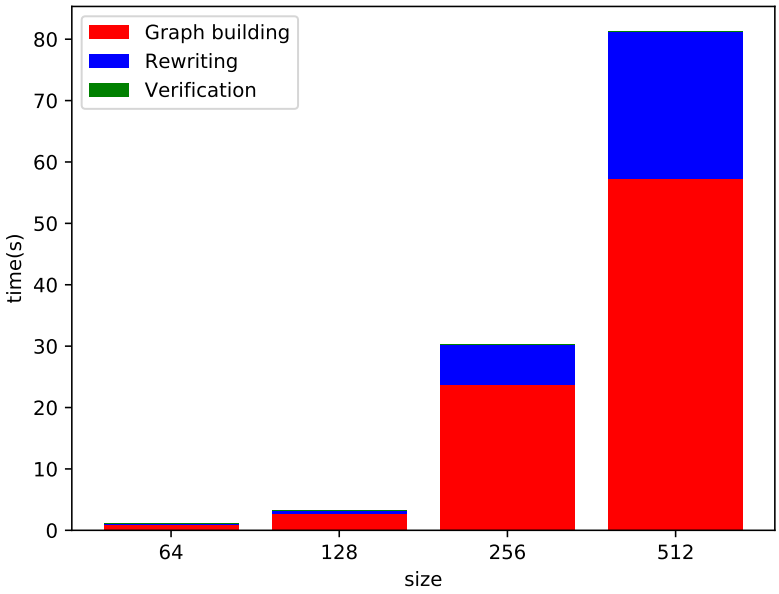


Figure 5.4: Timing analysis for verification of Karatsuba multiplier

5.3 String vs Graph

We have presented experiments results from string-based algorithm in chapter 3 and graph-based approach in chapter 4 in table 5.7 and plotted result in graph in 5.5. From the plot and tables, we can observe that the graph-based approach outperforms the string-based approach for all multipliers.

Table 5.7: String vs graph

size	Mastrovito (s)		Montgomery (s)		Karatsuba (s)	
	string	graph	string	graph	string	graph
16	0.82	0.62	0.99	0.63	0.79	0.6
32	2.52	1.31	1.29	0.98	1.21	0.91
64	10.31	2.87	3.46	2.25	2.71	1.69
128	12.98	13.17	32.47	6.65	7.59	3.98

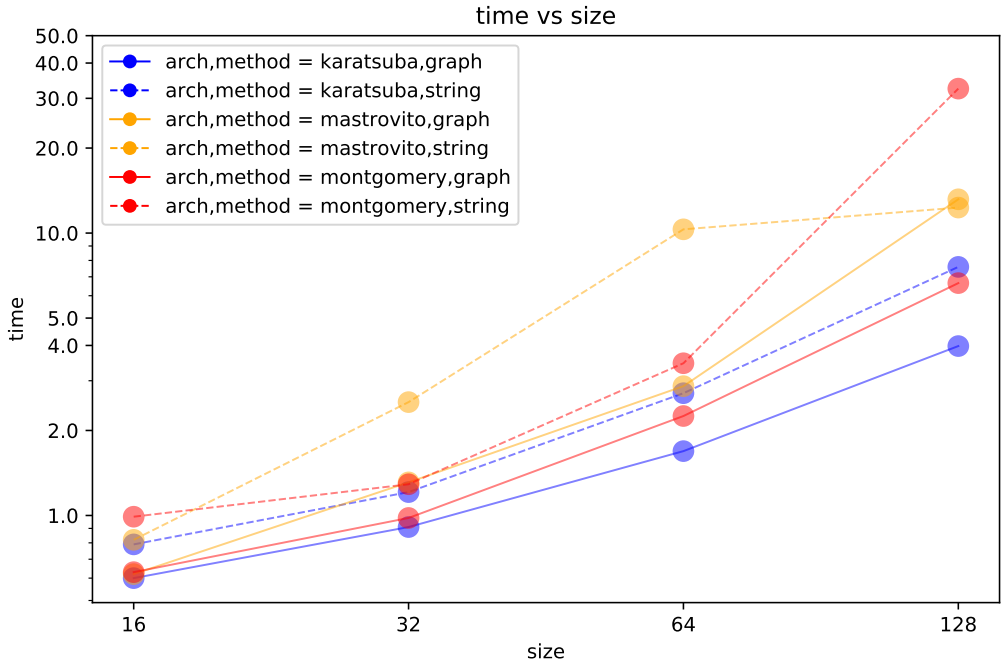


Figure 5.5: String vs Graph

5.4 Verification

In the string-based approach, the equivalence of output bit expressions can be verified in two ways namely string-based and SAT solver-based. Time taken by both approaches is presented in table 5.8. From the table 5.8, we can observe that the string-based approach is much faster then SAT solver based approach. We can observe that with an increase in size from 64 to 128, the time taken by the string-based approach is doubled but the time taken by SAT solver is three times for Mastrovito multiplier and 5 times for Montgomery multiplier.

Table 5.8: Verification based on string vs sat solver

size	Mastrovito (ms)		Montgomery (ms)	
	string	SAT solver	string	SAT solver
32	6	1060	6	1071
64	7	1147	9	1186
128	12	3878	10	5109

5.5 Analysis

Our graph-based approach outperforms Yu's [25] parallel algorithm with 30 threads because the common nodes within the model and between models are visited only once. Graph based approach outperforms the string-based algorithm although in string-based algorithm common nodes within the model are visited only once but it can't identify common sub-structure between two expressions.

Following key ideas in our algorithm. While traversing the graph, visit each node in the graph only once. The normal form which is developed in chapter 4 resembles the algebraic normal form. The normal form is a binary tree that extends only on the right side. Cubes in normal form are sorted from top to bottom. Two signals are semantically equivalent if their normal form is the same. Hash tables developed in chapter 4 prevent duplication of nodes in the graph and help to identify common nodes between two models. Equivalence verification of two expressions can be done by comparing node number which significantly takes less time than comparing two strings.

Following are the lessons learnt from our experiments. Recursion depth may exceeds limit when input is very large. So use custom stack which represents recursion stack in that case. In a string-based algorithm, we can't identify a common substructure between two expressions.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Verification of hardware implementation of finite field multipliers is important as they are involved in many security systems. Various methods were developed for the formal verification of finite field multipliers previously. But these methods have a limitation on the size of multipliers that they can process. The current state of art was developed by Yu [25] which involves parallel rewriting but common nodes in the model are computed multiple times.

We have developed two graph-based approaches for the formal verification of finite field multipliers. In these approaches, a graph is formed by parsing the netlist and symbolic expression for output bit is obtained by traversing the graph from output to inputs in a depth-first manner visiting every node only once. New data structures are developed for expressions which represents algebraic normal form. Various normalization methods are developed for the data structures. Various verification methods are developed to verify the symbolic expressions.

We have developed a string-based algorithm in chapter 3 in which expressions are stored as a list of strings. For Mastrovito multiplier of size 128, this approach achieves a speedup of 12× when compared to Yu's [25] algorithm of one thread and 4× in comparison to Yu's [25] with thirty threads. For Montgomery multiplier of size 128, this approach achieves a speedup of 10× when compared to Yu's [25] algorithm of one thread and 3× in comparison to Yu's [25] with thirty threads. This approach outperforms Yu's [25] method because the common nodes are computed only once. In this approach, since list of strings are used

to represent expressions, same sub-expressions may be repeated in many expressions. So when comparing two expressions, same sub-expression is traversed multiple times which takes time.

To overcome the drawbacks of a string-based algorithm, we have developed a graph-based approach which is introduced in chapter 4. In this approach, two hash tables are developed to prevent duplication of nodes in the graph. Normal form, a new representation for expression which represents algebraic normal form is developed. In this approach, the graph is built for both models together and the expression for output bits for both models is computed together by executing the rewriting step only once. For the Mastrovito multiplier of 283-bit size, this approach achieves 4× speedup for 30 threads and 14× speedup for one thread. For Montgomery multiplier of size 283, this approach achieves 256× speedup for 30 threads and 556× speedup for one thread. Yu's [25] algorithm is not able to verify Mastrovito and Montgomery multipliers of 571 bit with 30 threads but this approach is able to verify them. This approach outperforms both the string-based algorithm and Yu's [25] algorithm.

6.2 Future Work

The 571-bit Montgomery took longer time to verify than expected. One part of our future work is to explore impact of multiplier architecture and irreducible polynomial on time to normalize graph. Graph building algorithms for large circuits can be explored further in future. Since the developed algorithm runs in a single thread but implicitly running parallel as all the cores of computer are used, our future work includes extending our algorithm for explicit multi-threading on a machine with more cores. Currently, we have applied algorithm to only finite field multipliers, in future we want to extend to other combinational circuits. For multipliers, each cube in the final expression of output bit contains maximum of 2 inputs. But other combinational circuits may contain more and normalization could become expensive. Normalization is a critical step while obtaining an expression for an output bit. One doesn't know whether to do normalization at the current step or not. So, machine learning can be leveraged to make decisions.

References

[1] FDIV bug. https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html.

[2] Valentin Bakoev. Fast bitwise implementation of the algebraic normal form transform. *Serdica Journal of Computing*, pages 45–57, 2017.

[3] E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. *In Proceedings on Advances in Cryptology*, pages 221–240, 2008.

[4] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM 7*, page 201–215, 1960.

[6] A. Mishchenko et al. Abc: A system for sequential synthesis andverification. *[Online]. Available: http://www.eecs.berkeley.edu/alanmi/abc*, 2007.

[7] C. Barrett et al. CVC4. *Computer Aided Verification (CAV)*, pages 171–177, 2011.

[8] Farimah Farahmandi and Bijan Alizadeh. Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *Microprocessors and Microsystems*, 39(2):83–96, 2015.

[9] Moore GE. Progress in digital integrated electronics. *In: Proceedings of the IEEE electron devices meeting*, 21:21–25, 1975.

[10] K. Koc and T. Acar. Montgomery multiplication in $GF(2^k)$. *Designs Codes Cryptograph.*, 14(1):57–69, 1998.

[11] Jinpeng Lv, Priyank Kalla, and Florian Enescu. Efficient gröbner basis reductions for formal verification of galois field arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(9):1409–1420, 2013.

[12] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. Polycleaner: Clean your polynomials before backward rewriting to verify million-gate multipliers. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[13] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. Revsca: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[14] Leonardo De Moura and Nikolaj Bjrner. Z3: An efficient smt solver. *Springer*, 2008.

[15] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[16] Maria Pashinska-Gadzheva, Valentin Bakoev, Iliya Bouyukliev, and Dushan Bikov. Optimizations in computing the algebraic normal form transform of Boolean functions. In *2021 International Conference Automatics and Informatics (ICAI)*, pages 288–291. IEEE.

[17] K. Prasad and B. Sundar Rajan. Network-error correcting codes using small fields. *IEEE Transactions on Communications*, 62(2):423–433, 2014.

[18] Amr Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1048–1053, 2016.

[19] Xiaojun Sun, Priyank Kalla, Tim Pruss, and Florian Enescu. Formal verification of sequential galois field arithmetic circuits using algebraic geometry. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition*, DATE '15, page 1623–1628, San Jose, CA, USA, 2015. EDA Consortium.

[20] B. Sunar and Ç. K. Koç. Mastrovito multiplier for all trinomials. *IEEE Trans. Comput.*, 48(5):522–527, 1999.

[21] Niklas Sörensson and Niklas Een. Minisat v1.13 - a sat solver with conflict-clause minimization. 2005. sat-2005 poster. 1 perhaps under a generous notion of "part-time", but still concurrently taking a statistics course and leading a normal life. Technical report, 2002.

[22] Ou Wang, Dajin Wang, and Jianguo Yu. Network coding for linear finite-field deterministic network. In *2012 International Conference on Control Engineering and Communication Technology*, pages 874–879, 2012.

[23] André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptol. ePrint Arch.*, 2006:224, 2006.

[24] Cunxi Yu, Walter Brown, Duo Liu, André Rossi, and Maciej Ciesielski. Formal verification of arithmetic circuits by function extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):2131–2142, 2016.

[25] Cunxi Yu and Maciej J. Ciesielski. Formal analysis of galois field arithmetics - parallel verification and reverse engineering. *CoRR*, abs/1802.06870, 2018.