WORD-LEVEL TECHNIQUES FOR ABSTRACTION AND VERIFICATION OF SEQUENTIAL CIRCUITS USING ALGEBRAIC GEOMETRY

by

Xiaojun Sun

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

Dec 2016

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Xiaojun Sun

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Priyank Kalla

Ganesh Gopalakrishnan

Chris J. Myers

Kenneth S. Stevens

Rongrong Chen

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of _____Xiaojun Sun_____ in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

____________________ ________________________________

Date Priyank Kalla
Chair, Supervisory Committee

Approved for the Major Department

____

Gianluca Lazzi
Chair/Dean

Approved for the Graduate Council

____

David B. Kieda
Dean of The Graduate School

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Verification of sequential circuits remains to be a topic of all time. With increasing size of new integrated circuits, sequential circuit designers may face much more complicated problems on logic errors, unexpected delays and failures on critical path. This dissertation addresses the problem of sequential circuit verification at the word-level and is based on concepts derived from algebraic geometry.

Analyzing the sequential circuits at word level is an efficient way of *abstraction*, which may lower the complexity of verification by efficient representation of the state-space. We manage to model the verification properties and the gate-level sequential circuit implementations over Galois fields of the type $\mathbb{F}_{2^k}$ by means of polynomial ideals and their canonical representations — Gröbner bases — at the level of k-bit words. Subsequently, techniques from algebraic geometry can be used to reason about the state-space on high-level. In algebraic geometry, "bad" states are modeled as varieties while whole system is described using polynomial ideals. Without actually solving the system, the states in state space can be investigated by manipulating these ideals.

We propose to apply these techniques to traverse a finite state machine (FSM) for reachability analysis at the word-level, and also to implicitly unroll sequential arithmetic circuits to verify their function. Moreover, as unsatisfiable (UNSAT) cores play an important role in modern abstraction-refinement techniques for verification, we propose to investigate a word-level analogue of the UNSAT core problem using the Gröbner basis algorithm. The dissertation will not only derive new algorithms and techniques, but will also consider efficient CAD implementations to target sequential equivalence and model checking problems.

CHAPTER 1

INTRODUCTION

1.1 Design, Test and Verification of Modern Sequential Circuit Components

1.2 Formal Verification Techniques

1.2.1 Conventional Formal Verification

1.2.2 Word-level Verification Techniques

1.3 Objective, Motivation and Contribution of this Dissertation

1.3.1 Word-level Reachability and FSM Traversal

1.3.2 Application to Galois Field Multipliers

1.3.3 Abstractions for Sequential Circuits

1.3.4 Other Applications

1.4 Dissertation Organization

CHAPTER 2

PREVIOUS WORK

2.1 SAT,DD,AIG-based Sequential Circuit Verification

2.2 Bounded Model Checking

2.3 Model Checking with Abstraction Refinement

2.4 Word-level Techniques in Sequential Circuit Verification

Term rewriting?

2.5 Verification of Galois Field Circuits

2.6 Conventional UNSAT Core Extraction

2.7 Conclusion Remarks

CHAPTER 3

GALOIS FIELD AND SEQUENTIAL ARITHMETIC CIRCUITS

This chapter provides a mathematical background for understanding finite fields (Galois fields) and explains how to design Galois field arithmetic circuits. We first introduce the mathematical concepts of groups, rings, fields, and polynomials. We then apply these concepts to create Galois field arithmetic functions and explain how to map them to a Boolean circuit implementation. Additonally, we introduce a special type of sequential arithmetic hardware based on normal basis, as well as the normal basis theory behind the designing such hardware.

The material is referred from [1–3] for Galois field concepts, [4–8] for hardware design over Galois fields and previous work by *Lv* [9]. Normal basis theory in this section refers to [10, 11] and sequential normal basis arithmetic hardware designs come from [12–15].

3.1 Commutative Algebra

3.1.1 Group, ring and field

Definition 3.1 *An* **abelian group** *is a set* $\mathbb{S}$ *with a binary operation* $'+'$ *which satisfies the following properties:*

- *Closure Law: For every* $a, b \in \mathbb{S}, a + b \in \mathbb{S}$
- *Associative Law: For every* $a, b, c \in \mathbb{S}, (a + b) + c = a + (b + c)$
- *Commutativity: For every* $a, b \in \mathbb{S}, a + b = b + a$.
- *Additive Identity: There is an identity element* $0 \in \mathbb{S}$ *such that for all* $a \in \mathbb{S}$; $a + 0 = a$.

- *Additive Inverse: If $a \in \mathbb{S}$, then there is an element $a^{-1} \in \mathbb{S}$ such that $a + a^{-1} = 0$.*

The set of integers $\mathbb{Z}$ forms an abelian group under the addition operation.

Definition 3.2 *Given a set $\mathbb{R}$ with two binary operations, $'+'$ and $'\cdot'$, and element $0 \in \mathbb{R}$, the system $\mathbb{R}$ is called a* **commutative ring with unity** *if the following properties hold:*

- *$\mathbb{R}$ forms an abelian group under the '+' operation with additive identity element 0.*
- *Multiplicative Distributive Law: For all $a, b, c \in \mathbb{R}$, $a \cdot (b + c) = a \cdot b + a \cdot c$.*
- *Multiplicative Associative Law: For every $a, b, c \in \mathbb{R}$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.*
- *Multiplicative Commutative Law: For every $a, b \in \mathbb{R}$, $a \cdot b = b \cdot a$*
- *Identity Element: There exists an element $1 \in \mathbb{R}$ such that for all $a \in \mathbb{R}$, $a \cdot 1 = a = 1 \cdot a$*

Ring is a broad algebraic concept. In this dissertation, this word is used to refer a special sort of ring – **commutative ring with unity**. Two common examples of such rings are the set of integers, $\mathbb{Z}$, and the set of rational numbers, $\mathbb{Q}$. While both of these examples are rings with an infinite number of elements, the number of elements in a ring can also be finite, such as the ring of integers modulo n ($\mathbb{Z}_n$).

Definition 3.3 *A* **field** *$\mathbb{F}$ is a commutative ring with unity, where every non-zero element in $\mathbb{F}$ has a multiplicative inverse; i.e. $\forall a \in \mathbb{F} - \{0\}$, $\exists \hat{a} \in \mathbb{F}$ such that $a \cdot \hat{a} = 1$.*

A field is defined as a ring with one extra condition: the presence of a multiplicative inverse for all non-zero elements. Therefore, a field must be a ring while a ring is not necessarily a field. For example, the set $\mathbb{Z}_{2^k} = \{0, 1, \cdots, 2^k - 1\}$ forms a finite ring. However, $\mathbb{Z}_{2^k}$ is not a field because not every element in $\mathbb{Z}_{2^k}$ has a multiplicative inverse. In the ring $\mathbb{Z}_{2^3}$, for instance, the element 5 has an inverse ($5 \cdot 5 \pmod 8 = 1$) but the element 4 does not.

An important concept in field theory is **field extension**. The idea behind a field extension is to take a base field and construct a larger field which contains the base field as well as satisfies additional properties. For example, the set of real numbers $\mathbb{R}$ forms a field; one extension of $\mathbb{R}$ is the set of complex numbers $\mathbb{C} = \mathbb{R}(i)$. Every element of $\mathbb{C}$ can be represented as $a + b \cdot i$ where $a, b \in \mathbb{R}$, hence $\mathbb{C}$ is a two-dimensional extension of $\mathbb{R}$.

Like rings, fields can also contain either an infinite or a finite number of elements. In this dissertation we focus on finite fields – also known as Galois fields, the construction of their field extensions, and their applications on circuit verification and abstraction techniques.

3.1.2 Finite Field

Finite fields find widespread applications in many areas of electrical engineering and computer science such as error- correcting codes, elliptic curve cryptography, digital signal processing, testing of VLSI circuits, among others. In this dissertation, we specifically focus on their application to the FSM traversal of sequential Galois field circuits as well as abstraction refinement based on UNSAT core extraction. This section describes the relevant Galois field concepts [1] [2] [3] and hardware arithmetic designs over such fields [4] [5] [6] [7] [8].

Definition 3.4 *A* **Galois field***, denote $\mathbb{F}_q$, is a field with a finite number of elements, q. The number of elements q of the Galois field is a power of a prime integer, i.e. $q = p^k$, where p is a prime integer, and $k \geq 1$. Thus a Galois field can also be denoted as $\mathbb{F}_{p^k}$.*

Fields of the form $\mathbb{F}_{p^k}$ are called Galois extension fields. We are specifically interested in extension fields of type $\mathbb{F}_{2^k}$, where $k > 1$. These are extensions of the binary field $\mathbb{F}_2$.

Example 3.1 *Addition and multiplication operations over $\mathbb{F}_2$:*

Notice that addition over $\mathbb{F}_2$ is a Boolean XOR *operation, because it is performed modulo* 2*. Similarly, multiplication over $\mathbb{F}_2$ performs a Boolean* AND *operation.*

$+$	0	1
0	0	1
1	1	0

Addition over $\mathbb{F}_2$

$\cdot$	0	1
0	0	0
1	0	1

Multiplication over $\mathbb{F}_2$

Algebraic extensions of the binary field $\mathbb{F}_2$ are generally termed as *binary extension fields* $\mathbb{F}_{2^k}$. Where elements in $\mathbb{F}_2$ can only represent 1 bit, elements in $\mathbb{F}_{2^k}$ represent a k-bit vector. This allows them to be widely used in digital hardware applications. In order to construct a Galois field of the form $\mathbb{F}_{2^k}$, an **irreducible polynomial** is required:

Definition 3.5 *A polynomial $P(x) \in \mathbb{F}_2[x]$, i.e. the set of all polynomials in x with coefficients in $\mathbb{F}_2$, is* **irreducible** *if $P(x)$ is non-constant with degree k and cannot be factored into a product of polynomials of lower degree in $\mathbb{F}_2[x]$.*

Therefore, the polynomial $P(x)$ with degree k is irreducible over $\mathbb{F}_2$ if and only if it has no roots in $\mathbb{F}_2$, i.e if $\forall a \in \mathbb{F}_2$, $P(a) \neq 0$. For example, $x^2 + x + 1$ is an irreducible polynomial over $\mathbb{F}_2$ because it has no solutions in $\mathbb{F}_2$, i.e. $(0)^2 + (0) + 1 = 1 \neq 0$ and $(1)^2 + (1) + 1 = 1 \neq 0$ over $\mathbb{F}_2$. Irreducible polynomials exist for any degree ≥ 2 in $\mathbb{F}_2[x]$.

Given an irreducible polynomial $P(x)$ of degree k in the polynomial ring $\mathbb{F}_2[x]$, we can construct a binary extension field $\mathbb{F}_{2^k} \equiv \mathbb{F}_2[x] \pmod{P(x)}$. Let α be a root of $P(x)$, i.e., $P(\alpha) = 0$. Since $P(x)$ is irreducible over $\mathbb{F}_2[x]$, $\alpha \notin \mathbb{F}_2$. Instead, α is an element in $\mathbb{F}_{2^k}$. Any element $A \in \mathbb{F}_{2^k}$ is then represented as:

$$A = \sum_{i=0}^{k-1} (a_i \cdot \alpha^i) = a_0 + a_1 \cdot \alpha + \cdots + a_{k-1} \cdot \alpha^{k-1}$$

where $a_i \in \mathbb{F}_2$ are the coefficients and $P(\alpha) = 0$.

To better understand this field extension, compare its similarities to another commonplace field extension $\mathbb{C}$, the set of complex numbers. $\mathbb{C}$ is an extension of the field of real numbers $\mathbb{R}$ with an additional element $i = \sqrt{-1}$, which is an imaginary root in the algebraic closure of $\mathbb{R}$ – the closure is known as the field of complex numbers $\mathbb{C}$. Thus $i \notin \mathbb{R}$, rather $i \in \mathbb{C}$. Every element $A \in \mathbb{C}$ can be represented as:

$$A = \sum_{j=0}^{1}(a_j \cdot i^j) = a_0 + a_1 \cdot i \tag{3.1}$$

where $a_j \in \mathbb{R}$ are coefficients. Similarly, $\mathbb{F}_{2^k}$ is an extension of $\mathbb{F}_2$ with an additional element α, which is the "imaginary root" of an irreducible polynomial P in $\mathbb{F}_2[x]$.

Every element $A \in \mathbb{F}_{2^k}$ has a degree less than k because A is always computed modulo $P(x)$, which has degree k. Thus, $A \pmod{P(x)}$ can be of degree at most $k-1$ and at least 0. For this reason, the field $\mathbb{F}_{2^k}$ can be viewed as a k dimensional vector space over $\mathbb{F}_2$. The equivalent bit vector representation for element A is:

$$A = (a_{k-1}a_{k-2}\cdots a_0) \tag{3.2}$$

Example 3.2 *A 4-bit Boolean vector, $(a_3a_2a_1a_0)$ can be presented over $\mathbb{F}_{2^4}$ as:*

$$a_3 \cdot \alpha^3 + a_2 \cdot \alpha^2 + a_1 \cdot \alpha + a_0 \tag{3.3}$$

For instance, the Boolean vector 1011 is represented as the element $\alpha^3 + \alpha + 1$.

Example 3.3 *Let us construct $\mathbb{F}_{2^4}$ as $\mathbb{F}_2[x] \pmod{P(x)}$, where $P(x) = x^4 + x^3 + 1 \in \mathbb{F}_2[x]$ is an irreducible polynomial of degree $k = 4$. Let α be the root of $P(x)$, i.e. $P(\alpha) = 0$.*

Any element $A \in \mathbb{F}_2[x] \pmod{x^4 + x^3 + 1}$ has a representation of the type: $A = a_3x^3 + a_2x^2 + a_1x + a_0$ (degree < 4) where the coefficients $a_3, \dots, a_0$ are in $\mathbb{F}_2 = \{0, 1\}$. Since there are only 16 such polynomials, we obtain 16 elements in the field $\mathbb{F}_{2^4}$. Each element in $\mathbb{F}_{2^4}$ can then be viewed as a 4-bit vector over $\mathbb{F}_2$. Each element also has an exponential α representation. All three representations are shown in Table 3.1.

We can compute the polynomial representation from the exponential representation. Since every element is computed $\pmod{P(\alpha)} = \pmod{\alpha^4 + \alpha^3 + 1}$, we compute the element α^4 as

$$\alpha^4 \pmod{\alpha^4 + \alpha^3 + 1} = -\alpha^3 - 1 = \alpha^3 + 1 \tag{3.4}$$

Recall that all coefficients of $\mathbb{F}_{2^4}$ are in $\mathbb{F}_2$ where $-1 = +1$ modulo 2. The next element α^5 can be computed as

$$\alpha^5 = \alpha^4 \cdot \alpha = (\alpha^3 + 1) \cdot \alpha = \alpha^4 + \alpha = \alpha^3 + \alpha + 1 \tag{3.5}$$

Then α^6 can be computed as $\alpha^5 * \alpha$ and so on.

Table 3.1: Bit-vector, Exponential and Polynomial representation of elements in $\mathbb{F}_{2^4} = \mathbb{F}_2[x] \pmod{x^4 + x^3 + 1}$

$a_3a_2a_1a_0$	Exponential	Polynomial	$a_3a_2a_1a_0$	Exponential	Polynomial
0000	0	0	1000	α^3	α^3
0001	1	1	1001	α^4	$\alpha^3 + 1$
0010	α	α	1010	α^{10}	$\alpha^3 + \alpha$
0011	α^{12}	$\alpha + 1$	1011	α^5	$\alpha^3 + \alpha + 1$
0100	α^2	α^2	1100	α^{14}	$\alpha^3 + \alpha^2$
0101	α^9	$\alpha^2 + 1$	1101	α^{11}	$\alpha^3 + \alpha^2 + 1$
0110	α^{13}	$\alpha^2 + \alpha$	1110	α^8	$\alpha^3 + \alpha^2 + \alpha$
0111	α^7	$\alpha^2 + \alpha + 1$	1111	α^6	$\alpha^3 + \alpha^2 + \alpha + 1$

An irreducible polynomial can also be a primitive polynomial.

Definition 3.6 *A* **primitive polynomial** $P(x)$ *is a polynomial with coefficients in* $\mathbb{F}_2$ *which has a root* $\alpha \in \mathbb{F}_{2^k}$ *such that* $\{0,\ 1(= \alpha^{2^k-1}),\ \alpha,\ \alpha^2,\ \cdots,\ \alpha^{2^k-2}\}$ *is the set of all elements in* $\mathbb{F}_{2^k}$. *Here* α *is called a* **primitive element** *of* $\mathbb{F}_{2^k}$.

A primitive polynomial is guaranteed to generate all distinct elements of a finite field $\mathbb{F}_{2^k}$ while an arbitrary irreducible polynomial has no such guarantee. Often, there exists more than one irreducible polynomial of degree k. In such cases, any degree k irreducible polynomial can be used for field construction. For example, both x^3+x+1 and x^3+x^2+1 are irreducible in $\mathbb{F}_2$ and either one can be used to construct $\mathbb{F}_{2^3}$. This is due to the following:

Theorem 3.1 *There exist a* **unique** *field* $\mathbb{F}_{p^k}$*, for any prime* p *and any positive integer* k.

Theorem 3.1 implies that Galois fields with the same number of elements are **isomorphic** to each other up to the labeling of the elements.

Theorem 3.2 provides an important property for investigating solutions to polynomial equations in $\mathbb{F}_q$.

Theorem 3.2 $[Generalized\ Fermat's\ Little\ Theorem]$ *Given a Galois field* $\mathbb{F}_q$*, each element* $A \in \mathbb{F}_q$ *satisfies:*

$$\begin{aligned} A^q &\equiv A \\ A^q - A &\equiv 0 \end{aligned} \tag{3.6}$$

We can extend Theorem 3.2 to polynomials in $\mathbb{F}_q[x]$ as follows:

Definition 3.7 *Let $x^q - x$ be a polynomial in $\mathbb{F}_q[x]$. Every element $A \in \mathbb{F}_q$ is a solution to $x^q - x = 0$. Therefore, $x^q - x$ always vanishes in $\mathbb{F}_q$. Such polynomials are called* **vanishing polynomials** *of the field $\mathbb{F}_q$.*

Example 3.4 *Given $\mathbb{F}_{2^2} = \{0, 1, \alpha, \alpha + 1\}$ with $P(x) = x^2 + x + 1$, where $P(\alpha) = 0$.*

$$\begin{aligned} 0^{2^2} &= 0 \\ 1^{2^2} &= 1 \\ \alpha^{2^2} &= \alpha \pmod{\alpha^2 + \alpha + 1} \\ (\alpha + 1)^{2^2} &= \alpha + 1 \pmod{\alpha^2 + \alpha + 1} \end{aligned}$$

A Galois field $\mathbb{F}_q$ can be fully contained within a larger field $\mathbb{F}_{q^k}$. That is, $\mathbb{F}_q \subset \mathbb{F}_{q^k}$. For example, the containment relation of the fields $\mathbb{F}_2 \subset \mathbb{F}_{2^k}$ is usually used to represent bit-level Boolean variables as field elements in larger finite field which allows projection of k-bit word-level variables. Concretely, $\mathbb{F}_{16} = \mathbb{F}_{4^2} = \mathbb{F}_{2^4}$ contains $\mathbb{F}_4$ and $\mathbb{F}_2$. The elements $\{0, 1, \alpha, \ldots, \alpha^{14}\}$ designate $\mathbb{F}_{16}$. Of these, $\{0, 1, \alpha^5, \alpha^{10}\}$ create $\mathbb{F}_4$. From these, only $\{0, 1\}$ exist in $\mathbb{F}_2$.

Theorem 3.3 *$\mathbb{F}_{2^n} \subset \mathbb{F}_{2^m}$ iff $n \mid m$, i.e. if n divides m.*

Therefore:

- $\mathbb{F}_2 \subset \mathbb{F}_{2^2} \subset \mathbb{F}_{2^4} \subset \mathbb{F}_{2^8} \subset \ldots$
- $\mathbb{F}_2 \subset \mathbb{F}_{2^3} \subset \mathbb{F}_{2^9} \subset \mathbb{F}_{2^{27}} \subset \ldots$
- $\mathbb{F}_2 \subset \mathbb{F}_{2^5} \subset \mathbb{F}_{2^{25}} \subset \mathbb{F}_{2^{125}} \subset \ldots$, and so on

Definition 3.8 *The* **algebraic closure** *of the Galois field $\mathbb{F}_{2^k}$, denoted $\overline{\mathbb{F}_{2^k}}$, is the union of all fields $\mathbb{F}_{2^n}$ such that $k \mid n$.*

3.2 Normal Basis Multiplier over Galois Field

From algebraic view, a field is a space, and field elements are points in the space. Those elements can be represented with unique coordinates, which requires the pre-definition of a basis vector. In this section, we discuss a special basis called normal basis, as well as the advantages adopting it in GF operations esp. multiplication.

3.2.1 Normal Basis

Given a Galois field (GF) $\mathbb{F}_{2^k}$ is a finite field with 2^k elements and characteristic equals to 2. Its elements can be written in polynomials of α, when there is an irreducible polynomial $p(\alpha)$ defined.

If we use a basis $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{k-1}\}$, we can easily transform polynomial representations to binary bit-vector representations by recording the coefficients. For example, for elements in $\mathbb{F}_{2^4}$, the results are shown in table 3.1, column "Polynomial".

Basis $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{k-1}\}$ is called **standard basis** (StdB), which results in a straightforward representation for elements, and operations of elements such as addition and subtraction. The addition/subtraction of GF elements in StdB follows the rules of polynomial addition/subtraction where coefficients belong to $\mathbb{F}_2$. In other words, using the definition of *exclusive or* (XOR) in Boolean algebra, element A add/subtract by element B in StdB is defined as

$$\begin{aligned} A + B = A - B &= (a_0, a_1, \ldots, a_{k-1})_{StdB} \bigoplus (b_0, b_1, \ldots, b_{k-1})_{StdB} \\ &= (a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{k-1} \oplus b_{k-1})_{StdB} \end{aligned} \tag{3.7}$$

3.2.2 Multiplication using Normal Basis

Besides addition/subtraction, multiplication is also very common in arithmetic circuit design. The multiplication of GF elements in $\mathbb{F}_{2^k}$ in StdB follows the rule of polynomial multiplication. However, it will result in $O(k^2)$ bitwise operations. In other words, if we implement GF multiplication in bit-level logic circuit, it will contain $O(k^2)$ gates. When the datapath size k is large, the area and delay of circuit will be costly.

In order to lower down the complexity of arithmetic circuit design, Massey and Omura [13] use a new basis to represent GF elements, which is called **normal basis** (NB). A normal basis over $\mathbb{F}_{2^k}$ is written in the form of

$$N.B.\ \ \mathcal{N} = \{\beta, \beta^2, \beta^4, \beta^8, \dots, \beta^{2^{k-1}}\}$$

Normal element β is an element from the field which is used to construct the normal basis, and can be represent as a power of primitive element α:

$$\beta = \alpha^t, \quad 1 \leq t < 2^k$$

Exponent t takes value in the given range when $\mathcal{N}$ fulfills the definition of a basis.

Respectively, a field element in NB representation is actually

$$\begin{aligned} A &= (a_0, a_1, \dots, a_{k-1})_{NB} \\ &= a_0\beta + a_1\beta^2 + \dots + a_{k-1}\beta^{2^{k-1}} \\ &= \sum_{i=0}^{k-1} a_i \beta^{2^i} \end{aligned}$$

According to the definition, a normal basis is a vector where the next entry is the square of the former one. We note that the vector is cyclic, i.e. $\beta^{2^k} = \beta$ due to *Fermat's little theorem*.

The addition and subtraction of elements in NB representation are similar to equation 3.7. However, what makes NB powerful is its ease of implementation when doing multiplications and exponentiations. The following lemmas and examples illustrate this fabulous property very well.

Lemma 3.1 (Square of NB) *In* $\mathbb{F}_{2^k}$*,*

$$(a + b)^2 = a^2 + b^2$$

According to the ***binomial theorem****, it can be extended to*

$$\begin{aligned} \beta^2 =& (b_0\beta + b_1\beta^2 + b_2\beta^4 + \dots + b_{k-1}\beta^{2^{k-1}})^2 \\ =& b_0^2\beta^2 + b_1^2\beta^4 + b_2^2\beta^8 + \dots + b_{k-1}^2\beta \\ =& b_{k-1}^2\beta + b_0\beta^2 + b_1\beta^4 + \dots + b_{k-2}\beta^{2^{k-1}} \end{aligned}$$

This lemma concludes that the square of an element in NB equals to a simple right-cyclic shift of the bit-vector. Obviously, StdB representation does not have this benefit.

Example 3.5 (Square of NB) *In GF $\mathbb{F}_{2^3}$ constructed by irreducible polynomial x^3+x+1, the standard basis is denoted as $\{1, \alpha, \alpha^2\}$ where $\alpha^3+\alpha+1 = 0$. Let $\beta = \alpha^3$, then $\mathcal{N} = \{\beta, \beta^2, \beta^4\}$ forms a normal basis. Write down element E using both representations:*

$$\begin{aligned} E &= (a_0, a_1, a_2)_{StdB} = (b_0, b_1, b_2)_{NB} \\ &= a_0 + a_1\alpha + a_2\alpha^2 = b_0\beta + b_1\beta^2 + b_2\beta^4 \end{aligned}$$

Compute the square of E in StdB first:

$$\begin{aligned} E^2 &= a_0 + a_1\alpha^2 + a_2\alpha^4 \\ &= a_0 + a_2\alpha + (a_1 + a_2)\alpha^2 \\ &= (a_0, a_2, a_1 + a_2)_{StdB} \end{aligned}$$

When it is computed in NB, we can make it very simple:

$$\begin{aligned} E^2 &= \overset{\xrightarrow{Cyclic \quad shift}}{(b_0, b_1, b_2)}_{NB} \\ &= (b_2, b_0, b_1)_{NB} \end{aligned}$$

This example shows that convenience to use NB when computing 2^k power of an element. Multiplication is a bit complicated than squaring; but when it is decomposed as bit-wise operations, the property in lemma 3.1 can be well utilized.

Example 3.6 (Bit-wise NB multiplication) *Assume there are 2 binary vectors representing 2 operands in NB over $\mathbb{F}_{2^k}$: $A = (a_0, a_1, \ldots, a_{k-1}), B = (b_0, b_1, \ldots, b_{k-1})$. Note that in this example, by default we use normal basis representation so subscript "NB" is skipped. Their product can also be written as:*

$$C = A \times B = (c_0, c_1, \ldots, c_{k-1})$$

Assume the most significant bit (MSB) of the product can be represented by a function f_{mult}:

$$c_{k-1} = f_{mult}(a_0, a_1, \ldots, a_{k-1}; b_0, b_1, \ldots, b_{k-1}) \tag{3.8}$$

Before discussing the details of the function f_{mult}, we can take a square on both side of equation 3.8, i.e. $C^2 = A^2 \times B^2$. Obviously, using the property in lemma 3.1, *the original second most significant bit becomes the new MSB because of right-cyclic shifting. Concretely,*

$$(c_{k-1}, c_0, c_1, \ldots, c_{k-2}) = (a_{k-1}, a_0, a_1, \ldots, a_{k-2}) \times (b_{k-1}, b_0, b_1, \ldots, b_{k-2})$$

Note A^2, B^2 and C^2 still belong to $\mathbb{F}_{2^k}$, thus as a universal function implementing MSB multiplication over $\mathbb{F}_{2^k}$, f_{mult} still remains the same. As a result, the new MSB can be written as

$$c_{k-2} = f_{mult}(a_{k-1}, a_0, a_1, \ldots, a_{k-2}; b_{k-1}, b_0, b_1, \ldots, b_{k-2}) \tag{3.9}$$

Similarly, if we take a square again on the new equation, we can get c_{k-3}. Successively we can derive all bits of product C using the same function f_{mult}, and the only adjustment we need to make is to right-cyclic shift 2 operands by 1 bit each time.

From above example, it is known that a universal structure that implements f_{mult} can be reused k times in NB multiplication over $\mathbb{F}_{2^k}$. Compared to StdB, which requires a distinct design for every bit of multiplication, NB is less costly – as long as we can prove f_{mult} implies a structure with $o(k^2)$ complexity (symbol o denotes "strictly lower than bound").

If we want to make the complexity of f_{mult} lower than $O(k^2)$, then the best choice is to try out linear functions. As we know, matrix multiplication can simulate all possible combinations of linear functions. Imagine A is a k-bit row vector and B is a k-bit column vector, then the single bit product can be written as the product of matrix multiplication

$$c_l = A \times M \times B$$

where

$$A = (a_0, a_1, \ldots, a_{k-1})$$
$$B = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{k-1} \end{pmatrix}$$

Moreover, M is a $k \times k$ square matrix. If we can find M, we obtain the design of the multiplier.

Definition 3.9 (λ-Matrix) *A binary $k \times k$ matrix M is used to describe the bit-wise normal basis multiplication function f_{mult} where*

$$c_l = f_{mult}(A, B) = A \times M \times B^T \tag{3.10}$$

Symbol B^T denotes vector transposition. Matrix M is called λ-Matrix of k-bit NB multiplication over $\mathbb{F}_{2^k}$.

When taking different bits l of the product in equation 3.10, we obtain a series of conjugate matrices of M. Which means instead of shifting operands A and B, we can also shift the matrix.

More specifically, we denote the matrix by *l-th λ-Matrix* as

$$c_l = A \times M^{(l)} \cdot B^T$$

Meanwhile, the operator shifting rule in equation 3.9 still holds. Then we have relation

$$c_{l-1} = A \cdot M^{(l-1)} \cdot B^T = shift(A) \cdot M^{(l)} \cdot shift(B)^T$$

which means by right and down cyclically shifting $M^{(l-1)}$, we can get $M^{(l)}$.

Example 3.7 (NB multiplication using λ-Matrix) *Over GF $\mathbb{F}_{2^3}$ constructed by irreducible polynomial $\alpha^3 + \alpha + 1$, let normal element $\beta = \alpha^3$, $N = \{\beta, \beta^2, \beta^4\}$ forms a normal basis. Corresponding 0-th λ-Matrix is*

$$M^{(0)} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

i.e.,

$$c_0 = (a_0\ a_1\ a_2) \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

From 0-th λ-Matrix we can directly write down all remaining λ-Matrices:

$$M^{(1)} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \qquad M^{(2)} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

If we generalize the definition and explore the nature of λ-Matrix, it is defined as cross-product terms from multiplication, which is

$$Product\ vector\ C = (\sum_{i=0}^{k-1} a_i \beta^{2^i})(\sum_{j=0}^{k-1} b_j \beta^{2^j}) = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} a_i b_j \beta^{2^i}\beta^{2^j} \tag{3.11}$$

The expressions $\beta^{2^i}\beta^{2^j}$ are referred to as cross-product terms, and can be represented by NB, i.e.

$$\beta^{2^i}\beta^{2^j} = \sum_{l=0}^{k-1} \lambda_{ij}^{(l)} \beta^{2^l}, \ \ \lambda_{ij}^{(l)} \in \mathbb{F}_2. \tag{3.12}$$

Substitution yields, result is an expression for l-th digit of product as showed in equation 3.8:

$$c_l = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \lambda_{ij}^{(l)} a_i b_j \tag{3.13}$$

$\lambda_{ij}^{(l)}$ is the entry with coordinate (i, j) in l-th λ-Matrix.

The λ-Matrix can be implemented with XOR and AND gates in circuit design. The very naive implementation requires $O(C_N)$ gates, where C_N is the number of nonzero entries in λ-Matrix. There usually exists multiple NBs in $\mathbb{F}_{2^k}, k > 3$. If we employ a random NB, there is no mathematical guarantee that C_N has bound $o(k)$. However, Mullin et al. [12] proves that in certain GF $\mathbb{F}_{p^{k_{opt}}}$, there always exists at least one NB such that its corresponding λ-Matrix has $C_N = 2n - 1$ nonzero entries. A basis with this property is called optimal normal basis (ONB), details are introduced in appendix A.2.

In practice, large size NB multipliers are usually designed in $\mathbb{F}_{2^k}$ when ONB exists to minimize the number of gates. So in the following part of this chapter and our experiments, we only focus on ONB multipliers instead of general NB multipliers.

3.2.3 Comparison between Standard Basis and Normal Basis

At the end of this section, a detailed example is used to make a comparison between StdB multiplication and NB multiplication.

Example 3.8 (Rijndael's finite field) *Rijndael uses a characteristic 2 finite field with 256 elements, which can also be called the GF $\mathbb{F}_{2^8}$. Let us define the primitive element α using irreducible polynomial $\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + \alpha^2 + \alpha + 1$. Coincidently, α is also a normal element, i.e. $\beta = \alpha$ can construct a NB $\{\alpha, \alpha^2, \alpha^4, \alpha^8, \alpha^{16}, \alpha^{32}, \alpha^{64}, \alpha^{128}\}$.*

We pick a pair of elements from the Rijndael's field: $A = (0100\ 1011)_{StdB} = (4B)_{StdB},\ B = (1100\ 1010)_{StdB} = (CA)_{StdB}$. *First let us compute their product in StdB, the rule follows ordinary polynomial multiplication.*

$$\begin{aligned} A \cdot B &= (\alpha^6 + \alpha^3 + \alpha + 1)(\alpha^7 + \alpha^6 + \alpha^3 + \alpha) \\ &= (\alpha^{13} + \alpha^{10} + \alpha^8 + \alpha^7) + (\alpha^{12} + \alpha^9 + \alpha^7 + \alpha^6) + (\alpha^9 + \alpha^6 + \alpha^4 + \alpha^3) \\ &\quad + (\alpha^7 + \alpha^4 + \alpha^2 + \alpha) \\ &= \alpha^{13} + \alpha^{12} + \alpha^{10} + \alpha^8 + \alpha^7 + \alpha^3 + \alpha^2 + \alpha \end{aligned}$$

Note that this polynomial is not the final form of the product because it needs to be reduced modulo irreducible polynomial $\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + \alpha^2 + \alpha + 1$. *This can be done using base-2 long division. Note the dividend and divisor are written in pseudo Boolean vectors, not real Boolean vectors in any kind of bases.*

```
                    101001
111010111 ) 11010110001110
            111010111
              111101101
              111010111
                 111010110
                 111010111
                         1
```

The final remainder is 1, *i.e. the product equals to 1 in StdB.*

On the other hand, operands A *and* B *can be written in NB as*

$$A = (0010\ \ 1001)_{NB}, \quad B = (0100\ \ 0010)_{NB}$$

The λ*-Matrix for* $\mathbb{F}_2[x] \pmod{x^8+x^7+x^6+x^4+x^2+x+1}$ *is (Computation of* λ*-Matrix refers to Appendix A.1)*

$$M^{(0)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Taking matrix multiplication $c_0 = A \times M^{(0)} \times B^T$, the result is $c_0 = 1$. Then by cyclic shifting A and B (or shifting $M^{(0)}$, either is applicable), we can successively obtain other bits of product. The final answer is

$$C = (0000\ \ 0001)_{NB}$$

It is equivalent to the result in StdB.

Mastrovito multiplier [4] and Montgomery multiplier [5] are 2 common designs of GF multipliers using StdB. As a naive implementation of GF multiplication, Mastrovito multiplier uses most number of gates: k^2 AND gates and less than k^2 XOR gates [16]. Montgomery multiplier applies lazy reduction techniques and results in a better latency performance, while the number of gates are about the same with Mastrovito multiplier: k^2 AND gates and $k^2 - k/2$ XOR gates [7].

For an 8-bit ($\mathbb{F}_{2^8}$) multiplier, typical design of Mastrovito multiplier consists of 218 logic gates, while Montgomery multiplier needs 198 gates. However, the NB multiplier reuses the λ-Matrix logic, so this component will only need to be implemented for once. Consider the definition of matrix multiplication, it needs C_N AND gates to apply bit-wise multiplication and $C_N - 1$ XOR gates to sum the intermediate products up. The number of nonzero entries in the λ-Matrix can be counted: $C_N = 27$. As a result, the most naive NB multiplier design (or Massey-Omura multiplier [13]) contains 53 gates in total, which is a great saving in area cost comparing to StdB multipliers.

3.3 Design a Normal Basis Multiplier on Gate Level

The NB multiplier design consumes much less gates than ordinary StdB multiplier design, even if we use the most naive design. However, the modern NB multiplier design has been improved a lot from the very first design model proposed by Massey and Omura in 1986 [13]. In order to test our approach on practical contemporary circuits, it is necessary to learn the mechanism and design routine of several kinds of modern NB multipliers.

3.3.1 Sequential Multiplier with Parallel Outputs

The major benefit of NB multiplier origins from the sequential design. A straightforward design implementing the cyclic-shift of operands and λ-Matrix logic component is the Massey-Omura multiplier.

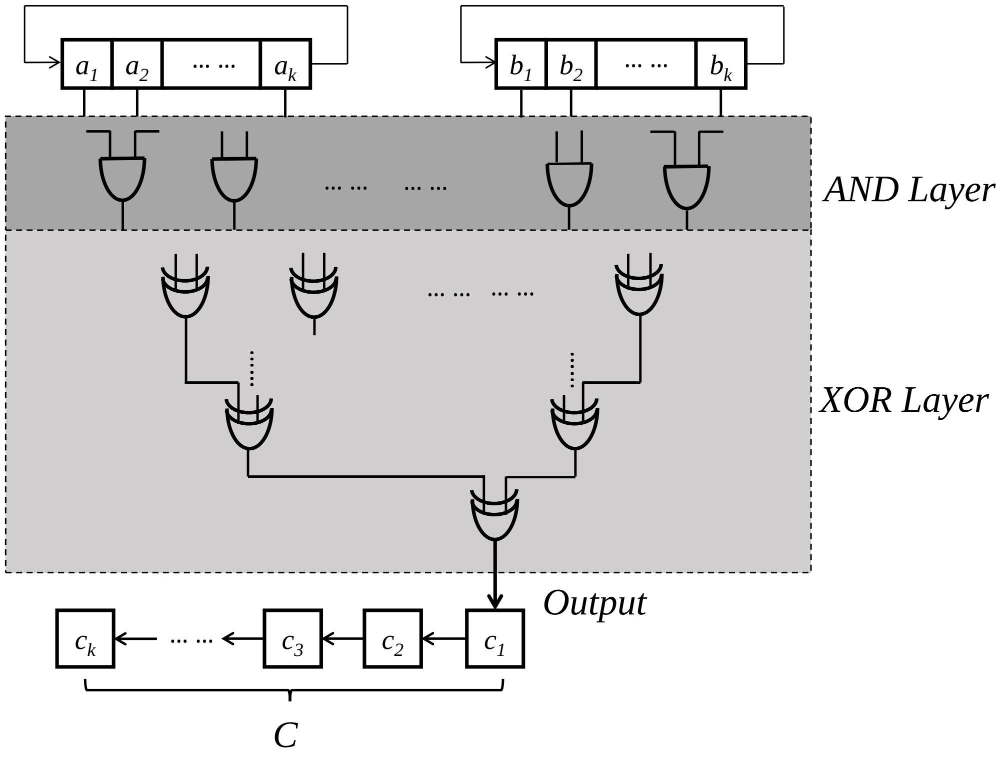


Figure 3.1: A typical SMSO structure of Massey-Omura multiplier

Figure 3.1 shows the basic architecture of a Massey-Omura multiplier. The operands A and B are 2 arrays of flip-flops which allow 1-bit right-cyclic shift every clock cycle. The logic gates in the boxes implements the matrix multiplication with λ-Matrix $M^{(0)}$, while each AND gate corresponds to term a_ib_j and each XOR gate corresponds to addition $a_ib_j + a_{i'}b_{j'}$. The XOR layer has only 1 output, giving out 1 bit of product C every clock cycle.

The behavior of Massey-Omura multiplier can be concluded as: pre-load operands A, B and reset C to 0, after executing for k clock cycles, the data stored in flip-flop array C is the product $A \times B$. We note that there is only one output giving 1 bit of the product each clock cycle, which matches the definition of serial output to communication channel. Therefore this type of design is named as sequential multiplier with serial output (SMSO). The SMSO architecture need C_N AND gates and $C_N - 1$ XOR gates, which equals to $2k-1$ AND gates and $2k-2$ XOR gates if it is designed using ONB. In fact, the number of gates can be reduced if the multiplication is implemented using a conjugate of SMSO.

The gate-level logic boxes are implementing following function:

$$c_l = row_1(A \times M^{(l)}) \times B + row_2(A \times M^{(l)}) \times B + \cdots + row_k(A \times M^{(l)}) \times B \quad (3.14)$$

It can be decomposed into k terms. If we only compute one term for each $c_l,\ 0 \leq l \leq k-1$ in one clock cycle, make k outputs and add them up using shift register after k clock cycles, it will generate the same result with SMSO. This kind of architecture is named as sequential multiplier with parallel outputs (SMPO). The basic SMPO, as a conjugate of Massey-Omura multiplier, is invented by Agnew et al. [14].

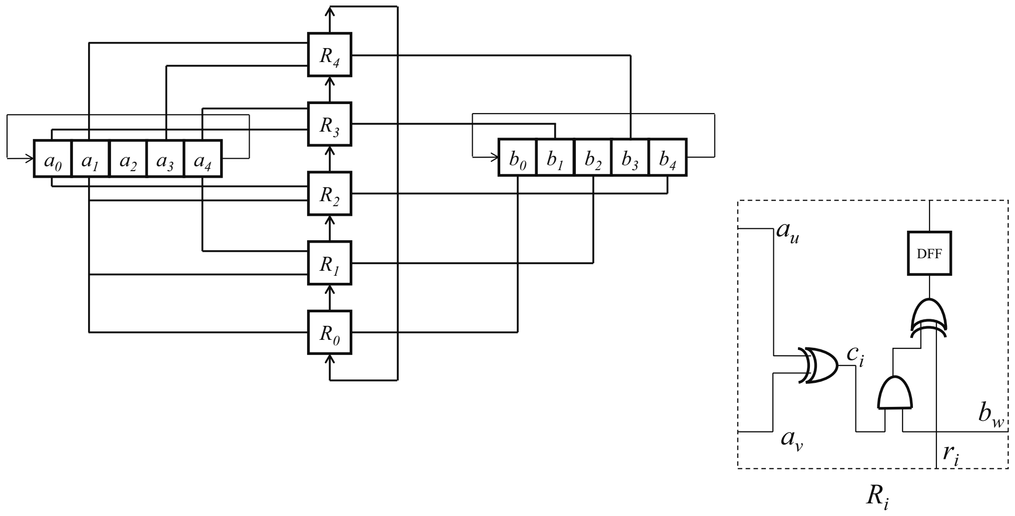


Figure 3.2: 5-bit Agnew's SMPO. Index i satisfies $0 < i < 4$, indices u, v are determined by column # of nonzero entries in i-th row of λ-Matrix $M^{(0)}$, i.e. if entry $M_{ij}^{(0)}$ is a nonzero entry, u or v equals to $i + j \pmod 5$. Index $w = 2i \pmod 5$

Example 3.9 (5-bit Agnew's SMPO) *Given GF $\mathbb{F}_{2^5}$ and primitive element α defined by irreducible polynomial $\alpha^5 + \alpha^2 + 1 = 0$, normal element $\beta = \alpha^5$ constructs an ONB $\{\beta, \beta^2, \beta^4, \beta^8, \beta^{16}\}$. The 0-th λ-Matrix for this ONB is*

$$M^{(0)} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Then a typical design of 5-bit Agnew's SMPO is depicted in Figure 3.2.

The operands part of this circuit is the same with Massey-Omura multiplier. The differences are on the matrix multiplication part, while it is implemented as separate logic blocks for 5 outputs, and the 5 blocks are connected in a shift register fashion. By analyzing the detailed function of logic blocks, we can reveal the mechanism of Agnew's SMPO.

Suppose we implement $M^{(0)}$ as the logic block in SMSO. In the first clock cycle, the output is

$$c_0 = a_1 b_0 + (a_0 + a_3) b_1 + (a_3 + a_4) b_2 + (a_1 + a_2) b_3 + (a_2 + a_4) b_4 \tag{3.15}$$

Note it is written in the form of Equation 3.14. In next clock cycles we can obtain remaining bits of the product, which can be written in following general form polynomial:

$$\begin{aligned} c_i =& b_i a_{i+1} + b_{i+1}(a_i + a_{i+3}) + b_{i+2}(a_{i+3} + a_{i+4}) \\ &+ b_{i+3}(a_{i+1} + a_{i+2}) + b_{i+4}(a_{i+2} + a_{i+4}),\ 0 \leq i \leq 4 \end{aligned}$$

Note all index calculations are reduced modulo 5.

Now let us observe the behavior of 5-bit Agnew's SMPO. Initially all DFFs are reset to 0. In the first clock cycle, signal sent to the flip-flop in block R_0 denotes function:

$$R_0^{(1)} = a_1 b_0$$

It equals to the first term of Equation 3.15. In the second clock cycle, this signal is sent to block R_1 through wire r_0, and this block also receives data from operands (shifted by 1 bit), generating signal a_u, a_v and b_w. Concretely, signal sent to flip-flop in block R_1 is:

$$R_1^{(2)} = R_0^{(1)} + (a_0 + a_3) b_1 = a_1 b_0 + (a_0 + a_3) b_1$$

which forms first 2 terms of Equation 3.15. Similarly, we track the signal on R_2 in third clock cycle, signal on R_3 in fourth clock cycle, finally we can get

$$R_4^{(5)} = a_1 b_0 + (a_0 + a_3) b_1 + (a_3 + a_4) b_2 + (a_1 + a_2) b_3 + (a_2 + a_4) b_4$$

which equals to c_0 in Equation 3.15. After the fifth clock cycle ends, this signal can be detected on wire r_0. It shows that the result of c_0 is computed after 5 clock cycles and given on r_0.

If we track $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \cdots \rightarrow R_0$, we can obtain c_1 respectively. Thus we conclude that Agnew's SMPO functions the same with Massey-Omura multiplier.

The design of Agnew's SMPO guarantees that there is only one AND gate in each R_i block. For ONB, adopting Agnew's SMPO will reduce the number of AND gates from $2k-1$ to k.

3.3.2 Multiplier not based on λ-Matrix

Both Massey-Omura multiplier and Agnew's SMPO rely on the implementation of λ-Matrix, which means that they will be identical if unrolled to full combinational circuits. After Agnew's work of parallelization, researchers proposed more designs of SMPO, some of them jump out of the circle and are independent from λ-Matrix. One competitive multiplier design of this type is invented by Reyhani-Masoleh and Hasan [15], which is therefore called RH-SMPO.

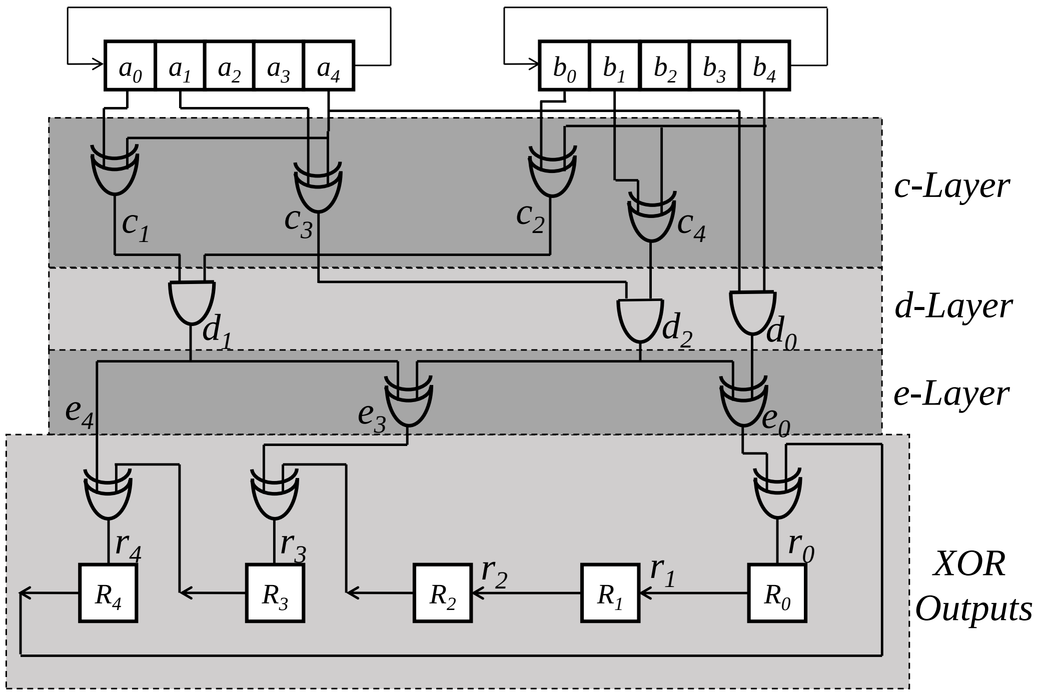

a_0 a_1 a_2 a_3 a_4
b_0 b_1 b_2 b_3 b_4
c_1 c_3 c_2 c_4
c-Layer
d_1 d_2 d_0
d-Layer
e_4 e_3 e_0
e-Layer
r_4 r_3 r_2 r_1 r_0
R_4 R_3 R_2 R_1 R_0
XOR Outputs

Figure 3.3: 5-bit RH-SMPO

Figure 3.3 is a 5-bit RH-SMPO which is functionally equivalent to 5-bit Agnew's SMPO in Figure 3.2. A brief proof is as follows:

Proof. First, we define an auxiliary function for i-th bit

$$F_i(A, B) = a_i b_i \beta + \sum_{j=1}^{v} d_{i,j} \beta^{1+2^j} \tag{3.16}$$

where $0 \leq i \leq k-1, v = \lfloor k/2 \rfloor, 1 \leq j \leq v$. The d-layer index $d_{i,j}$ is defined as

$$d_{i,j} = c_{a,i}c_{b,i} = (a_i + a_{i+j})(b_i + b_{i+j}), \ \ 1 \leq j \leq v \tag{3.17}$$

$i+j$ here is the result reduced modulo k. Note that there is a special boundary case when k is an even number ($v = \frac{k}{2}$):

$$d_{i,v} = (a_i + a_{i+v})b_i$$

With the auxiliary function, we can utilize following theorem (proof refers to [15]):

Theorem 3.4 *Consider three elements A, B and R such that $R = A \times B$ over $\mathbb{F}_{2^k}$. Then,*

$$R = (((F_{k-1}^2 + F_{k-2})^2 + F_{k-3})^2 + \cdots + F_1)^2 + F_0$$

This form is called inductive sum of squares, and corresponds to the cyclic shifting on R_i flip-flops. Concretely, the multiplier behavior is an implementation of following algorithm:

Algorithm 1: NB Multiplication Algorithm in RH-SMPO [15]

```
Input: $A, B \in \mathbb{F}_{2^k}$ given w.r.t. NB $N$
Output: $R = A \times B$
Initialize $A, B$ and aux var $X$ to 0;
for ($i = 0$; $i < k$; ++$i$ ) do
    $X \leftarrow X^2 + F_{k-1}(A, B)$ /*use aux-func from Equation 3.16*/;
    $A \leftarrow A^2$, $B \leftarrow B^2$ /*Right-cyclic shift $A$ and $B$*/;
end
$R \leftarrow X$
```

In this algorithm, we use a fixed auxiliary function F_{k-1} inside the loop. This is because of equation

$$F_{k-l} = F_{k-1}(A^{2^{l-1}}, B^{2^{l-1}}), \ \ 1 \leq l \leq k$$

So using fixed F_{k-1} and squaring A^{2^i} every time inside the loop is equivalent to computing $F_{k-1}, F_{k-2}, \ldots, F_0$ with fixed operands A, B. ■

To better understand the mechanism of RH-SMPO, we will use this 5-bit RH-SMPO as an example and introduce the details on how to design it.

Example 3.10 (Designing a 5-bit RH-SMPO) *From Equation 3.16 we can deploy AND gates in d-layer according to $d_{i,j}$, and XOR gates in c-layer according to Equation 3.17. Concretely, as Algorithm 1 describes, we implement auxiliary function F_{k-1} in the logic:*

$$i = k-1 = 4;\ \ v = \lfloor 5/2 \rfloor = 2$$

$$F_4(A,B) = a_4 b_4 \beta + \sum_{j=1}^{2} d_{4,j}\beta^{1+2^j} = d_0\beta + \sum_{j=1}^{2} d_{4,j}\beta^{1+2^j} \tag{3.18}$$

Consider indices $4+1 = 0 \bmod 5,\ 4+2 = 1 \bmod 5$, write down gates in c-layer and d-layer (besides d_0)

$$c_1 = a_0 + a_4,\ c_2 = b_0 + b_4,\ d_1 = d_{4,1} = c_1 c_2 = (a_4 + a_0)(b_4 + b_0)$$

$$c_3 = a_1 + a_4,\ c_4 = b_1 + b_4,\ d_2 = d_{4,2} = c_3 c_4 = (a_4 + a_1)(b_4 + b_1)$$

The difficult part of the whole design is to deploy XOR gates in e-layer. As the logic layer closest to the outputs R_i, e-layer actually finishes the implementation of $F_{k-1}(A,B)$. But it is not a simple addition; the reason is before bit-wise adding to X^2, it is necessary to turn the sum to NB form. In other words, theoretically we need k XOR gates in e-layer, the output of i-th gate corresponds to the bit multiplying β^{2^i}.

In order to obtain information indicating interconnections between d-layer and e-layer, we need to interpret β^{1+2^j} to NB representation. There is a concept called **multiplication table** *(M-table) which can assist this interpretation. It is defined as a $k \times k$ matrix T over $\mathbb{F}_2$:*

$$\begin{bmatrix} \beta^{1+2^0} \\ \beta^{1+2^1} \\ \beta^{1+2^2} \\ \vdots \\ \beta^{1+2^{k-1}} \end{bmatrix} = \beta \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} T_{0,0} & T_{0,1} & \dots & T_{0,k-1} \\ T_{1,0} & T_{1,1} & \dots & T_{1,k-1} \\ T_{2,0} & T_{2,1} & \dots & T_{2,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ T_{k-1,0} & T_{k-1,1} & \dots & T_{k-1,k-1} \end{bmatrix} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \mathbf{T} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} \tag{3.19}$$

It is a known fact that M-table T can be converted from λ-Matrix M:

$$M_{i,j}^{(0)} = T_{j-i,-i}$$

with indices reduced modulo k (proof given in Appendix A.2). Thus we can write down the M-table of $\mathbb{F}_{2^5}$ with current NB N:

$$\begin{array}{c|ccccc|}
 & 0 & 1 & 0 & 0 & 0 \\
d_1 & 1 & 0 & 0 & 1 & 0 \\
d_2 & 0 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 1 & 0 & 0 \\
 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & e_0 & & & e_3 & e_4
\end{array}$$

Figure 3.4: A 5×5 multiplication table

Note that we only use row 1 and row 2 from the M-table since range $1 \leq j \leq 2$. All nonzero entries in these 2 rows corresponds to the interconnections between d-layer and e-layer. For example, row 1 has two nonzero entries at column 0 and column 3, which corresponds to interconnections between d_1 and e_0, e_3. This conclusion comes from row 1 in Equation 3.19:

$$\beta \cdot \beta^2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \beta^8 \\ \beta^{16} \end{bmatrix} = \beta + \beta^{2^3}$$

Similarly, from row 2 of M-table we derive that d_2 has fanouts e_3, e_4:

$$\beta \cdot \beta^{2^2} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \beta^8 \\ \beta^{16} \end{bmatrix} = \beta^{2^3} + \beta^{2^4}$$

Let us look back at Equation 3.18, we already dealt with the latter part. The first term is always $d_0\beta$, which denotes d_0 should always be connected to $e_0(\beta)$. After gathering all interconnection information, we can translate it to gate-level circuit implementation:

$$e_0 = d_0 + d_1, \; e_3 = d_1 + d_2, \; e_4 = d_2$$

Then the last mission is to implement the output R_i layer. Assume r_{i-1} is the output of R_{i-1} in last clock cycle, we can connect using relation

$$R_i = r_{i-1} + e_i$$

In this example, according to the M-table in Figure 3.4, columns e_1, e_2 have only zeros in its intersection with row d_1, d_2. Thus gates for e_1, e_2 can be omitted.

This finishes the full design procedure for a 5-bit RH-SMPO.

The area cost of RH-SMPO is even smaller than Agnew's SMPO. XOR gates corresponds to all nonzero entries in M-table, which is with the same number of nonzero entries in λ-Matrix (C_N). The number of AND gates equals to v plus 1 (for gate d_0). When using ONB ($C_N = 2k - 1$), the total number of gates is $2k + \lfloor \frac{k}{2} \rfloor$.

3.4 Concluding Remarks

In this section, we introduce basic concepts such as the definition of finite fields and the construction of finite fields. Moreover, we describes a special kind of basis in finite field, and its application on Galois field hardware design. The sequential Galois field multipliers working based on these principles are used as good candidates for applying our functional verification approach in subsequent chapters.

CHAPTER 4

GRÖBER BASIS AND ALGEBRAIC GEOMETRY

This chapter reviews fundamental concepts of commutative and computer algebra which are used in this work. Specifically, this chapter covers monomial ordering, polynomial ideals and varieties, and the computation of Gröbner bases. It also overviews elimination theory as well as Hilbert's Nullstellensatz theorems and how they apply to Galois fields. The results of these theorems are used in polynomial abstraction and formal verification of Galois field circuits and are discussed in subsequent chapters. The material of this chapter is mostly referred from the textbooks [17] [18] and previous work by *Lv* [9] as well as *Pruss* [19].

4.1 Algebraic Geometry Fundamentals

4.1.1 Monomials, Polynomials and Polynomial Arithmetics

Definition 4.1 *A* **monomial** *in variables $x_1, x_2, \cdots, x_d$ is a product of the form:*

$$x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \cdots \cdot x_d^{\alpha_d}, \tag{4.1}$$

where $\alpha_i \geq 0, i \in \{1, \cdots, d\}$. The total degree of the monomial is $\alpha_1 + \cdots + \alpha_d$.

Thus, $x^2 \cdot y$ is a monomial in variables x, y with total degree 3. For simplicity, we will henceforth denote a monomial $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \cdots \cdot x_d^{\alpha_d}$ as x^α, where $\alpha = (\alpha_1, \cdots, \alpha_d)$ is a vector size d of integers ≥ 0, i.e., $\alpha \in \mathbb{Z}_{\geq 0}^d$.

Definition 4.2 *Let $\mathbb{R}$ be a ring. A* **polynomial** *over $\mathbb{R}$ in the indeterminate x is an expression of the form:*

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k = \sum_{i=0}^{k} a_i x^i, \forall a_i \in \mathbb{R}. \tag{4.2}$$

The constants a_i are the coefficients and k is the degree of the polynomial. For example, $8x^3 + 6x + 1$ is a polynomial in x over $\mathbb{Z}$, with coefficients 8, 6, and 1 and degree 3.

Definition 4.3 *The set of all polynomials in the indeterminate x with coefficients in the ring $\mathbb{R}$ forms a* **ring of polynomials** *$\mathbb{R}[x]$. Similarly, $\mathbb{R}[x_1, x_2, \cdots, x_n]$ represents the ring of multivariate polynomials with coefficients in $\mathbb{R}$.*

For example, $\mathbb{Z}_{2^4}[x]$ stands for the set of all polynomials in x with coefficients in $\mathbb{Z}_{2^4}$. $8x^3 + 6x + 1$ is an instance of a polynomial contained in $\mathbb{Z}_{2^4}[x]$.

Definition 4.4 *A* **multivariate polynomial** *f in variables $x_1, x_2, \ldots, x_d$ with coefficients in any given field $\mathbb{F}$ is a finite linear combination of monomials with coefficients in $\mathbb{F}$:*

$$f = \sum_{\alpha} a_\alpha \cdot x^\alpha, \;\; a_\alpha \in \mathbb{F}$$

The set of all polynomials in $x_1, x_2, \ldots, x_d$ with coefficients in field $\mathbb{F}$ is denoted by $\mathbb{F}[x_1, x_2, \ldots, x_d]$. Thus, $f \in \mathbb{F}[x_1, x_2, \ldots, x_d]$

1. *We refer to the constant $a_\alpha \in \mathbb{F}$ as the* **coefficient** *of the monomial $a_\alpha x^\alpha$.*

2. *If $a_\alpha \neq 0$, we call $a_\alpha x^\alpha$ a term of f.*

As an example, $2x^2 + y$ is a polynomial with two terms $2x^2$ and y, with 2 and 1 as coefficients respectively. In contrast, $x + y^{-1}$ is not a polynomial because the exponent of y is less than 0.

Since a polynomial is a sum of its terms, these terms have to be arranged unambiguously so that they can be manipulated in a consistent manner. Therefore, we need to establish a concept of **term ordering** (also called monomial ordering). A term ordering, represented by $>$, defines how terms in a polynomial are ordered.

Common term orderings are lexicographic ordering (LEX) and its variants: degree-lexicographic ordering (DEGLEX) and reverse degree-lexicographic ordering (DEGREVLEX).

A **lexicographic ordering** (lex) is a total-ordering $>$ such that variables in the terms are lexicographically ordered, i.e. simply based on when the variables appear in the ordering. Higher variable-degrees take precedence over lower degrees for equivalent variables (e.g. $a^3 > a^2$ due to $a \cdot a \cdot a > a \cdot a \cdot 1$).

Definition 4.5 Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d);\ \beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \textit{Starting from the left, the first co-ordinates of } \alpha_i, \beta_i \\ \textit{that are different satisfy } \alpha_i > \beta_i \end{cases} \tag{4.3}$$

A **degree-lexicographic ordering** (deglex) is a total-ordering $>$ such that the total degree of a term takes precedence over the lexicographic ordering. A **degree-reverse-lexicographic ordering** (degrevlex) is the same as a deglex ordering, however terms are lexed in reverse.

Definition 4.6 Degree Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d);\ \beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i & \textit{or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and } x^\alpha > x^\beta & \textit{w.r.t. lex order} \end{cases} \tag{4.4}$$

Definition 4.7 Degree Reverse Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d);\ \beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i \textit{ or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and the first co-ordinates} \\ \alpha_i, \beta_i \textit{ from the right, which are different, satisfy } \alpha_i < \beta_i \end{cases} \tag{4.5}$$

Based on the *monomial ordering*, we have the following concepts:

Definition 4.8 *The* **leading term** *is the first term in a term-ordered polynomial. Likewise, the* **leading coefficient** *is the coefficient of the leading term. Finally, a* **leading monomial** *is the leading term lacking the coefficient. We use the following notation:*

$$lt(f) \quad \textit{— Leading Term} \tag{4.6}$$

$$lc(f) \quad \textit{— Leading Coefficient} \tag{4.7}$$

$$lm(f) \quad \textit{— Leading Monomial} \tag{4.8}$$

$$tail(f) \quad f - lt(f) \tag{4.9}$$

Example 4.1

$$\begin{aligned} f &= 3a^2b + 2ab + 4bc && (4.10)\\ lt(f) &= 3a^2b && (4.11)\\ lc(f) &= 3 && (4.12)\\ lm(f) &= a^2b && (4.13)\\ tail(f) &= 2ab + 4bc && (4.14) \end{aligned}$$

Polynomial division is an operation over polynomials that is dependent on the imposed monomial ordering. Dividing a polynomial f by another polynomial g cancels the leading term of f to derive a new polynomial.

Definition 4.9 *Let $\mathbb{F}$ be a field and let $f, g \in \mathbb{F}[x_1, x_2, \ldots, x_d]$ be polynomials over the field.* **Polynomial division** *of f by g is computes following:*

$$f - \frac{lt(f)}{lt(g)} \cdot g \tag{4.15}$$

This polynomial division is denoted

$$f \xrightarrow{g} r \tag{4.16}$$

where r is the resulting polynomial of the division. If $\frac{lt(f)}{lt(g)}$ is non-zero, then f is considered divisible by g, i.e. $g \mid f$.

Notice that if $g \nmid f$, that is if f is not divisible by g, then the division operation gives $r = f$.

Example 4.2 *Over $\mathbb{R}[x, y, z]$, set the lex term order $x > y > z$. Let $f = -2x^3 + 2x^2yz + 3xy^3$ and $g = x^2 + yz$.*

$$\frac{lt(f)}{lt(g)} = \frac{-2x^3}{x^2} = -2x \tag{4.17}$$

Since $\frac{lt(f)}{lt(g)}$ is non-zero $g|f$. The division, $f \xrightarrow{g} r$, is computed as:

$$\begin{aligned} r &= f - \frac{lt(f)}{lt(g)} \cdot g = -2x^3 + 2x^2yz + 3xy^3 - (-2x \cdot (x^2 + yz)) \\ &= -2x^3 + 2x^2yz + 3xy^3 - (-2x^3 - 2xyz) = 2x^2yz + 3xy^3 + 2xyz \end{aligned} \tag{4.18}$$

Notice that the division $f \xrightarrow{g} r$ cancels the leading term of f.

Similarly, we can also define when a polynomial is divided (reduced) by a set of polynomials.

Definition 4.10 *The* **reduction** *of a polynomial f, by another polynomial g, to a reduced polynomial r is denoted:*

$$f \xrightarrow{g}_{+} r$$

which is transitive and reflective closure of relation $f \xrightarrow{g} r$. Reduction is carried out using multivariate, polynomial long division.

For sets of polynomials, the notation

$$f \xrightarrow{F}_{+} r$$

represents the reduced polynomial r resulting from f as reduced by a set of non-zero polynomials $F = \{f_1, \ldots, f_s\}$. The polynomial r is considered **reduced** *if $r = 0$ or no term in r is divisible by $lm(f_i), \forall f_i \in F$.*

The reduction process $f \xrightarrow{F}_{+} r$, of dividing a polynomial f by a set of polynomials of F, can be modeled as repeated long-division of f by each of the polynomials in F until no further reductions can be made. The result of this process is then r. This reduction process is shown in Algorithm 2.

Algorithm 2: Polynomial Reduction

```
Input: $f, f_1, \dots, f_s$
Output: $r, a_1, \dots, a_s$, such that $f = a_1 \cdot f_1 + \cdots + a_s \cdot f_s + r$.
$a_1 = a_2 = \cdots = a_s = 0$; $r = 0$;
$p := f$;
while $p \neq 0$ do
    i=1;
    divisionmark = false;
    while $i \leq s$ && divisionmark = false do
        if $f_i$ can divide $p$ then
            $a_i = a_i + lt(p)/lt(f_i)$;
            $p = p - lt(p)/lt(f_i) \cdot f_i$;
            divisionmark = true;
        else
            i=i+1;
        end
    end
    if divisionmark = false then
        $r = r + lt(p)$;
        $p = p - lt(p)$;
    end
end
```

The reduction algorithm keeps canceling the leading terms of polynomials until no more leading terms can be further canceled. So the key step is $p = p - lt(p)/lt(f_i) \cdot f_i$, as the following example shows.

Example 4.3 *Given $f = y^2 - x$ and $f_1 = y - x$ in $\mathbb{Q}[x, y]$ with deglex: $y > x$, perform $f \xrightarrow{f_1}_+ r$:*

1. $f = y^2 - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = y^2 - x - (y^2/y) \cdot (y - x) = y \cdot x - x$

2. $f = y \cdot x - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = (y \cdot x - x)/f_1 = x^2 - x$

3. $f = x^2 - x$, *no more operations possible, so* $r = x^2 - x$

4.1.2 Varieties and Ideals

In computer-algebra based formal verification, it is often necessary to analyze the presence or absence of solutions to a given system of constraints. In our applications, these constraints are polynomials and their solutions are modeled as **varieties**.

Definition 4.11 *Let $\mathbb{F}$ be a field, and let $f_1, \ldots, f_s \in \mathbb{F}[x_1, x_2, \ldots, x_d]$. We call $V(f_1, \ldots, f_s)$ the* **affine variety** *defined by $f_1, \ldots, f_s$ as:*

$$V(f_1, \ldots, f_s) = \{(a_1, \ldots, a_d) \in \mathbb{F}^d : f_i(a_1, \ldots, a_d) = 0, \forall i, 1 \leq i \leq s\} \tag{4.19}$$

$V(f_1, \ldots, f_s) \in \mathbb{F}^d$ is **the set of all solutions** in $\mathbb{F}^d$ of the system of equations: $f_1(x_1, \ldots, x_d) = \cdots = f_s(x_1, \ldots, x_d) = 0$.

Example 4.4 *Given $\mathbb{R}[x, y]$, $V(x^2 + y^2)$ is the set of all elements that satisfy $x^2 + y^2 = 0$ over $\mathbb{R}^2$. So $V(x^2 + y^2) = \{(0, 0)\}$. Similarly, in $\mathbb{R}[x, y]$, $V(x^2 + y^2 - 1) = \{all\ points\ on\ the\ circle : x^2 + y^2 - 1 = 0\}$. Note that varieties depend on which field we are operating on. For the same polynomial $x^2 + 1$, we have:*

- *In $\mathbb{R}[x]$, $V(x^2 + 1) = \emptyset$.*
- *In $\mathbb{C}[x]$, $V(x^2 + 1) = \{(\pm i)\}$.*

The above example shows the variety can be infinite, finite (non-empty set) or empty. It is interesting to note that since we will be operating over finite fields $\mathbb{F}_q$, and any finite set of points is a variety. Likewise, any variety over $\mathbb{F}_q$ is finite (or empty). Consider the points $\{(a_1, \ldots, a_d) : a_1, \ldots, a_d \in \mathbb{F}_q\}$ in $\mathbb{F}_q^d$. Any single point is a variety of some polynomial system: e.g. $(a_1, \ldots, a_d)$ is a variety of $x_1 - a_1 = x_2 - a_2 = \cdots = x_d - a_d = 0$. **Finite unions** and **finite intersections** of varieties are also varieties.

Example 4.5 *Let $U = V(f_1, \ldots, f_s)$ and $W = V(g_1, \ldots, g_t)$ in $\mathbb{F}_q$. Then:*

- $U \cap W = V(f_1, \ldots, f_s, g_1, \ldots, g_t)$
- $U \cup W = V(f_i g_j : 1 \leq i \leq s, 1 \leq j \leq t)$

One important distinction we need to make about varieties is that a variety depends not just on the given system of polynomial equations, but rather on the **ideal** generated by the polynomials.

Definition 4.12 *A subset $I \subset \mathbb{F}[x_1, x_2, \ldots, x_d]$ is an* **ideal** *if it satisfies:*

- *$0 \in I$*
- *I is closed under addition: $x, y \in I \Rightarrow x + y \in I$*
- *If $x \in \mathbb{F}[x_1, x_2, \ldots, x_d]$ and $y \in I$, then $x \cdot y \in I$ and $y \cdot x \in I$.*

An ideal is generated by its *basis* or *generators*.

Definition 4.13 *Let $f_1, f_2, \ldots, f_s$ be polynomials of the ring $\mathbb{F}[x_1, x_2, \ldots, x_d]$. Let I be an ideal generated by $f_1, f_2, \ldots, f_s$. Then:*

$$I = \langle f_1, \ldots, f_s \rangle = \{h_1 f_1 + h_2 f_2 + \ldots + h_s f_s : h_1, \ldots, h_s \in \mathbb{F}[x_1, \ldots, x_d]\}$$

then, $f_1, \ldots, f_s$ are called the **basis (or generators)** *of the ideal I and correspondingly I is denoted as $I = \langle f_1, f_2, \ldots, f_s \rangle$.*

Example 4.6 *The set of even integers, which is a subset of the ring of integers Z, forms an ideal of Z. This can be seen from the following;*

- *0 belongs to the set of even integers.*
- *The sum of two even integers x and y is always an even integer.*
- *The product of any integer x with an even integer y is always an even integer.*

Example 4.7 *Given $\mathbb{R}[x, y]$, $I = \langle x, y \rangle$ is an ideal containing all polynomials generated by x and y, such as $x^2 + y$ and $x + x \cdot y$. The ideal $J = \langle x^2, y^2 \rangle$ is an ideal containing all polynomials generated by x^2 and y^2, such as $x^2 + y^3$ and $x^{10} + x^2 \cdot y^2$. Notice that $J \subset I$ because every polynomial generated by J can be generated by I. But $I \neq J$ because $x + y$ can only be generated by I.*

The same ideal may have many different bases. For instance, it is possible to have different sets of polynomials $\{f_1, \ldots, f_s\}$ and $\{g_1, \ldots, g_t\}$ that may generate the same ideal, i.e., $\langle f_1, \ldots, f_s \rangle = \langle g_1, \ldots, g_t \rangle$. Since variety depends on the ideal, these sets of polynomials have the same solutions.

Proposition 4.1 *If $f_1, \dots, f_s$ and $g_1, \dots, g_t$ are bases of the same ideal in $\mathbb{F}[x_1, \dots, x_d]$, so that $\langle f_1, \dots, f_s \rangle = \langle g_1, \dots, g_t \rangle$, then $V(f_1, \dots, f_s) = V(g_1, \dots, g_t)$.*

Example 4.8 *Consider the two bases $F_1 = \{(2x^2 + 3y^2 - 11, x^2 - y^2 - 3\}$ and $F_2 = \{x^2 - 4, y^2 - 1\}$. These two bases generate the same ideal, i.e., $\langle F_1 \rangle = \langle F_2 \rangle$. Therefore, they represent the same variety, i.e.,*

$$V(F_1) = V(F_2) = \{\pm 2, \pm 1\}. \tag{4.20}$$

Ideals and their varieties are a key part of computer-algebra based formal verification. A given hardware design can be transformed into a set of polynomials over a field, $f_1, \dots, f_s \in F$. This set of polynomials gives the system of equations:

$$\begin{aligned} f_1 &= 0 \\ &\vdots \\ f_s &= 0 \end{aligned}$$

Using algebra, it is possible to derive new equations from the original system. The ideal $\langle f_1, \dots, f_s \rangle$ provides a way of analyzing such *consequences* of a system of polynomials.

Example 4.9 *Given two equations in $\mathbb{R}[x, y, z]$:*

$$\begin{aligned} x &= z + 1 \\ y &= x^2 + 1 \end{aligned}$$

we can eliminate x to obtain a new equation:

$$y = (z + 1)^2 + 1 = z^2 + 2z + 2$$

Let $f_1, f_2, h \in \mathbb{R}[x, y, z]$ be polynomials based on these equations:

$$\begin{aligned} f_1 &= x - z - 1 &&= 0 \\ f_2 &= y - x^2 - 1 &&= 0 \\ h &= y - z^2 - 2z - 2 &&= 0 \end{aligned}$$

If I is the ideal generated by f_1 and f_2, i.e. $I = \langle f_1, f_2 \rangle$, then we find $h \in I$ as follows:

$$\begin{aligned} g_1 &= x + z + 1 \\ g_2 &= 1 \\ h &= g_1 \cdot f_1 + g_2 \cdot f_2 = y - z^2 - 2z - 2 \end{aligned}$$

where $g_1, g_2 \in \mathbb{R}[x, y, z]$. Thus, we call h a **member of the ideal** *I.*

Let $\mathbb{F}$ be any field and let $\mathbf{a} = (a_1, \ldots, a_d) \in \mathbb{F}^d$ be a point, and $f \in \mathbb{F}[x_1, \ldots, x_d]$ be a polynomial. We say that f *vanishes* on $\mathbf{a}$ if $f(\mathbf{a}) = 0$, i.e., $\mathbf{a}$ is in the variety of f.

Definition 4.14 *For any variety V of $\mathbb{F}^d$, the ideal of polynomials that vanish on V, called the vanishing ideal of V, is defined as* **ideal of variety***:*

$$I(V) = \{f \in \mathbb{F}[x_1, \ldots, x_d] : \forall \mathbf{a} \in V, f(\mathbf{a}) = 0\}$$

Proposition 4.2 *If a polynomial f vanishes on a variety V, then $f \in I(V)$.*

Example 4.10 *Let ideal $J = \langle x^2, y^2 \rangle$. Then $V(J) = \{(0, 0)\}$. All polynomials in J will obviously agree with the solution and vanish on this variety. However, the polynomials x, y are not in J but they also vanish on this variety. Therefore, $I(V(J))$ is the set of all polynomials that vanish on $V(J)$, and the polynomials x, y are members of $I(V(J))$.*

Definition 4.15 *Let $J \subset \mathbb{F}[x_1, \ldots, x_d]$ be an ideal. The* **radical of** *J is defined as $\sqrt{J} = \{f \in \mathbb{F}[x_1, \ldots, x_d] : \exists m \in \mathbb{N}, f^m \in J\}$.*

Example 4.11 *Let $J = \langle x^2, y^2 \rangle \subset \mathbb{F}[x, y]$. Neither x nor y belongs to J, but they belong to $\sqrt{J}$. Similarly, $x \cdot y \notin J$, but since $(x \cdot y)^2 = x^2 \cdot y^2 \in J$, therefore, $x \cdot y \in \sqrt{J}$.*

When $J = \sqrt{J}$, then J is said to be a *radical ideal*. Moreover, $I(V)$ is a radical ideal. By analyzing the ideal J, generated by a system of polynomials derived from a hardware design, its variety $V(J)$, and the ideal of polynomials that vanish over this variety, $I(V(J))$, we can reason about the existence of certain properties of the design. To check for the validity of a property, we formulate the property as a polynomial and then perform an **ideal membership test** to determine if this polynomial is contained within the ideal $I(V(J))$. A **Gröbner basis** provides a decision procedure for performing this test, which is described in the following part.

4.1.3 Gröbner Bases

As mentioned earlier, different polynomial sets may generate the same ideal. Some of these generating sets may be a better representation of the ideal, and thus provide more information and insight into the properties of the ideal. One such ideal representation is a **Gröbner basis**, which has a number of important properties that can solve numerous polynomial decision questions:

- Presence or absence of solutions (varieties)
- Dimension of the varieties
- Ideal membership of a polynomial
- ......

In essence, a Gröbner basis is a canonical representation of an ideal. There are many equivalent definitions of Gröbner bases, so we start with the definition that best describes their properties:

Definition 4.16 *A set of non-zero polynomials $G = \{g_1, \ldots, g_t\}$ which generate the ideal $I = \langle g_1, \ldots, g_t \rangle$, is called a* **Gröbner basis** *for I if and only if for all $f \in I$ where $f \neq 0$, there exists a $g_i \in G$ such that $lm(g_i)$ divides $lm(f)$.*

$$G = \textit{GröbnerBasis}(I) \iff \forall f \in I : f \neq 0, \exists g_i \in G : lm(g_i) \mid lm(f) \qquad (4.21)$$

Gröbner basis has an important property, and therefore can be used to perform ideal membership test. Formally speaking, a Gröbner basis gives a decision procedure to test for polynomial membership in an ideal. This is explained in the following Theorem.

Theorem 4.1 Ideal Membership Test *Let $G = \{g_1, \cdots, g_t\}$ be a Gröbner basis for an ideal $I \subset \mathbb{K}[x_1, \cdots, x_d]$ and let $f \in \mathbb{K}[x_1, \ldots, x_d]$. Then $f \in I$ if and only if the remainder on division of f by G is zero.*

In other words,

$$f \in I \iff f \xrightarrow{G}_{+} 0 \tag{4.22}$$

Example 4.12 *Consider Example 4.13. Let $f = y^2x - x$ be another polynomial. Note that $f = yf_1 + f_2$, so $f \in I$. If we divide f by f_1 first and then by f_2, we will obtain a zero remainder. However, since the set $\{f_1, f_2\}$ is not a Gröbner basis, we find that the reduction $f \xrightarrow{f_2} x^2 - x \xrightarrow{f_1} x^2 - x \neq 0$; i.e. dividing f by f_2 first and then by f_1 does not lead to a zero remainder. However, if we compute the Gröbner basis G of I, $G = \{x^2 - x, yx - y, y^2 - x\}$, dividing f by polynomials in G in any order will always lead to the zero remainder. Therefore, one can decide ideal membership unequivocally using the Gröbner basis.*

The foundation for computing the Gröbner basis of an ideal was laid out by Buchberger [20]. Given a set of polynomials $F = \{f_1, \ldots, f_s\}$ that generate ideal $I = \langle f_1, \ldots, f_s \rangle$, Buchberger gives an algorithm to compute a Gröbner basis $G = \langle g_1, \ldots, g_t \rangle$. This algorithm relies on the notions of S-polynomials and polynomial reduction.

Definition 4.17 *For $f, g \in \mathbb{F}[x_1, \ldots, x_d]$, an* **S-polynomial** *$Spoly(f, g)$ is defined as:*

$$Spoly(f,g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g \tag{4.23}$$

$$\text{where } L = lcm\,(lt(f), lt(g))$$

Note lcm denotes least common multiple.

With the notions of S-polynomials and polynomial reduction in place, we can now present Buchberger's Algorithm for computing Gröbner bases [20]. Note that a fixed monomial (term) ordering is required for a Gröbner basis computation to ensure that polynomials are manipulated in a consistent manner.

Algorithm 3: Buchberger's Algorithm

```
Input: $F = \{f_1, \dots, f_s\}$, such that $I = \langle f_1, \dots, f_s \rangle$
, and term order $>$ Output: $G = \{g_1, \dots, g_t\}$, a Gröbner basis of $I$
$G := F$;
repeat
    $G' := G$;
    for each pair $\{f_i, f_j\}, i \neq j$ in $G'$ do
        $Spoly(f_i, f_j) \xrightarrow{G'}_+ r$ ;
        if $r \neq 0$ then
            $G := G \cup \{r\}$ ;
        end
    end
until $G = G'$;
```

Buchberger's algorithm takes pairs of polynomials (f_i, f_j) in the basis G and combines them into "S-polynomials" ($Spoly(f_i, f_j)$) to cancel leading terms. The S-polynomial is then reduced (divided) by all elements of G to a remainder r, denoted as $Spoly(f_i, f_j) \xrightarrow{G}_+ r$. This process is repeated for all unique pairs of polynomials, including those created by newly added elements, until no new polynomials are generated; ultimately constructing the Gröbner basis.

Example 4.13 *Consider the ideal $I \subset \mathbb{Q}[x, y]$, $I = \langle f_1, f_2 \rangle$, where $f_1 = yx - y,\ f_2 = y^2 - x$. Assume a degree-lexicographic term ordering with $y > x$ is imposed.*

First, we need to compute $Spoly(f_1, f_2) = x \cdot f_2 - y \cdot f_1 = y^2 - x^2$. Then we conduct a polynomial reduction $y^2 - x^2 \xrightarrow{f_2} x^2 - x \xrightarrow{f_1} x^2 - x$. Let $f_3 = x^2 - x$. Then G is updated as $\{f_1, f_2, f_3\}$. Next we compute $Spoly(f_1, f_3) = 0$. So there is no new polynomial generated. Similarly, we compute $Spoly(f_2, f_3) = x \cdot y^2 - x^3$, followed by $x \cdot y^2 - x^3 \xrightarrow{f_1} y^2 - x^3 \xrightarrow{f_2} x - x^3 \xrightarrow{f_2} 0$. Again, no polynomial is generated. Finally, $G = \{f_1, f_2, f_3\}$.

When computing a Gröbner basis, it's important to note that if $lt(f_i)$ and $lt(f_j)$ have no common variables, the S-poly reduction step in Buchberger's algorithm will reduce to 0.

Lemma 4.1 *In Buchberger's algorithm, when $gcd(lt(f_i), lt(f_j)) = 0$, the S-poly reduction*

$$Spoly(f_i, f_j) \xrightarrow{G'}_+ r$$

will produce $r = 0$.

Proof. If $lt(f)$ and $lt(g)$ have no common variables, $L = lcm(lt(f), lt(g)) = lt(f) \cdot lt(g)$. Then:

$$Spoly(f, g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g = \frac{lt(f) \cdot lt(g)}{lt(f)} \cdot f - \frac{lt(f) \cdot lt(g)}{lt(g)} \cdot g = lt(g) \cdot f - lt(f) \cdot g$$

Thus, every monomial in $Spoly(f, g)$ is divisible by either $lt(f)$ or $lt(g)$, so computing $Spoly(f, g) \xrightarrow{f,g}_+ r$ will give $r = 0$. ■

A Gröbner basis is not a canonical representation of an ideal, but a **reduced Gröbner basis** is. To compute a reduced Gröbner basis, we first must compute a minimal Gröbner basis.

Definition 4.18 *A* **minimal Gröbner basis** *for a polynomial ideal I is a Gröbner basis G for I such that*

- $lc(g_i) = 1, \forall g_i \in G$
- $\forall g_i \in G$, $lt(g_i) \notin \langle lt(G - \{g_i\}) \rangle$

A **minimal** Gröbner basis is a Gröbner basis such that all polynomials have a coefficient of 1 and no leading term of any element in G divides another in G. Given a Gröbner basis G, a minimal Gröbner basis can be computed as follows:

1. Minimize every $g_i \in G$, i.e $g_i = g_i / lc(g_i)$
2. For $g_i, g_j \in G$ where $i \neq j$, remove g_i from G if $lt(g_i) \mid lt(g_j)$, i.e. remove every polynomial in G whose leading term is divisible by the leading term of some other polynomial in G.

A minimal Gröbner basis can then be further reduced.

Definition 4.19 *A* **reduced Gröbner basis** *for a polynomial ideal I is a Gröbner basis $G = \{g_1, \ldots, g_t\}$ such that:*

- *$lc(g_i) = 1, \forall g_i \in G$*
- *$\forall g_i \in G$, no monomial of g_i lies in $\langle lt(G - \{g_i\})\rangle$*

G is a reduced Gröbner basis when no monomial of any element in G divides the leading term of another element. This reduction is achieved as follows:

Definition 4.20 *Let $H = \{h_1, \dots, h_t\}$ be a minimal Gröbner basis. Apply the following reduction process:*

- *$h_1 \xrightarrow{G_1}_+ g_1$, where g_1 is reduced w.r.t. $G_1 = \{h_2, \dots, h_t\}$*
- *$h_2 \xrightarrow{G_2}_+ g_2$, where g_2 is reduced w.r.t. $G_2 = \{g_1, h_3, \dots, h_t\}$*
- *$h_3 \xrightarrow{G_3}_+ g_3$, where g_3 is reduced w.r.t. $G_3 = \{g_1, g_2, h_4, \dots, h_t\}$*

  $\vdots$

- *$h_t \xrightarrow{G_t}_+ g_t$, where g_t is reduced w.r.t. $G_t = \{g_1, g_2, g_3, \dots, g_{t-1}\}$*

Then $G = \{g_1, \dots, g_t\}$ is a **reduced Gröbner basis.**

Subject to the given term order $>$, such a reduced Gröbner basis $G = \{g_1, \dots, g_t\}$ is a **unique canonical representation of the ideal**, as given by Proposition 4.3 below.

Proposition 4.3 *[18] Let $I \neq \{0\}$ be a polynomial ideal. Then, for a given monomial ordering, I has a unique reduced Gröbner basis.*

The high computational complexity if Gröbner basis computation is treated as an issue because of its high cost on time and space. Concretely, for arbitrary polynomial set, the worst case computation time/space cost of its Gröbner basis is doubly exponential bounded – $q^{q^{O(|\phi|)}}$ [21]. However, in practice such as applications of circuit verification, the polynomial is well restricted rather than arbitrary. Gao et al. [22] proves that when the number of variables and degree of polynomials are restricted, the complexity reduces to single exponential $q^O(|\phi|)$. This provides the possibility to make use of Gröbner basis under restricted situations, which is the theoretic basis of this dissertation.

Gröbner basis computation depends on the $Spoly$ computation, which in turn depends on the leading terms of polynomials. Thus, different monomial orderings can result in different Gröbner basis computations for the same ideal. Computation using a degrevlex ordering tends to be least difficult, while lex ordering tends to be computationally complex. However, lex ordering used in the computation of Gröbner basis is an **elimination ordering**; that is, the polynomials contained in the resulting Gröbner basis have continuously eliminated variables in the ordering. This is the topic of elimination theory, which is described in the following sections as well as its theoretic basis – the Nullstellensatz theory.

4.2 Hilbert's Nullstellensatz

In this section, we further describe some correspondence between ideals and varieties in the context of algebraic geometry. The celebrated results of Hilbert's Nullstellensatz establish these correspondences.

Definition 4.21 *A field $\overline{\mathbb{F}}$ is an* **algebraically closed** *field if every polynomial in one variable with degree at least 1, with coefficients in $\overline{\mathbb{F}}$, has a root in $\overline{\mathbb{F}}$.*

In other words, any non-constant polynomial equation over $\overline{\mathbb{F}}\,[x]$ always has at least one root in $\overline{\mathbb{F}}$. Every field $\mathbb{F}$ is contained in an algebraically closed one $\overline{\mathbb{F}}$. For example, the field of real numbers $\mathbb{R}$ is not an algebraically closed field, because $x^2 + 1 = 0$ has no root in $\mathbb{R}$. However, $x^2 + 1 = 0$ has roots in the field of complex numbers $\mathbb{C}$, which is an algebraically closed field. In fact, $\mathbb{C}$ is the algebra closure of $\mathbb{R}$. Every algebraically closed field is an infinite field.

Theorem 4.2 *Weak Nullstellensatz Let $J \subset \overline{\mathbb{F}}[x_1, x_2, \cdots, x_d]$ be an ideal satisfying $V(J) = \emptyset$. Then $J = \overline{\mathbb{F}}[x_1, x_2, \cdots, x_d]$, Or equivalently,*

$$V(J) = \emptyset \iff J = \overline{\mathbb{F}}[x_1, x_2, \cdots, x_d] = \langle 1 \rangle \tag{4.24}$$

Corollary 4.1 *Let $J = \langle f_1, \ldots, f_s \rangle \subset \overline{\mathbb{F}}[x_1, x_2, \cdots, x_d]$. Let G be the reduced Gröbner basis of J. Then $V(J) = 0 \iff G = \{1\}$.*

Weak Nullstellensatz offers a way to evaluate whether or not the system of multivariate polynomial equations (ideal J) has common solutions in $\overline{\mathbb{F}}^d$. For this purpose, we only need to check if the ideal is generated by the unit element, i.e., $1 \in J$. This approach can be used to evaluate the feasibility of constraints in verification problems. An interesting result is one of **Strong Nullstellensatz**. The strong Nullstellensatz establishes the correspondence between radical ideals and varieties.

Theorem 4.3 *(The Strong Nullstellensatz [18]) Let $\overline{\mathbb{F}}$ be an algebraically closed field, and let J be an ideal in $\overline{\mathbb{F}}[x_1, \dots, x_d]$. Then we have $I(V_{\overline{\mathbb{F}}}(J)) = \sqrt{J}$.*

Strong Nullstellensatz holds a special form over Galois fields $\mathbb{F}_q$. Recall the notion of vanishing polynomials over Galois fields from the previous chapter: for every element $A \in \mathbb{F}_q$, $A - A^q = 0$; then the polynomial $x^q - x$ in $\mathbb{F}_q[x]$ vanishes over $\mathbb{F}_q$. Thus, if $J_0 = \langle x^q - x \rangle$ is the ideal generated by the vanishing polynomial, $V(J_0) = \mathbb{F}_q$. Similarly, over $\mathbb{F}_q[x_1, \dots, x_d]$, J_0 is $\langle x_1^q - x_1, \dots, x_d^q - x_d \rangle$ and $V(J_0) = (F_q)^d$.

Definition 4.22 *Given two ideals, $I_1 = \langle f_1, \dots, f_s \rangle$ and $I_2 = \langle g_1, \dots g_t \rangle$, then the* **sum of ideals** $I_1 + I_2 = \langle f_1, \dots, f_s, g_1, \dots g_t \rangle$

Theorem 4.4 *(Strong Nullstellensatz over $\mathbb{F}_q$) For any Galois field $\mathbb{F}_q$, let $J \subset \mathbb{F}_q[x_1, \dots, x_d]$ be any ideal and let $J_0 = \langle x_1^q - x_1, x_d^q - x_d \rangle$ be the ideal of all vanishing polynomials. Let $V_{\mathbb{F}_q}(J)$ denote the variety of J over $\mathbb{F}_q$. Then, $I(V_{\mathbb{F}_q}(J)) = J + J_0$.*

The proof is given in [23]. Here, we provide a proof outline.

Proof.

1. $\sqrt{J + J_0} = J + J_0$. That is, $J + J_0$ is a radical ideal.

2. $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J + J_0)$.

3. Due to (2), $I(V_{\mathbb{F}_q}(J)) = I(V_{\overline{\mathbb{F}_q}}(J + J_0))$. By Strong Nullstellensatz, this is equivalent to $\sqrt{J + J_0}$. Finally, due to (1), this is equivalent to $J + J_0$.

■

Using this result, Weak Nullstellensatz can be modified to be applicable over finite fields $\mathbb{F}_q$.

Theorem 4.5 $[\mathbf{Weak\ Nullstellensatz\ in\ \mathbb{F}_q}]$ *[24]*
Given $f_1, f_2, \cdots, f_s \in \mathbb{F}_q[x_1, x_2, \cdots, x_d]$*. Let* $J = \langle f_1, f_2, \cdots, f_s\rangle \subset \mathbb{F}_q[x_1, x_2, \cdots, x_d]$ *be an ideal. Let* $J_0 = \langle x_1^{2^k} - x_1, x_2^{2^k} - x_2, \cdots, x_d^{2^k} - x_d\rangle$ *be the ideal of vanishing polynomials in* $\mathbb{F}_q$*. Then* $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J + J_0) = \emptyset$*, if and only if the reduced GröbnerBasis*$(J + J_0) = \{1\}$.

The proof is given in [24]. Here, we provide a proof outline.

Proof. The variety of J over $\mathbb{F}_q[x_1, x_2, \cdots, x_d]$ is equivalent to the variety over the algebraic closure of $\mathbb{F}_q$ intersected by the entire field $\mathbb{F}_q$. That is, $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J) \cap \mathbb{F}_q$.

Let $J_0 = \langle x_1^{2^k} - x_1, x_2^{2^k} - x_2, \cdots, x_d^{2^k} - x_d\rangle$ be the ideal generated by all vanishing polynomials in $\mathbb{F}_q[x_1, x_2, \cdots, x_d]$. Then $V_{\overline{\mathbb{F}_q}}(J_0) = \mathbb{F}_q$.

Thus, $V_{\mathbb{F}_q}(J) = V_{\overline{\mathbb{F}_q}}(J) \cap V_{\overline{\mathbb{F}_q}}(J_0) = V_{\overline{\mathbb{F}_q}}(J + J_0)$. ■

4.3 Elimination and Abstraction Theory

Elimination of certain variables in a system of polynomials is a common operation when some variables is not needed in modeling and analysis. In this section, eliminating variables targets a tight-bound over-approximation which is equivalent to existential quantifier elimination in first-order logic, or variable smoothing in Boolean operations. We introduce a eliminating method based on algebraic geometry concepts, and use it as the fundamentals of abstraction of a circuit.

4.3.1 Elimination Theory

Assume we are given a set of polynomials $f_1, \ldots, f_s$ belongs to ring $\mathbb{F}_q[x_1, \ldots, x_l, \ldots, x_d]$. First, we show that eliminating the $x_1, \ldots, x_l$ variables is equivalent to *projecting* the variety $V(\langle f_1, \ldots, f_s\rangle)$ from $\mathbb{F}_q^d$ to $\mathbb{F}_q^{d-l}$. Figure 4.1 is an example of projection in space of varieties from $\mathbb{F}_q^3$ to $\mathbb{F}_q^2$, corresponding to eliminating variable x_1 from a system of polynomials belonging to $\mathbb{F}_q[x_1, x_2, x_3]$.

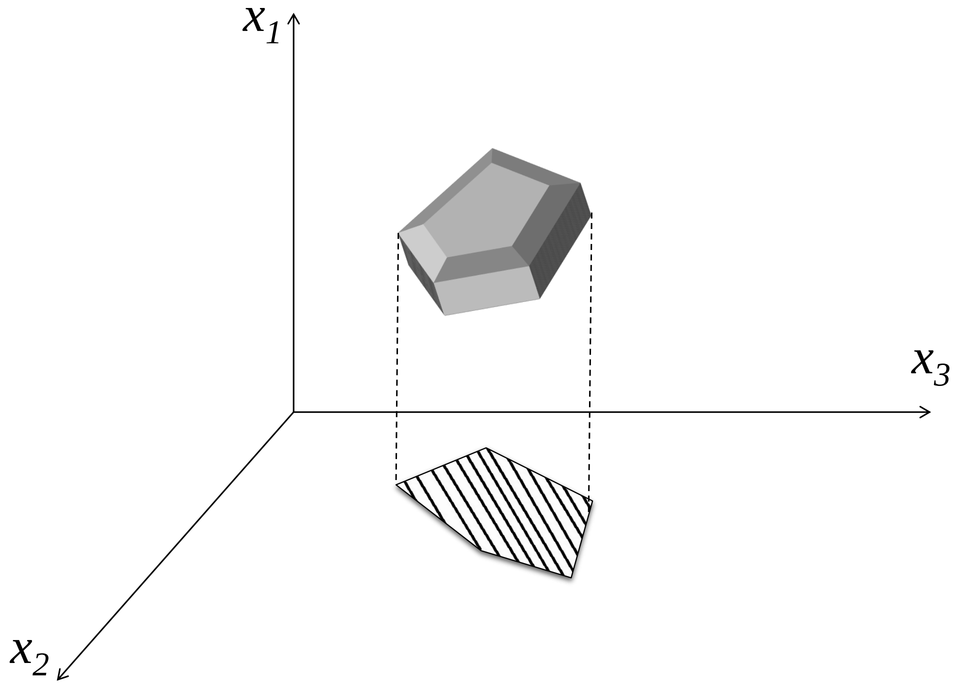


Figure 4.1: An example of projection from $\mathbb{F}_q^3$ to $\mathbb{F}_q^2$

Formally, we define concepts of projection and elimination ideal:

Definition 4.23 *The l-***th projection** *mapping is defined as:*

$$\pi_l : \mathbb{F}_q^d \to \mathbb{F}_q^{d-l}, \;\; \pi_l((x_1, \ldots, x_d)) = (x_{l+1} \ldots, x_d)$$

where $l < d$. For any set $A \subseteq \mathbb{F}_q^d$, we write

$$\pi_l(A) = \{\pi_i(x) : x \in A\} \subseteq \mathbb{F}_q^{d-l}$$

Definition 4.24 *Let J be an ideal in $\mathbb{F}_q[x_1, \ldots, x_d]$. The l-th* **elimination ideal** *J_l is the ideal of $\mathbb{F}_q[x_{l+1}, \ldots, x_d]$ defined by*

$$J_l = J \cap \mathbb{F}_q[x_{l+1}, \ldots, x_d] \tag{4.25}$$

The elimination ideal J_l has eliminated all the variables $x_1, \ldots, x_l$, i.e. it only contains polynomials with variables in $x_{l+1}, \ldots, x_d$. This shows that in finite fields, projection of the variety $V_d(\langle f_1, \ldots, f_s\rangle)$ from $\mathbb{F}_q^d$ to $\mathbb{F}_q^{d-l}$, is exactly the variety $V_{d-l}(\langle f_1, \ldots, f_s\rangle \cap \mathbb{F}_q[x_{l+1}, \ldots, x_d])$.

Elimination theory uses **elimination ordering** to systematically eliminate variables from a system of polynomial equations. We can generate elimination ideals by computing Gröbner bases using elimination orderings.

Theorem 4.6 *Elimination Theorem Let J be an ideal in $\mathbb{F}[x_1, \ldots, x_d]$ and let G be the Gröbner basis of J with respect to the lex order (elimination order) $x_1 > x_2 > \cdots > x_d$. Then, for every $0 \leq l \leq d$,*

$$G_l = G \cap \mathbb{F}[x_{l+1}, \ldots, x_d] \tag{4.26}$$

is a Gröbner basis of the l-th elimination ideal J_l.

This can be better understood using the following example.

Example 4.14 *Given the following equations in $\mathbb{R}[x, y, z]$*

$$\begin{aligned} x^2 + y + z &= 1 \\ x + y^2 + z &= 1 \\ x + y + z^2 &= 1 \end{aligned}$$

let I be the ideal generated by these equations:

$$I = \langle x^2 + y + z - 1, x + y^2 + z - 1, x + y + z^2 - 1\rangle$$

The Gröbner basis for I with respect to lex order $x > y > z$ is found to be $G = \{g_1, g_2, g_3, g_4\}$ where

$$\begin{aligned} g_1 &= x + y + z^2 - 1 \\ g_2 &= y^2 - y - z^2 + z \\ g_3 &= 2yz^2 + z^4 - z^2 \\ g_4 &= z^6 - 4z^4 + 4z^3 - z^2 \end{aligned}$$

Notice that while g_1 has variables in $\mathbb{R}[x, y, z]$, g_2 and g_3 only have variables in $\mathbb{R}[y, z]$ and g_4 only has variables in $\mathbb{R}[z]$. Thus, $G_1 = G \cap \mathbb{R}[y, z] = \{g_2, g_3, g_4\}$ and $G_2 = G \cap \mathbb{R}[z] = \{g_4\}$

Also notice that since g_4 only contains variable z, and since $g_4 = 0$, a solution for z can be obtained. This solution can then be applied to g_2 and g_3 to obtain solutions for y, and so on.

Elimination theory provides the basis for following abstraction approach.

4.3.2 Abstraction using Nullstellensatz and Gröbner Basis

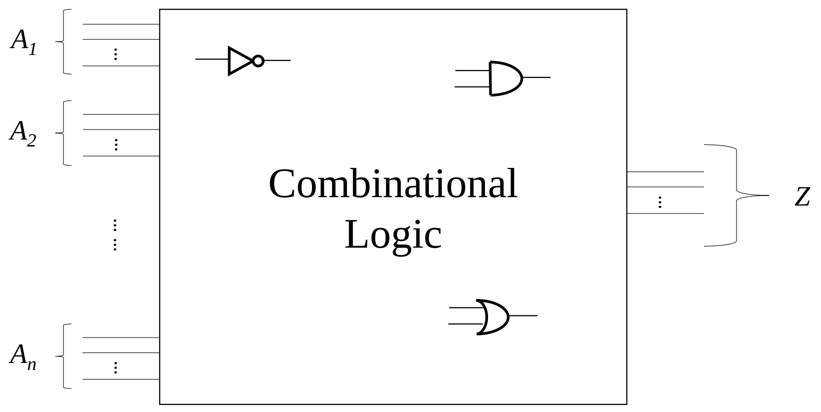


Figure 4.2: Galois field arithmetic circuit model for abstraction

Problem setup: Let S be the system of polynomials, $\{f_1, \dots, f_s, f_{A_1}, \dots, f_{A_n}, f_Z\} \subset \mathbb{F}_{2^k}$, derived from the hardware implementation of the Galois field arithmetic circuit over $\mathbb{F}_{2^k}$, as Figure 4.2 shows. This circuit performs some unknown function f over $\mathbb{F}_{2^k}$ in the form of $Z = \mathcal{F}(A_1, \dots, A_n)$, where Z is the k-bit output and $A_1, \dots, A_n$ are the k-bit inputs. The polynomial representation of $\mathcal{F}$ over $\mathbb{F}_{2^k}$ is thus:

$$f_{\mathcal{F}} : Z + \mathcal{F}(A_1, \dots, A_n)$$

Since $f_{\mathcal{F}}$ is ultimately derived from the circuit implementation, it agrees with the solution to the system of polynomials $\{S\} = 0$, i.e.:

$$f_1 = \cdots = f_s = f_{A_1} = \cdots = f_{A_n} = f_Z = 0$$

Thus, if we let $J = \langle f_1, \dots, f_s, f_{A_1}, \dots, f_{A_n}, f_Z \rangle$ be the ideal generated by S, $f_{\mathcal{F}}$ **vanishes** on the variety $V_{\mathbb{F}_{2^k}}(J)$. Therefore, due to Proposition 4.2, $f_{\mathcal{F}}$ must be contained in the ideal of polynomials that vanish on this variety, $f_{\mathcal{F}} \in I(V_{\mathbb{F}_{2^k}}(J))$.

By applying Strong Nullstellensatz over $\mathbb{F}_{2^k}$ (Theorem 4.3), $I(V_{\mathbb{F}_{2^k}}(J)) = J + J_0$ where J_0 is the ideal generated by all vanishing polynomials in $\mathbb{F}_{2^k}$. Recall that a vanishing polynomial in $\mathbb{F}_{2^k}[x]$ is $x^q - x = x^q + x$. In our case, $\{x_1, \dots, x_d\} \in \mathbb{F}_2$ and $\{A_1, \dots, A_n, Z\} \in \mathbb{F}_{2^k}$. Thus, for $\mathbb{F}_{2^k}[x_1, \dots, x_d, A_1, \dots, A_n, Z]$:

$$J_0 = \langle x_1^2 + x_1, \dots, x_d^2 + x_d, A_1^{2^k} + A_1, \dots, A_n^{2^k} + A_n, Z^{2^k} + Z \rangle$$

The generators of the ideal sum $J + J_0$ are simply the combination of the generators of J and the generators J_0.

The variety $V_{\mathbb{F}_q}(J)$ is the set of all consistent assignments to the nets (signals) in the circuit C. If we *project this variety on the word-level input and output variables of the circuit C, we essentially generate the function $\mathcal{F}$ implemented by the circuit.* Projection of varieties from d-dimensional space $\mathbb{F}_q^d$ onto a lower dimensional subspace $\mathbb{F}_q^{d-l}$ is equivalent to *eliminating l variables* from the corresponding ideal. This can be done by computing a Gröbner basis of the ideal with elimination ordering, as described in the Elimination Theorem (Theorem 4.6). Thus, we can find the polynomial $f_{\mathcal{F}} : Z + \mathcal{F}(A_1, \dots, A_n)$ by computing the Gröbner basis of $J + J_0$ using the proper elimination ordering.

The proposed elimination order for abstraction is defined as the **abstraction term order**.

Definition 4.25 *Given a circuit C, let $x_1, \dots, x_d$ denote all the bit-level variables, let $A_1, \dots, A_n$ denote the k-bit word-level inputs, and let Z denote the k-bit word-level output. Using any refinement of the partial variable order $\{x_1, \dots, x_d\} > Z > \{A_1, \dots, A_n\}$,* *impose a lex term order $>$ on the polynomial ring $R = \mathbb{F}_q[x_1, \ldots, x_d, Z, A_1, \ldots, A_n]$. This elimination term order $>$ is defined as the* **Abstraction Term Order (ATO)**. *The relative ordering among $x_1, \ldots, x_d$ is not important and can be chosen arbitrarily. Likewise, the relative ordering among $A_1, \ldots, A_n$ is also unimportant.*

Theorem 4.7 Abstraction Theorem: *Using the notations from problem setup at the beginning of this subsection, we compute a Gröbner basis G of ideal $(J + J_0)$ using the abstraction term order $>$. Then:*
(i) For every word-level input A_i, G must contain the vanishing polynomial $A_i^q - A_i$ as the only polynomial with A_i as its only variable;
(ii) G must contain a polynomial of the form $Z + \mathcal{G}(A_1, \ldots, A_n)$; and
(iii) $Z + \mathcal{G}(A_1, \ldots, A_n)$ is such that $\mathcal{F}(A_1, \ldots, A_n) = \mathcal{G}(A_1, \ldots, A_n), \forall A_1, \ldots, A_n \in \mathbb{F}_q$. In other words, $\mathcal{G}(A_1, \ldots, A_n)$ and $\mathcal{F}(A_1, \ldots, A_n)$ are equal as polynomial functions over $\mathbb{F}_q$.

Corollary 4.2 *By computing a* **reduced** *Gröbner basis G_r of $J + J_0$, G_r will contain one and only one polynomial in of the form $Z + \mathcal{G}(A_1, \ldots, A_n)$, such that $Z = \mathcal{G}(A_1, \ldots, A_n)$ is the* **unique, minimal, canonical** *representation of the function $\mathcal{F}$ implemented by the circuit.*

Example 4.15 *Consider a 2-bit multiplier over $\mathbb{F}_{2^2}$ with $P(x) = x^2 + x + 1$, given in Figure 4.3. Variables a_0, a_1, b_0, b_1 are primary inputs, z_0, z_1 are primary outputs, and c_0, c_1, c_2, c_3, r_0 are intermediate variables.*

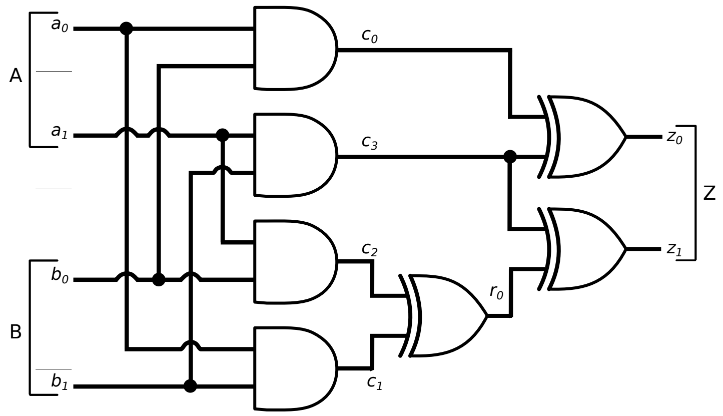


Figure 4.3: A 2-bit multiplier over $\mathbb{F}_{2^2}$.

Polynomials extracted from the circuit implementation represent the ideal J. Along with the ideal of vanishing polynomials J_0, the following polynomials represent the generators of $J + J_0$ for the multiplier circuit.

$$\left.\begin{aligned} f_1 &: c_0 + a_0 \cdot b_0 \\ f_2 &: c_1 + a_0 \cdot b_1 \\ f_3 &: c_2 + a_1 \cdot b_0 \\ f_4 &: c_3 + a_1 \cdot b_1 \\ f_5 &: r_0 + c_1 + c_2 \\ f_6 &: z_0 + c_0 + c_3 \\ f_7 &: z_1 + r_0 + c_3 \end{aligned}\right\} \quad \textit{Bit-level circuit constraints } (\subset J)$$

$$\left.\begin{aligned} f_A &: A + a_0 + a_1 \cdot \alpha \\ f_B &: B + b_0 + b_1 \cdot \alpha \\ f_Z &: Z + z_0 + z_1 \cdot \alpha \end{aligned}\right\} \quad \textit{Word-level designation } (\subset J)$$

$$\left.\begin{aligned} a_0^2 - a_0,\ a_1^2 - a_1,\ b_0^2 - b_0,\ b_1^2 - b_1 \\ c_0^2 - c_0,\ c_1^2 - c_1,\ c_2^2 - c_2,\ c_3^2 - c_3 \\ r_0^2 - r_0,\ z_0^2 - z_0,\ z_1^2 - z_1 \\ A^4 - A,\ B^4 - B,\ Z^4 - Z \end{aligned}\right\} \quad \textit{vanishing polynomials}(J_0)$$

We apply abstraction term order $>$, i.e a lex order with "bit-level variables" $>$ "Output Z" $>$ "Inputs A, B".

When we compute the reduced Gröbner basis, G_r, of $\{J + J_0\}$ with respect to this ordering, G_r = $\{g_1, \dots, g_{14}\}$:

$$g_1 : B^4 + B;\;\; g_2 : b_0 + b_1\alpha + B;\;\; g_3 : a_0 + a_1\alpha + A;$$
$$g_4 : c_0 + c_1\alpha + c_2\alpha + c_3(\alpha + 1) + Z; g_5 : r_0 + c_1 + c_2;\;\; g_6 : z_0 + c_0 + c_3;$$
$$g_7 : z_1 + r_0 + c_3;\;\; \mathbf{g_8} : \mathbf{Z} + \mathbf{A} \cdot \mathbf{B};\;\; g_9 : b_1 + B^2 + B;\;\; g_{10} : a_1 + A^2 + A;$$
$$g_{11} : c_3 + a_1 \cdot b_1 g_{12} : c_2 + a_1 \cdot b_1\alpha + a_1 \cdot B;\;\; g_{13} : c_1 + a_1 \cdot b_1\alpha + b_1 A;\;\; g_{14} : A^4 + A$$

$g_8 = Z + A \cdot B$ *is the* **canonical, word-level polynomial** *representing the function performed by the multiplier $Z = A \cdot B$.*

4.4 Concluding Remarks

Our approach to word-level abstraction of Galois field arithmetic circuits applies concepts of polynomial ideals, varieties, Gröbner basis, and abstraction theory to implement verifications on sequential circuits. These approaches are described in the following chapters.

CHAPTER 5

WORD-LEVEL TRAVERSAL OF FINITE STATE MACHINES USING ALGEBRAIC GEOMETRY

Reachability analysis is a basic component of sequential circuit verification, especially for formal equivalence checking and model checking techniques. Concretely, in modern synthesis tools, in order to improve various performance indicators such as latency, clock skew or power density, sequential optimization techniques such as retiming [25], scan logic [26], sequential power optimization [27] and clock-gating techniques [28]. These modifications may introduce malfunctions to the original logic and cause problems. Based on traditional localized simulation or formal verification method (e.g. equivalence checking), designers are reluctant to make aggressive optimization since the malfunctions are considered as "faults" in this circuit. However, if the circuit behavior is carefully investigated, it may become evident that those "faults" will never be activated/excitated during a restricted execution starting from legal initial states and with legal inputs. Thus we will call those "faults" as **spurious faults** (false negatives), since they will not affect the circuit's normal behavior.

Almost all practical sequential logic components can be modeled as finite state machines (FSMs). If we apply constraints upon the machine to make it start from designated initial states, and take specific legal inputs, a set of reachable states can be derived. As long as the "faults" can be modeled as "bad states", we can judge whether they are spurious faults by checking if they belong to the reachable states. From the spurious fault validation perspective, reachability analysis is a must when developing full set of sequential circuit verification techniques.

There are quite a few methods to perform reachability checking on FSMs. One among them is state space traversal. Conventionally the algorithm is based on bit-level techniques such as binary decision diagrams (BDDs) and Boolean logic. We propose a new traversal algorithm on word-level, which brings critical advantages. In this chapter the approach will be described and discussed in depth, with examples and experiments showing its feasibility when applied on general circuit benchmarks.

5.1 Motivation

Before introducing the details of our approach, there are a couple of questions to ask: why is it necessary to execute word-level FSM traversal? Why does algebraic geometry make this happen? The answers can be found in this section, as statement of the motivation of our research.

5.1.1 FSM Traversal Algorithms

Sequential circuits are modeled as FSMs, which can be implemented as graph. Thus a graph-traversal based algorithm is created to analysis the reachable states [29]. A traversal algorithm using concept "implicit state enumeration" is proposed [30]. Concretely, the algorithm is written as follows:

Algorithm 4: FSM Traversal using Implicit State Enumeration [31]

```
Input: Sequential circuit with given number of registers
Output: Scan registers listed in decreasing order of their non-controllability
$from^0 = reached = S^0$;
$i = 0$;
while TRUE do
    $i++$;
    $to^i$ =IMAGE($\Delta, from^{i-1}$);
    $new^i = to^i \cdot \overline{reached}$;
    for each state variable $r_j$ do
        record_if_transitions_present_or_missing($r_j, new^i$);
        compute_degree_of_unsettability($r_j, new^i$);
    end
    if $new^i == 0$ then
        break;
    end
    $reached = reached + new^i$;
    $from^i = new^i$;
end
return $reached$
```

Above algorithm describes a breath-first-search (BFS) traversal in state space. The traversal algorithm is a simple variation of BFS algorithm where states are nodes and transitions are arcs. Each state is uniquely encoded by a combination of a set of register data, which is usually represented by a Boolean vector.

Since a typical sequential circuit usually contains a combinational logic component, the traversal algorithm analyzes the combinational logic and derives the transition function for one-step reachability within current time-frame, and extends the result to complete execution paths through unrolling. If each state encoding (i.e. exact values in the selected registers) is explicitly analyzed and counted during the unrolling procedure, this unrolling is called **explicit unrolling**. In the BFS traversal algorithm, the states cannot be directly read in the execution; instead, they are implicitly represented using a conjunction of several Boolean formulas. Such techniques differs from explicit unrolling are called **implicit unrolling**.

However, the BFS traversal algorithm proposed by the author is usually not practical. The conjunctions of Boolean formulas are stored as BDDs, which is a canonical and convenient structure. Nevertheless, the size of BDD explodes when the formulas become too long and too complicated. In [30], the authors make a compromise between accuracy and cost, and turn to approximate reachability. In this research work, the aim is to explore a word-level technique which can make a accurate reachability analysis available.

5.1.2 Word-level Data Flow on Modern Datapath Designs

The level of integration of modern digital circuit designs is very high. For example, processor A10 designed by Apple integrates 3.3 billion transistors on a $125\ mm^2$ chip [32]. Such a high density makes the silicon implementation of large datapaths possible. In recent decades, 64-bit or even larger datapaths frequently appear in modern digital IC designs such as powerful central processing units (CPUs) and high bandwidth memory (HBM). Meanwhile, with the development of electronic design automation (EDA) tools, data flow is described by the designer as word-level specifications. Therefore, it will be straightforward and beneficial for users if formal verification tools can work on word-level. Moreover, adopting word-level techniques will greatly reduce the state space and make verification more efficient.

In order to throw light on the advantages using word-level techniques, we pick a typical digital circuit component in modern 64-bit MIPS processor as an example.

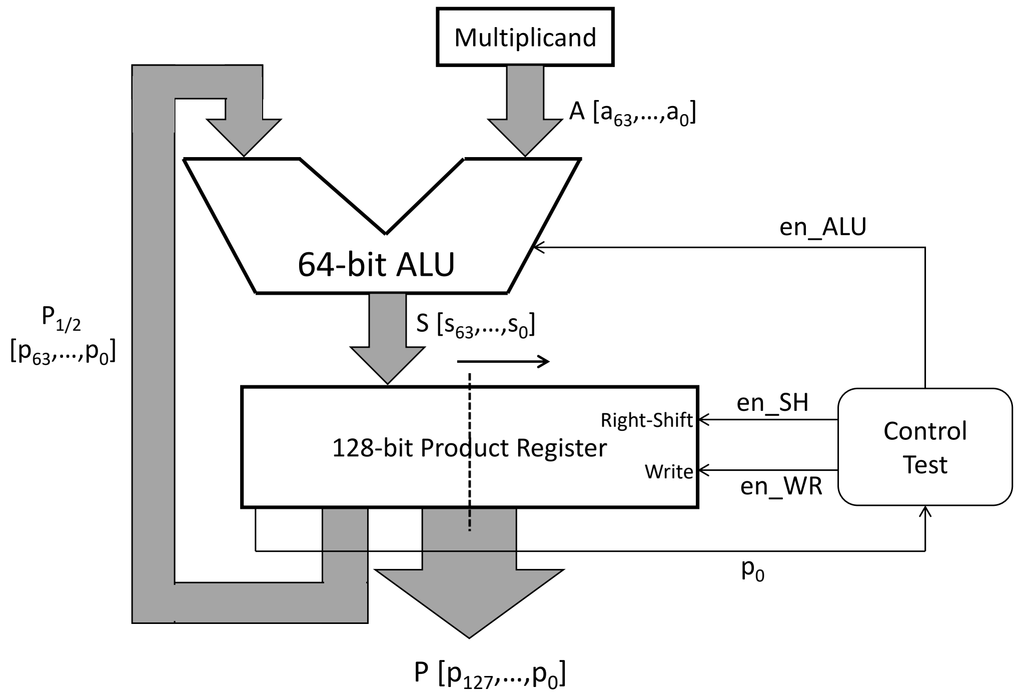


Figure 5.1: A 64-bit sequential multiplication hardware design

Example 5.1 *Figure 5.1 depicts a sequential multiplication hardware implementation within a 64-bit MIPS. Initially, one multiplicand is preloaded to the lower 64 bits of the product registers. Iteratively, the last significant bit (LSB) of current (temporary) product is used as flag to activate the ALU to add on the other multiplicand. For each iteration the data in product registers shifts right by 1 bit. Finally when the most significant bit (MSB) of preloaded multiplicand arrives at the MSB of product registers, the registers contains the result – 128 bits product. The behavior can be described by the following algorithm:*

Algorithm 5: Sequential multiplication hardware in 64-bit MIPS

```
Input: Multiplicand $A, B$
Output: Product $C$
  Preload $B$ into lower 64-bit of Product Register $P$;
  repeat
      if Last Bit of Product Register $LSB(P) == 1$ then
          $P = P_{1/2} + B$;
      end
      Right shift $P$;
  until 64 Repetitions;
  return $C = P$
```

Traditionally, to verify the functional correctness of this multiplier, satisfiability (SAT) based or BDD based model checking is applied on basic function units. For example, as a part of functional verification, we would like to check "$P = P_{1/2} + B$" is correctly executed. Then in a model checker we need to add following specifications:

$$
\begin{aligned}
& en\_ALU \\
& \wedge s_0 = a_0 \oplus p_0 \\
& \wedge s_1 = a_1 \oplus p_1 \oplus (a_0 \wedge p_0) \\
& \wedge s_2 = a_2 \oplus p_2 \oplus ((a_1 \oplus p_1) \wedge a_0 \wedge p_0 \vee a_0 \wedge p_0) \\
& \wedge \cdots \\
& \wedge s_{63} = a_{63} \oplus p_{63} \oplus (c_{63})
\end{aligned}
$$

We can see that when checking a single part of the whole structure, the number of clauses needed will increase to $k+1$ when using k-bit datapath. Considering the formula representing carry-in will become longer and longer, the final conjunction of all clauses will contain $O(2^k)$ Boolean operators. If by some means we can write the specification with only 3 variables

$$S = P_{1/2} + B \tag{5.1}$$

Since the abstraction to word-level reduces symbolic storage and execution cost, the complexity to traverse the state space is greatly reduced.

On the other hand, when implementation details of the datapath are not available, it is not convenient any more for users of conventional model checker. The reason is that the user has to write all clauses for the implementation, which contains cross-literals, e.g. s_2 may associate with a_1, p_0, etc. If the user is not familiar with the implementation of this adder, those cross-literals will bring confusions. However, if word-level techniques allow specification like Equation 5.1, the verification tool will be very user-friendly and straightforward even if the implementation details are in a black box.

5.1.3 On the Existence of Word-level Abstraction

When given a bit-level netlist, the prerequisites to use word-level techniques are to convert bit-level to word-level first. This conversion is usually completed by abstraction techniques.

An old but universally effective abstraction method is *Lagrange's interpolation*. Instead of using it in real algebra, it can also be extended to Galois field. Here we use an example to illustrate the conversion in Galois field using Lagrange's interpolation.

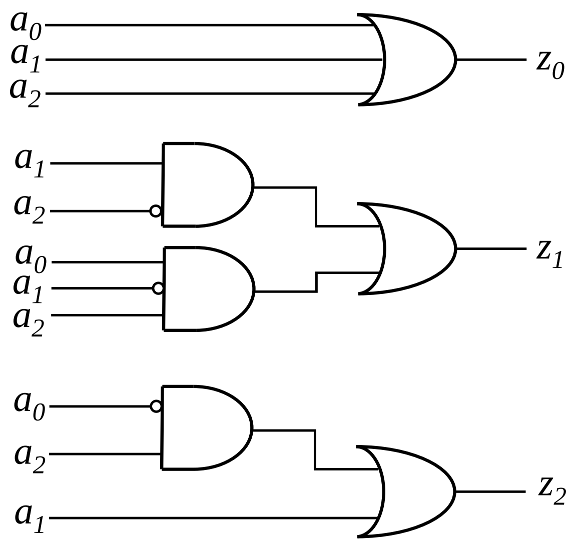


Figure 5.2: Gate-level netlist for Lagrange's interpolation example

Example 5.2 (Lagrange's interpolation) *Assume we are given gate-level netlist shown* *in Figure 5.2. It can be written as 3 Boolean equations:*

$$z_0 = a_0 \vee a_1 \vee a_2$$

$$z_1 = a_1 \wedge \neg a_2 \vee a_0 \wedge \neg a_1 \wedge a_2$$

$$z_2 = a_1 \vee \neg a_0 \wedge a_2$$

Define 2 word-level variables A, Z as input and output:

$$A = \{a_2 a_1 a_0\},\ Z = \{z_2 z_1 z_0\}$$

To convert bit-level to word-level, we need to find mapping $\mathbb{B}^3 \to \mathbb{B}^3$, or $\mathbb{F}_{2^3} \to \mathbb{F}_{2^3}$. The latter one, as mentioned in preliminaries, is a polynomial function in $\mathbb{F}_{2^3}$.

For each element in $\mathbb{F}_{2^3}$, we write down the truth table as follows:

$\{a_2a_1a_0\} \in \mathbb{B}^3$	$A \in \mathbb{F}_{2^3}$	$\to$	$\{z_2z_1z_0\} \in \mathbb{B}^3$	$Z \in \mathbb{F}_{2^3}$
000	0	$\to$	000	0
001	1	$\to$	001	1
010	α	$\to$	111	$\alpha^2+\alpha+1$
011	$\alpha+1$	$\to$	111	$\alpha^2+\alpha+1$
100	α^2	$\to$	101	α^2+1
101	α^2+1	$\to$	011	$\alpha+1$
110	$\alpha^2+\alpha$	$\to$	101	α^2+1
111	$\alpha^2+\alpha+1$	$\to$	101	α^2+1

Table 5.1: Truth table for mappings in $\mathbb{B}^3$ and $\mathbb{F}_{2^3}$

Now our objective is to abstract a function over GF $\mathbb{F}_{2^3}$ about word-level variables, i.e. $Z = \mathcal{F}(A)$. Recall the definition of Lagrange's interpolation:

$$\mathcal{F}(x) = \sum_{k=1}^{N} \left[\prod_{(0 \le j \le k-1),(j \ne i)} \frac{x - x_j}{x_i - x_j} \cdot y_k \right] \tag{5.2}$$

The geometric meaning of Lagrange's interpolation in real algebra is: given N dots with coordinates (x_i, y_i), they can always be fitted into a polynomial function with at most $N-1$ degree, and that function can be written in the form of Equation 5.2. In this example, although defined in a Galois field instead of the real number field, the essential *concept of Lagrange's interpolation remains the same. We can get 8 points in the affine space:*

$$\begin{aligned}
\textit{Generic form} &: (a_2\alpha^2 + a_1\alpha + a_0, z_2\alpha^2 + z_1\alpha + z_0) \leftarrow (A, Z)\\
\textit{Point}\ 1 &: (0, 0) \leftarrow (000, 000)\\
\textit{Point}\ 2 &: (1, 1) \leftarrow (001, 001)\\
\textit{Point}\ 3 &: (\alpha, \alpha^2 + \alpha + 1) \leftarrow (010, 111)\\
\textit{Point}\ 4 &: (\alpha + 1, \alpha^2 + \alpha + 1) \leftarrow (011, 111)\\
\textit{Point}\ 5 &: (\alpha^2, \alpha^2 + 1) \leftarrow (100, 101)\\
\textit{Point}\ 6 &: (\alpha^2 + 1, \alpha + 1) \leftarrow (101, 011)\\
\textit{Point}\ 7 &: (\alpha^2 + \alpha, \alpha^2 + 1) \leftarrow (110, 101)\\
\textit{Point}\ 8 &: (\alpha^2 + \alpha + 1, \alpha^2 + 1) \leftarrow (111, 101)
\end{aligned}$$

Substitute 8 (x_i, y_i) pairs in Equation 5.2 with these 8 points in $\mathbb{F}_{2^3}$. The result is a polynomial function with degree no greater than 7:

$$\begin{aligned}
Z =& \mathcal{F}(A)\\
=& (\alpha^2 + \alpha + 1)A^7 + (\alpha^2 + 1)A^6 + \alpha A^5 + (\alpha + 1)A^4\\
& + (\alpha^2 + \alpha + 1)A^3 + (\alpha^2 + 1)A
\end{aligned}$$

The Lagrange's interpolation theorem also proves the existence of a word-level abstraction for a bit-level netlist. In practice, Lagrange's interpolation is not scalable. The reason is that it needs entire function (state space), but usually we only have the circuit representation of the FSM. Considering this fact, a symbolic method is needed. Our approach in this section uses abstraction based on Gröbner basis with abstraction term order (ATO), which is briefly introduced in Section 4.3.

5.1.4 Significance of Developing Word-level FSM Traversal

At the end of this section, we summarize each subsection and conclude the motivation of our research. First, we start from the basic FSM traversal algorithms and exhibit their difficulties to overcome. Secondly, we state the observation that word-level description is increasingly important and common in characterizing the data flow on modern large-size datapaths. Last but not least, we give instances that some prerequisites for supporting word-level techniques are already available.

To conclude, first we state the importance of performing FSM traversal in sequential circuit reachability analysis. To overcome the vast cost brought by searching in a large space, we propose to use word-level polynomials to represent the states and transition relations. As a result, states are categorized into sets represented by a small number of word-level polynomials (more specifically, the varieties to the polynomial ideals), and multiple transition relations are therefore merged together. All of these efforts reduces the space cost of state space, meanwhile lower the time complexity to traverse such a state space. By using Lagrange's interpolation, we prove the feasibility of word-level techniques on general circuits' verification. Thus we cover both the necessity and sufficiency of developing such a word-level traversal technique in our work. They are also answers to the questions at the beginning of this section.

5.2 FSM Reachability using Algebraic Geometry

We use symbolic state reachability with algebraic geometry concepts. It is an abstraction based on word operand definition of datapaths in circuits, and it can be applied to arbitrary FSMs by bundling a set of bit-level variables together as one or several word-level variables. The abstraction polynomial, encoding the reachable state space of the FSM, is obtained through computing a GB over $\mathbb{F}_{2^k}$ of the polynomials of the circuit using an elimination term order based on Theorem 4.6.

5.2.1 FSM model for sequential circuits

A finite state machine (FSM) is a mathematical model of computation for designing and analyzing sequential logic circuits. If a FSM's primary outputs depend on primary inputs and present state inputs, it is named as a *Mealy machine*; the formal definition is as follows:

Definition 5.1 *A Mealy machine is an n-tuple $\mathcal{M} = (\Sigma, O, S, S^0, \Delta, \Lambda)$ where*

- *Σ is the input label, O is the output label;*

- *S is the set of states, $S^0 \subseteq S$ is the set of initial states;*
- *$\Delta :\ S \times \Sigma \to S$ is the next state transition function;*
- *$\Lambda :\ S \times \Sigma \to O$ is the output function.*

The other kind of FSM is *Moore machine*, its difference from Mealy machine is that its primary outputs only depend on the present states, i.e. the output function is defined as

$$\Lambda :\ S \to O$$

Typical sequential circuits can be depicted as Figure 5.3(a). Primary inputs $x_1, \ldots, x_m \in \Sigma$, and primary outputs $z_1, \ldots, z_n \in O$. Signals $s_1, \ldots, s_k$ are present state (PS) variables, $t_1, \ldots, t_k$ are next state (NS) variables. We can define 2 k-bit words denoting the PS/NS variables as there are k flip-flops in the datapath: $S = (s_1, \ldots, s_k),\ T = (t_1, \ldots, t_k)$. Transition function at bit level are defined as Δ_i:

$$t_i = \Delta_i(s_1, \ldots, s_k, x_1, \ldots, x_m)$$

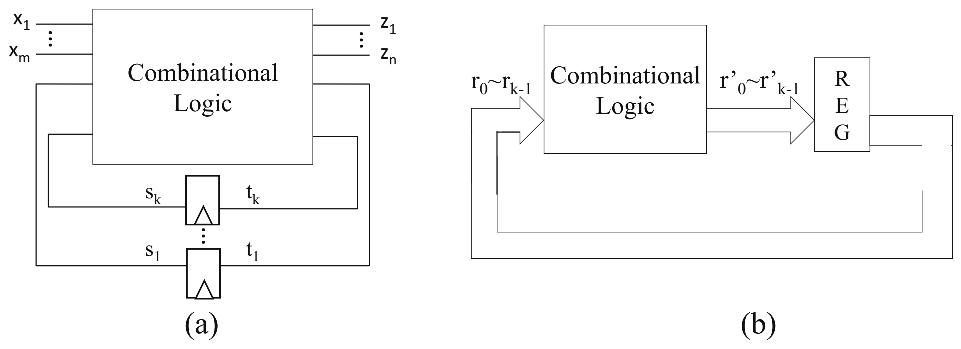


Figure 5.3: FSM models of sequential circuits

In some cases, arithmetic computations are implemented as Moore machines where input operands are loaded into register files R and the FSM is executed for k clock cycles. We can simplify them to the model in Figure 5.3(b).

5.2.2 Conventional Traversal Method

Conceptually, the state-space of a FSM is traversed in a breadth-first manner, as shown in Algorithm 6. The algorithm operates on the FSM $\mathcal{M} = (\sum, O, S, S^0, \Delta, \Lambda)$ underlying a sequential circuit. In such cases, the transition function Δ and the initial states are represented and manipulated using Boolean representations such as BDDs or SAT solvers. The variables $from, reached, to, new$ represent characteristic functions of sets of states. Starting from the initial state $from^i = S^0$, the algorithm computes the states reachable in 1-step from $from^i$ in each iteration. In line 4 of Algorithm 6, the *image computation* is used to compute the reachable states in every execution step.

The *transition function* Δ is given by Boolean equations of the flip-flops of the circuit: $t_i = \Delta_i(s, x)$, where t_i is a next state variable, s represents the present state variables and x represents the input variables. The *transition relation of the FSM* is then represented as:

$$T(s, x, t) = \prod_{i=1}^{n} (t_i \overline{\oplus} \Delta_i) \tag{5.3}$$

where n is the number of flip flops, and $\overline{\oplus}$ is XNOR operation. Let $from$ denote the set of initial states, then the image of the initial states, under the transition function Δ is finally computed as:

$$to = \text{Img}(\Delta, from) = \exists_s \, \exists_x \, [T(s, x, t) \cdot from] \tag{5.4}$$

Here, $\exists x(f)$ represents the *existential quantification of f w.r.t. variable x*. In Boolean logic, this operator is implemented as

$$\exists x(f) = f_x \vee f_{\overline{x}}$$

Let us describe the application of the algorithm on the FSM circuit of Figure 5.4. *We will first describe its operation at the Boolean level, and then describe how this algorithm can be implemented using algebraic geometry at word level.*

Algorithm 6: BFS Traversal for FSM Reachability

```
Input: Transition functions $\Delta$, initial state $S^0$
  $from^0 = reached = S^0$;
  repeat
      $i \leftarrow i + 1$;
      $to^i \leftarrow$Img$(\Delta, from^{i-1})$;
      $new^i \leftarrow to^i \cap \overline{reached}$;
      $reached \leftarrow reached \cup new^i$;
      $from^i \leftarrow new^i$;
  until $new^i == 0$;
  return $reached$
```

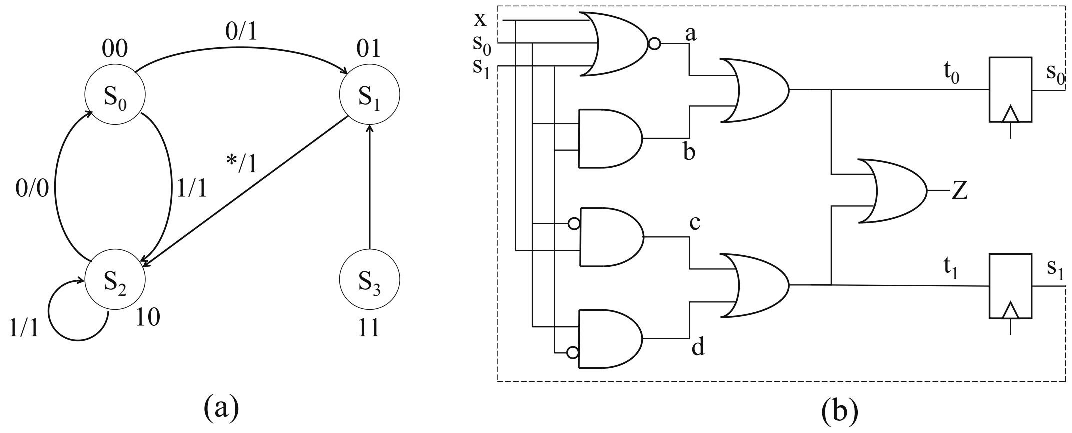


Figure 5.4: The example FSM and the gate-level implementation.

In Line 1 of the BFS algorithm, assume that the initial state is S_3 in Figure 5.4(b), which is encoded as $S_3 = \{11\}$. Using Boolean variables s_0, s_1 for the present states, $from^0 = s_0 \cdot s_1$ is represented as a Boolean formula.

Example 5.3 *For the circuit in Figure 5.4(b), we have the transition functions of the machine as:*

$$\Delta_1 : t_0 \overline{\oplus} ((\overline{x \vee s_0 \vee s_1}) \vee s_0 s_1)$$

$$\Delta_2 : t_0 \overline{\oplus} (\overline{s_0} x \vee \overline{s_1} s_0)$$

$$from : from^0 = s_0 \cdot s_1$$

When the formula of Equation 5.4 is applied to compute 1-step reachability, $to = \exists_{s_0,s_1,x}(\Delta_1 \cdot \Delta_2 \cdot from^0)$, we obtain $to = \overline{t_0} \cdot t_1$, which denotes the state $S_1 = \{01\}$ reached in 1-step from S_3. In the next iteration, the algorithm uses state $S_1 = \{01\}$ as the current (initial) state, and computes $S_2 = \{10\} = t_0 \cdot \overline{t_1}$ as the next reachable state, and so on.

Our objective is to model the transition functions Δ as a polynomial ideal J, and to perform the image computations using Gröbner bases over Galois fields. *This requires to perform quantifier elimination; which can be accomplished using the GB computation over $\mathbb{F}_{2^k}$ using elimination ideals* [22]. Finally, the set union, intersection and complement operations are also to be implemented in algebraic geometry.

5.2.3 FSM Traversal at word-level over $\mathbb{F}_{2^k}$

The state transition graph (STG) shown in Figure 5.4(a) uses a 2-bit Boolean vector to represent 4 states $\{S_0, S_1, S_2, S_3\}$. We map these states to elements in $\mathbb{F}_{2^2}$, where $S_0 = 0, S_1 = 1, S_2 = \alpha, S_3 = \alpha + 1$. Here, we take $P(X) = X^2 + X + 1$ as the irreducible polynomial to construct $\mathbb{F}_4$, and $P(\alpha) = 0$ so that $\alpha^2 + \alpha + 1 = 0$.

Initial state: In Line 1 of Algorithm 6, the initial state is specified by means of a corresponding polynomial $f = \mathcal{F}(S) = S - 1 - \alpha$. Notice that if we consider the ideal generated by the initial state polynomial, $I = \langle f \rangle$, then its variety $V(I) = 1 + \alpha$ corresponds to the state encoding $S_3 = \{11\} = 1 + \alpha$, where a polynomial in word-level variable S encodes the initial state.

Set operations: In Lines 5 and 6 of Algorithm 6, we need **union**, **intersection** and **complement** of varieties over $\mathbb{F}_{2^k}$, for which we again use algebraic geometry concepts.

Definition 5.2 *(***Sum/Product of Ideals** *[17]) If $I = \langle f_1, \dots, f_r \rangle$ and $J = \langle g_1, \dots, g_s \rangle$ are ideals in R, then the* **sum** *of I and J is defined as $I + J = \langle f_1, \dots, f_r, g_1, \dots, g_s \rangle$. Similarly, the* **product** *of I and J is $I \cdot J = \langle f_i g_j \mid 1 \le i \le r, 1 \le j \le s \rangle$.*

Theorem 5.1 *If I and J are ideals in R, then $\mathbf{V}(I + J) = \mathbf{V}(I) \bigcap \mathbf{V}(J)$ and $\mathbf{V}(I \cdot J) = \mathbf{V}(I) \bigcup \mathbf{V}(J)$.*

In Line 5 of Algorithm 6, we need to compute the complement of a set of states. Assume that J denotes a polynomial ideal whose variety $V(J)$ denotes a set of states. We require the computation of another polynomial ideal J', such that $V(J') = \overline{V(J)}$. We show that this computation can be performed using the concept of **ideal quotient**:

Definition 5.3 *(**Quotient of Ideals**) If I and J are ideals in a ring R, then $I : J$ is the set $\{f \in R \mid f \cdot g \in I, \forall g \in J\}$ and is called the* **ideal quotient** *of I by J.*

Example 5.4 *In $\mathbb{F}_q[x, y, z]$, ideal $I = \langle xz, yz\rangle$, ideal $J = \langle z\rangle$. Then*

$$\begin{aligned} I : J &= \{f \in \mathbb{F}_q[x, y, z] \mid f \cdot z \in \langle xz, yz\rangle\} \\ &= \{f \in \mathbb{F}_q[x, y, z] \mid f \cdot z = Axz + Byz\} \\ &= \{f \in \mathbb{F}_q[x, y, z] \mid f = Ax + By\} \\ &= \langle x, y\rangle \end{aligned}$$

We can now obtain the complement of a variety through the following results which are stated and proved below:

Lemma 5.1 *Let $f, g \in \mathbb{F}_{2^k}[x]$, then $\langle f : g\rangle = \left\langle \frac{f}{gcd(f,g)} \right\rangle$.*

Proof. Let $d = gcd(f, g)$. So, $f = df_1, g = dg_1$ with $gcd(f_1, g_1) = 1$. Note that $f_1 = \frac{f}{gcd(f,g)}$.

Take $h \in \langle f : g\rangle$. According to the Definition 5.3, $hg \in \langle f\rangle$, which means $hg = f \cdot r$ with $r \in \mathbb{F}_{2^k}[x]$. Therefore, $hdg_1 = df_1r$ and $hg_1 = f_1r$. But considering $gcd(g_1, f_1) = 1$ we have the fact that f_1 divides h. Hence $h \in \langle f_1\rangle$.

Conversely, let $h \in \langle f_1\rangle$. Then $h = s \cdot f_1$, where $s \in \mathbb{F}_{2^k}[x]$. So, $hg = hdg_1 = sf_1dg_1 = sg_1f \in \langle f\rangle$. Therefore, $h \in \langle f : g\rangle$. ■

Theorem 5.2 *Let J be an ideal generated by a single univariate polynomial in variable x over $\mathbb{F}_{2^k}[x]$, and let the vanishing ideal $J_0 = \langle x^{2^k} - x\rangle$. Then*

$$V(J_0 : J) = V(J_0) - V(J),$$

where all the varieties are considered over the field $\mathbb{F}_{2^k}$.

Proof. Since $\mathbb{F}_{2^k}[x]$ is a principal ideal domain, $J = \langle g \rangle$ for some polynomial $g \in \mathbb{F}_{2^k}[x]$. Let $d = gcd(g, x^{2^k} - x)$. So, $g = dg_1, x^{2^k} - x = df_1$, with $gcd(f_1, g_1) = 1$. Then $J_0 : J = \langle f_1 \rangle$ by Lemma 5.1.

Let $x \in V(J_0) - V(J)$. From the definition of set complement, we get $x \in \mathbb{F}_{2^k}$ while $g(x) \neq 0$.

Since $x^{2^k} = x$, we see that either $d(x) = 0$ or $f_1(x) = 0$. Considering $g(x) \neq 0$, we can assert that $d(x) \neq 0$. In conclusion, $f_1(x) = 0$ and $x \in V(f_1)$.

Now let $x \in V(f_1)$, we get $f_1(x) = 0$. So, $x^{2^k} - x = 0$ gives $x \in V(J_0) = \mathbb{F}_{2^k}$ which contains all elements in the field.

Now we make an assumption that $x \in V(g)$. Then $g(x) = 0 = d(x)g_1(x)$ which means either $d(x) = 0$ or $g_1(x) = 0$.

If $g_1(x) = 0$, then since $f_1(x) = 0$ we get that f_1, g_1 share a root. This contradicts the fact that $gcd(f_1, g_1) = 1$.

On the other hand, if $d(x) = 0$, then since $f_1(x) = 0$ and $x^{2^k} - x = df_1$, we get that $x^{2^k} - x$ has a double root. But this is impossible since the derivative of $x^{2^k} - x$ is -1.

So, $x \notin V(g)$ and this concludes the proof. ■

Let $x^{2^k} - x$ be a vanishing polynomial in $\mathbb{F}_{2^k}[x]$. Then $V(x^{2^k} - x) = \mathbb{F}_{2^k}$ i.e. the variety of vanishing ideal contains all possible valuations of variables, so it constitutes the **universal set**. Subsequently, based on Theorem 5.2, the **absolute complement** $V(J')$ of a variety $V(J)$ can be computed as:

Corollary 5.1 *Let $J \subseteq \mathbb{F}_{2^k}[x]$ be an ideal, and $J_0 = \langle x^{2^k} - x \rangle$. Let J' be an ideal computed as $J' = J_0 : J$. Then*

$$V(J') = \overline{V(J)} = V(J_0 : J)$$

With Corollary 5.1, we are ready to demonstrate the concept of word-level FSM traversal over $\mathbb{F}_{2^k}$ using algebraic geometry. The algorithm is given in Algorithm 7. Note that in the algorithm, $from^i, to^i, new^i$ are *univariate polynomials in variables S or T* only, due to the fact that they are the result of a GB computation with an elimination term order, where the bit-level variables are abstracted and quantified away.

Algorithm 7: Algebraic Geometry based FSM Traversal

```
Input: The circuit's characteristic polynomial ideal $J_{ckt}$, initial state polynomial
       $\mathbb{F}(S)$, and LEX term order: bit-level variables $x, s, t >$ PS word $S >$ NS
       word $T$
  $from^0 = reached = \mathbb{F}(S)$;
  repeat
     $i \leftarrow i + 1$;
     $G \leftarrow$GB($\langle J_{ckt}, J_v, from^{i-1}\rangle$);
     /* Compute Gröbner basis with elimination term
        order:  $T$ smallest                                   */
     $to^i \leftarrow G \cap \mathbb{F}_{2^k}[T]$;
     /* There will be a univariate polynomial in $G$
        denoting the set of next states in word-level
        variable $T$                                           */
     $\langle new^i\rangle \leftarrow \langle to^i\rangle + (\langle T^{2^k} - T\rangle : \langle reached\rangle)$;
     /* Use quotient of ideals to attain complement of
        reached states, then use sum of ideals to attain
        an intersection with next state                        */
     $\langle reached\rangle \leftarrow \langle reached\rangle \cdot \langle new^i\rangle$;
     /* Use product of ideals to attain a union of newly
        reached states and formerly reached states             */
     $from^i \leftarrow new^i(S \setminus T)$;
     /* Start a new iteration by replacing variable $T$ in
        newly reached states with current state variable
        $S$                                                    */
  until $\langle new^i\rangle == \langle 1\rangle$;
  /* Loop until a fixpoint reached:  newly reached state
     is empty                                                  */
  return $\langle reached\rangle$
```

5.2.4 Word-level FSM Traversal Example

Example 5.5 *We apply Algorithm 7 to the example shown in Figure 5.4 to execute the FSM traversal. Let the initial state* $from^0 = \{00\}$ *in* $\mathbb{B}^2$ *or* $0 \in \mathbb{F}_4$*. Polynomially, it is written as* $from^0 = S - 0$*. In the first iteration, we compose an ideal* J *with*

$$f_1 : t_0 - (xs_0s_1 + xs_0 + xs_1 + x + s_0 + s_1 + 1)$$

$$f_2 : t_1 - (xs_0 + x + s_0s_1 + s_0)$$

$$f_3 : S - s_0 - s_1\alpha; \quad f_4 : T - t_0 - t_1\alpha$$

$J_{ckt} = \langle f_1, f_2, f_3, f_4 \rangle$, and the vanishing polynomials:

$$f_5 : x^2 - x; \;\; f_6 : s_0^2 - s_0, \;\; f_7 : s_1^2 - s_1$$
$$f_8 : t_0^2 - t_0, \;\; f_9 : t_1^2 - t_1; \;\; f_{10} : S^4 - S, \;\; f_{11} : T^4 - T$$

with $J_v = \langle f_5, f_6, \ldots, f_{11} \rangle$.

Compute $G = GB(J)$ for $J = J_{ckt} + J_0 + \langle from^0 \rangle$, with an elimination term order

$$\underbrace{\{x, s_0, s_1, t_0, t_1\}}_{\text{all bit-level variables}} > \underbrace{S}_{\text{(PS word)}} > \underbrace{T}_{\text{(NS word)}} .$$

The resulting GB G contains a polynomial generator with only T as the variable. In Line 5, assign it to the next state

$$to^1 = T^2 + (\alpha + 1)T + \alpha.$$

Note that the roots or variety of $T^2 + (\alpha+1)T + \alpha$ is $\{1, \alpha\}$, denoting the states $\{01, 10\}$.

Since the formerly reached state "$reached = T$", its complement is computed using Corollary 5.1

$$\langle T^4 - T \rangle : \langle T \rangle = \langle T^3 + 1 \rangle.$$

$V(\langle T^3 + 1 \rangle) = \{1, \alpha, \alpha+1\}$ denoting the states $\{01, 10, 11\}$. Then the newly reached state set in this iteration is

$$\langle T^3 + 1, T^2 + (\alpha + 1)T + \alpha \rangle = \langle T^2 + (\alpha + 1)T + \alpha \rangle$$

We add these states to formerly reached states

$$\begin{aligned} reach &= \langle T \rangle \cdot \langle T^2 + (\alpha + 1)T + \alpha \rangle \\ &= \langle T \cdot T^2 + (\alpha + 1)T + \alpha \rangle \\ &= \langle T^3 + (\alpha + 1)T^2 + \alpha T \rangle \end{aligned}$$

i.e. states $\{00, 01, 10\}$. We update the present states for next iteration

$$from^1 = S^2 + (\alpha + 1)S + \alpha.$$

In the second iteration, we compute the reduced GB with the same term order for ideal $J = J_{ckt} + J_v + \langle from^1 \rangle$*. It includes a polynomial generator*

$$to^2 = T^2 + \alpha T$$

denotes states $\{00, 10\}$*. The complement of* $reached$ *is*

$$\langle T^4 - T \rangle : \langle T^3 + (\alpha + 1)T^2 + \alpha T \rangle = \langle T + 1 + \alpha \rangle$$

(i.e. states $\{11\}$*). We compute the newly reached state*

$$\langle T^2 + \alpha T, T + 1 + \alpha \rangle = \langle 1 \rangle$$

Since the GB contains the unit ideal, it means the newly reached state set is empty, thus a fixpoint has been reached. The algorithm terminates and returns

$$reached = \langle T^3 + (\alpha + 1)T^2 + \alpha T \rangle$$

which, as a Gröbner basis of the elimination ideal, canonically encodes the final reachable state set.

5.2.5 Significance of using Algebraic Geometry and Gröbner Basis

The essences of out approach consist of utilizations of algebraic geometry and Gröbner basis concepts. One the one hand, we use GB with elimination term ordering as an analog of image function. As mentioned in Section 4.3, we construct the elimination ideal $J + J_0$ to describe the circuit in Figure 5.5 using algebraic geometry. When given the present states, the next states are implicitly represented in the variety of the elimination ideal. Imposing elimination term order and computing GB, is actually projecting the variety on NS output T by eliminating all other variables. This projection gives us the canonical representation of NS in a polynomial with T.

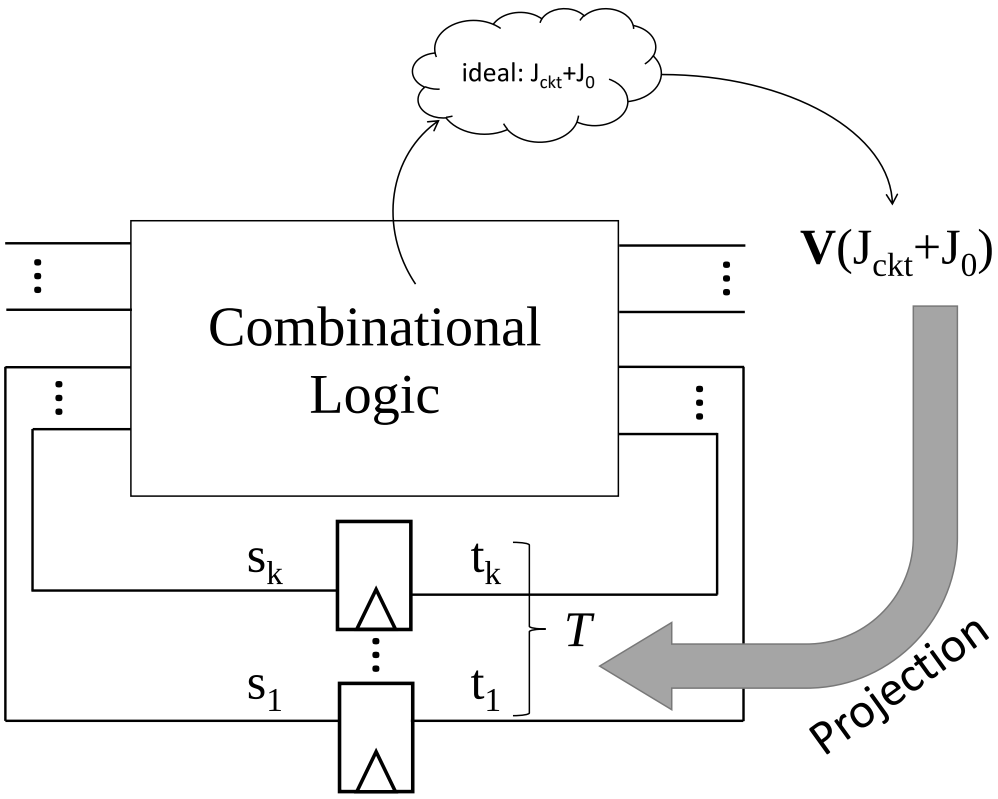


Figure 5.5: Projection of the variety of circuit description ideal

On the other hand, we use ideal arithmetic to implement set operations in Boolean domain. In algebraic geometry, manipulating ideals provides a high-level solution to operate on their varieties without actually solving the system. The intersection, union and complement of varieties are mapped to varieties of the sum, product and quotient of ideals, respectively. Furthermore, the varieties are equivalent to set of states because of correspondence of affine space and state space.

As a result, we find an analog of the FSM traversal algorithm in algebraic geometry which is compatible with word-level variables (e.g. S, T). We show it is effective to perform the reachability analysis.

5.3 Improving our Approach

Using elimination term ordering on computing GB for large set of polynomial is time-consuming, and usually intractable. The reason is the exponential computational complexity [23]:

Theorem 5.3 *Let $J + J_0 = \langle f_1, \dots, f_s,\ x_1^q - x_1, \dots, x_d^q - x_d \rangle \subset \mathbb{F}_q[x_1, \dots, x_d]$ be an ideal. The time and space complexity of Buchberger's algorithm to compute a Gröbner basis of $J + J_0$ is bounded by $q^{O(d)}$.*

In our case $q = 2^k$, and when k and d are large, this complexity makes abstraction infeasible.

Buchberger's algorithm consists of two major operations: one is finding S-poly pairs, the other is dividing the S-poly by the set of polynomials. Its actual cost is very sensitive to the term ordering: if there exists a term order making most S-poly computation unnecessary, then the cost to compute S-poly is saved. Moreover, it prevents new polynomials generating from unnecessary S-poly reductions, which further reduces the cost because there are less S-poly pairs as well as candidate divisors. Besides tuning the term ordering, directly cutting down the number of polynomials in the set is also effective.

In order to make our approach scalable, we propose improvements from two aspects: on the one hand, using another term ordering which can lower down the computational complexity of obtaining GB; on the other hand, reducing the polynomials by collecting bit-level primary inputs (PIs) and integrating them as word-level variables which are compatible with our working GF.

5.3.1 Simplifying the Gröbner Basis Computation

In Algorithm 7, a Gröbner basis is computed for each iteration to attain the word-level polynomial representation of the next states. In practice, for a sequential circuit with complicated structure and large size, Gröbner basis computation is intractable. To overcome the high computational complexity of computing a GB, we describe a method that computes a GB of a smaller subset of polynomials. The approach draws inspirations from [19]. According to proposition 2 in [19], if the GB is computed using *refined abstraction term order* (RATO), there will be only one pair of polynomials $\{f_w, f_g\}$ such that their leading monomials are not relatively prime, i.e.

$$gcd(LM(f_w), LM(f_g)) \neq 1$$

As a well-known fact from Buchberger's algorithm, the S-polynomial (Spoly) pairs with relatively prime leading monomials will always reduce to 0 modulo the basis and have no contribution to the Gröbner basis computation. Therefore, by removing the relatively prime polynomials from J_{ckt}, the Gröbner basis computation is transformed to the reduction of $Spoly(f_w, f_g)$ modulo J_{ckt}. More specifically, we turn the GB computation into one-step multivariate polynomial division, and the obtained remainder r will only contain bit-level inputs and word-level output.

Example 5.6 *After adding intermediate bit-level signal a, b, c, d, the elimination ideal for example circuit (Example 5.5) can be rewritten LEX order with RATO:*

$$(t_0, t_1) > (a, b, c, d)$$
$$> T > (x, s_0, s_1)$$

We can write down all polynomial generators of J_{ckt}:

$$f_1 : a + xs_0s_1 + xs_0 + xs_1 + x + s_0s_1 + s_0 + s_1 + 1$$

$$\begin{array}{ll} f_2 : b + s_0s_1 & f_3 : c + x + xs_0 \\ f_4 : d + s_0s_1 + s_0 & f_5 : t_0 + ab + a + 1 \\ f_6 : t_1 + cd + c + d & f_7 : t_0 + t_1\alpha + T \end{array}$$

From observation, the only pair which is not relatively prime is (f_5, f_7), thus the critical candidate polynomial pair is (f_w, f_g), where

$$f_w = t_0 + a \cdot b + a + b, \ \ f_g = t_0 + t_1\alpha + T$$

Result after reduction is:

$$Spoly(f_w, f_g) \xrightarrow{J+J_0}_+ T + s_0s_1x + \alpha s_0s_1$$
$$+ (1 + \alpha)s_0x + (1 + \alpha)s_0 + s_1x + s_1 + (1 + \alpha)x + 1$$

The remainder contains only bit-level inputs (x, s_0, s_1) and word-level output T.

The remainder from *Spoly* reduction contains bit-level PS variables, and our objective is to get a polynomial containing only word-level PS variables. One possible method is to rewrite bit-level variables in term of word-level variables, i.e.

$$s_i = \mathcal{G}(S) \tag{5.5}$$

Then we can substitute all bit-level variables with the word-level variable and obtain a word-level expression. The authors of [19] propose a method to construct a system of equations and solution to the system consists of Equation 5.5. This can be obtained by Gaussian elimination, which could compute corresponding $\mathcal{G}(S)$ efficiently with time complexity $O(k^3)$.

Example 5.7 Objective*: Abstract polynomial $s_i + \mathcal{G}_i(S)$ from $f_0 : s_0 + s_1\alpha + S$.*

First, compute $f_0^2 : s_0 + s_1\alpha^2 + S^2$. Apparently variable s_0 can be eliminated by operation

$$\begin{aligned} f_1 =& f_0 + f_0^2 \\ =& (\alpha^2 + \alpha)s_1 + S^2 + S \end{aligned}$$

Now we can solve univariate polynomial equation $f_1 = 0$ and get solution

$$s_1 = S^2 + S$$

Using this solution we can easily solve equation $f_0 = 0$. The result is

$$s_0 = \alpha S^2 + (1 + \alpha)S$$

More formally, polynomial expressions for s_i in terms of S can be obtained by setting up and solving the following system of equations:

$$\begin{bmatrix} S \\ S^2 \\ S^{2^2} \\ \vdots \\ S^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \cdots & \alpha^{k-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(k-1)} \\ 1 & \alpha^4 & \alpha^8 & \cdots & \alpha^{4(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{2^{k-1}} & \alpha^{2 \cdot 2^{k-1}} & \cdots & \alpha^{(k-1)\cdot 2^{k-1}} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{k-1} \end{bmatrix} \tag{5.6}$$

Let $\vec{S}$ be a vector of k unknowns $(s_0, \ldots, s_{k-1})$, then Equation 5.6 can be solved by using Cramer's rule or Gaussian elimination. In other words, we can obtain $s_i = \mathcal{G}(S)$ by solving Equation 5.6 symbolically.

In this approach we get word-level variable representation for each bit-level PS variables. By substitution, a new polynomial in word-level PS/NS variables could be obtained.

After processing with RATO and bit-to-word conversions, we get a polynomial in the form of $f_T = T + \mathcal{F}(S, x)$ denoting the **transition function**. We include a polynomial in S to define the present states f_S, as well as the set of vanishing polynomials for primary inputs $J_0^{PI} = \langle x_1^2 - x_1, \ldots, x_d^2 - x_d \rangle$. Using elimination term order with $S > x_i > T$, we can compute a GB of the elimination ideal $\langle f_T, f_S \rangle + J_0^{PI}$. This GB contains a univariate polynomial denoting next states. The improved algorithm is depicted in Algorithm 8.

5.3.2 PI Partition

Using above techniques we can get a remainder polynomial with only word-level PS/NS variables. However in most cases, the number of bit-level PIs will be too large for the last-step Gröbner basis computation. Therefore it is necessary to convert bit-level PIs to word-level PI variables.

Assume we have k-bit datapath and n-bit PIs. In finite field, we need to carefully partition n PIs such that states of each partition can be covered by a univariate polynomial respectively.

Proposition 5.1 *Divide n-bit PIs into partitions $n_1, n_2, \ldots, n_s$ where each $n_i | k$. Then let n_1-bit, n_2-bit, ..., n_s-bit word-level variables represent their evaluations in $\mathbb{F}_{2^k}$ as $\mathbb{F}_{2^{n_i}} \subset \mathbb{F}_{2^k}$.*

Again, assume a partition $n_i \mid k$ and corresponding word-level variable is P. Then we can use polynomial $P^{2^{n_i}} - P$ to represent all signals at free-end PIs, according to following theorem about **composite field**:

Theorem 5.4 *Let $k = m \cdot n_i$, such that $\mathbb{F}_{2^k} = \mathbb{F}_{(2^{n_i})^m}$. Let α be primitive root of $\mathbb{F}_{2^k}$, β be primitive root of the ground field $\mathbb{F}_{2^{n_i}}$. Then*

$$\beta = \alpha^{\omega}, \ \ \textit{where} \ \ \omega = \frac{2^k - 1}{2^{n_i} - 1}$$

Algorithm 8: Refined Algebraic Geometry based FSM Traversal

```
Input: Input-output circuit characteristic polynomial ideal $J_{ckt}$, initial state
       polynomial $\mathcal{F}(S)$
Output: Final reachable states represented by polynomial $\mathcal{G}(T)$
  $from^0 = reached = \mathcal{F}(S)$;
  $f_T$ =Reduce($Spoly(f_w, f_g), J_{ckt}$);
  /* Compute S-poly for the critical pair, then reduce it
     with circuit ideal under RATO                                  */
  Eliminate bit-level variables in $f_T$;
  repeat
      $i \leftarrow i + 1$;
      $G \leftarrow$GB($\langle f_T, from^{i-1}\rangle + J_0^{PI}$);
      /* Compute Gröbner basis with elimination term
         order: $T$ smallest; $J_0^{PI}$ covers all possible
         inputs from PIs                                            */
      $to^i \leftarrow G \cup \mathbb{F}_{2^k}[T]$;
      /* There will be a univariate polynomial in $G$
         denoting next state in word-level variable $T$             */
      $\langle new^i\rangle \leftarrow \langle to^i\rangle + (\langle T^{2^k} - T\rangle : \langle reached\rangle)$;
      /* Use quotient of ideals to attain complement of
         reached states, then use sum of ideals to attain
         an intersection with next state                            */
      $\langle reached\rangle \leftarrow \langle reached\rangle \cdot \langle new^i\rangle$;
      /* Use product of ideals to attain a union of new
         reached states and formerly reached states                 */
      $from^i \leftarrow new^i(S \setminus T)$;
      /* Start a new iteration by replacing variable $T$ in
         new reached states with current state variable $S$
         */
  until $\langle new^i\rangle == \langle 1\rangle$;
  /* Loop until a fixpoint reached: newly reached state
     is empty                                                       */
  return $\langle reached\rangle$
```

Example 5.8 *In a sequential circuit, PS/NS inputs/outputs are 4-bit signals, which means we will use $\mathbb{F}_{2^4}$ as working field. PIs are partitioned to 2-bit vectors, which means the ground field is $\mathbb{F}_{2^2}$. In ground field we can represent all possible evaluations of this PI partition $\{p_0, p_1\}$ with*

$$P^4 + P, \text{ where } P = p_0 + p_1 \cdot \beta$$

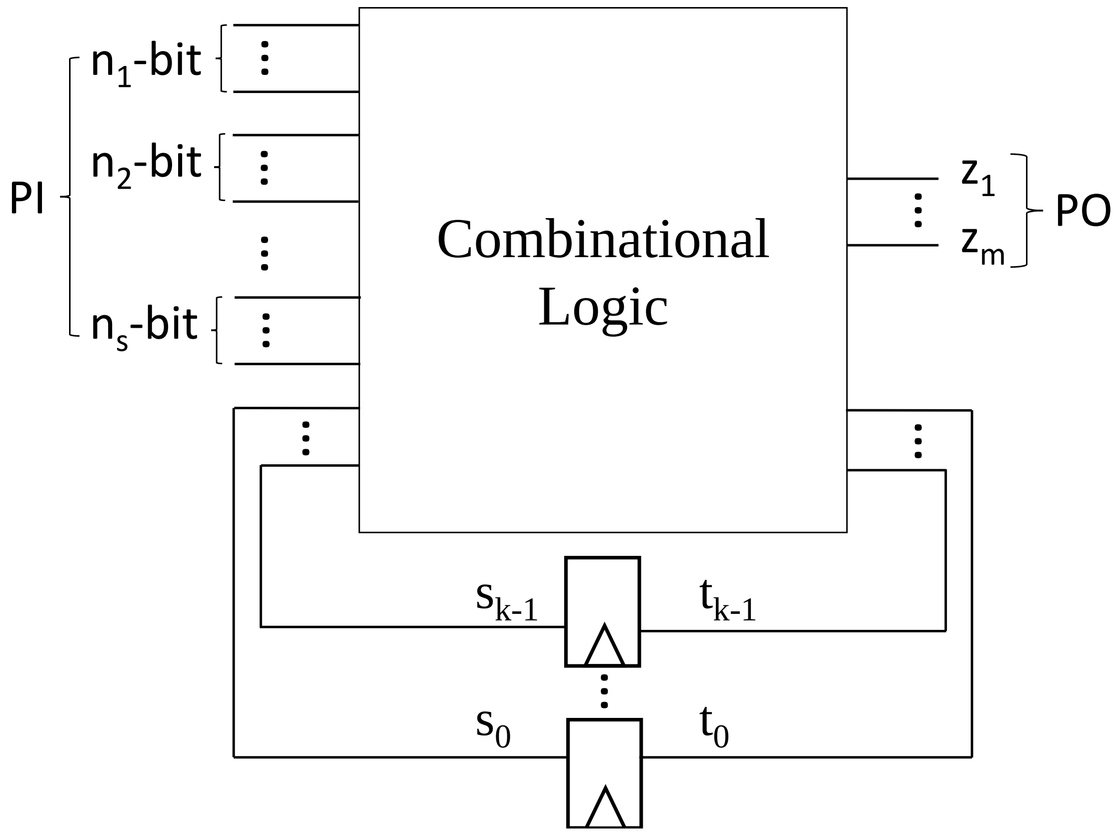


Figure 5.6: PI partition of a sequential circuit

Using Theorem 5.4 we get $\beta = \alpha^5$, so we can redefine word P as element from $\mathbb{F}_{2^4}$:

$$P = p_0 + p_1 \cdot \alpha^5$$

Using this method we can efficiently partition large size PIs to small number of word-level PI variables. One limitation of this approach is PIs cannot be partitioned when k is prime.

5.4 Implementation of Word-level FSM Traversal Algorithm

In this section, we described the architecture of our tool which can perform word-level FSM traversal on FSM benchmark circuits. Briefly speaking, our tool consists of 3 functional components. First, the gate-level netlist of circuit is translated to polynomial form and variables are sorted in RATO. This part is implemented using scripting language such as *Perl*. If the given benchmark is a structural/behavior hybrid description, we perform a pre-process on it to get a synthesized netlist. Secondly, the polynomial reduction is executed using our customized reduction engine, which is written in C++. Finally, we utilize symbolic computation engine – SINGULAR [33] to code Algorithm 8 and execute the BFS traversal. The tool outputs a univariate polynomial for NS word T, denoting the set of reachable states from given initial state.

Figure 5.7 illustrates the execution of reachability analysis approach based on C++ and SINGULAR implementation.

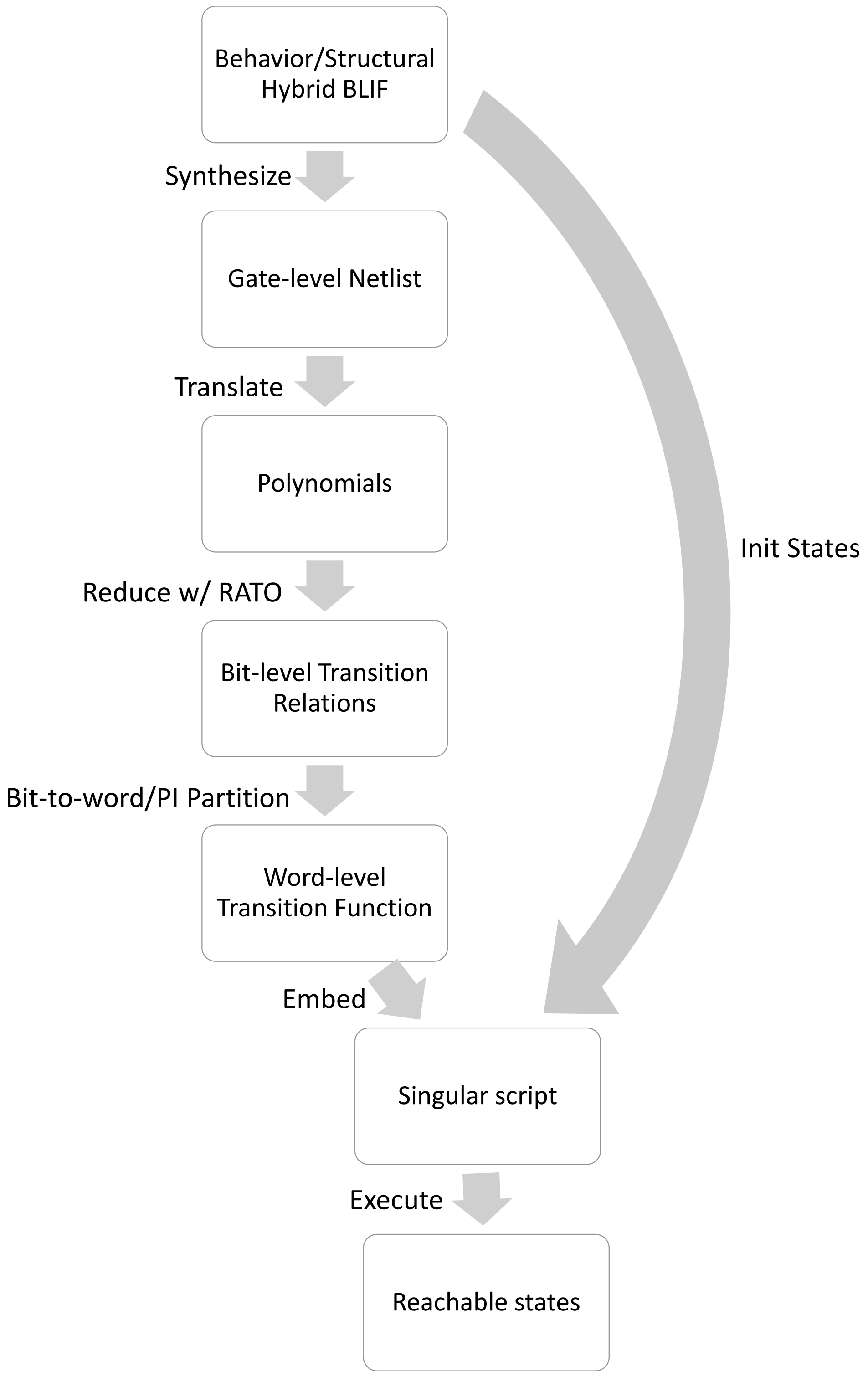


Figure 5.7: Execution process of word-level FSM traversal tool

Usually FSM designs can be described in behavior/structural hybrid languages. One of these languages is Berkeley logic interchange format (BLIF) [34], it allows state behavior representation and logic component representation.

Example 5.9 *In this example, we use benchmark FSM "lion9" from MCNC benchmark library. This benchmark circuit is given as truth-table based structural BLIF representation of a FSM.*

In order to compose the polynomial set in elimination ideal, we need to synthesize it to gate-level netlist. Modern synthesizers including ABC [35] and SIS [36] can complete this work. In this example we use a synthesis library containing only 2-input AND, NAND, OR, NOR, XOR, XNOR gates as well as the inverter, the output synthesized FSM is also given in BLIF format.

Using our interpreter, the synthesized BLIF file turns to a polynomial file customized for our polynomial reduction engine. The input file format includes all variables in RATO, along with the S-poly need to be reduced and polynomials in J_{ckt}.

The result given by our reduction engine is in the format

$$T + \mathcal{F}(s_0, \ldots, s_{k-1}, x_0, \ldots, x_{n-1})$$

where s_i and x_j denotes bit-level PS variables and PIs. Concretely in this example, the result is

$$\begin{aligned} T &+ (\alpha^2+\alpha+1)x_0x_1s_1s_3 + (\alpha^3+\alpha)x_0x_1s_1 + (\alpha+1)x_0x_1s_3 \\ &+ (\alpha^3+1)x_0s_1s_3 + (\alpha^3+\alpha)x_0s_1 + (\alpha+1)x_0s_3 + (\alpha^2)x_0 \\ &+ (\alpha^2+\alpha+1)x_1s_1s_3 + (\alpha^3+\alpha)x_1s_1 + (\alpha^2+\alpha+1)x_1s_3 \\ &+ (\alpha)x_1 + (\alpha^3+1)s_1s_3 + (\alpha^3+\alpha)s_1 + (\alpha^3+1)s_3 + \alpha^2 \end{aligned}$$

This is the transition function of this FSM. We utilized SINGULAR *to integrat both bit-word substitution and tranversal algorithm. In Figure 5.8, "tran" is the transition function we just obtained. "init_S" is the initial state, note it equals to "0100" register reset values preloaded in the BLIF file. Moreover, 2 bits PIs x_0, x_1 are combined to a word-level PI variable P using our conclusion in Example 5.8: "def_X" is the definition of 2-bit word, and "red_X" denotes the vanishing polynomial for word P.*

After the script is executed, the trversal finishes after 4 transition iterations, which denotes BFS depth equals to 4. The final reachable states is a degree-9 polynomial about T*, indicating final reachable states set contains 9 states. And state encodings can be obtained by solving this polynomial equation.*

```
<"FSM.lib";
// ring var: just all bit-level inputs (PS and PI) followed by S and T
ring rr = (2,X), (n23,n43,n42,n18,n40,n39,n38,n8,n31,n13,n36,n35,n34,n30,n33,v6_4,n28,n27,
n26,n25,n24,n23_1,n22,n21,n20,n19,n18_1,n17,n16,v0,v1,v2,v3,v4,v5,P,S,T), lp;
minpoly = X^4+X+1;

ideal A_in = v2,v3,v4,v5;
poly def_S = v2+ v3*X+ v4*X^2+ v5*X^3+S;
ideal X_in = v0,v1;
poly def_X = v0 + v1*X^5+P;
poly  red_S  =  S^16+S;
poly red_T = T^16+T;
poly red_X = P^4 + P;
// red_all: all bit-level vars and red_S
ideal red_all = v0^2+v0, v1^2+v1, v2^2+v2, v3^2+v3, v4^2+v4, v5^2+v5,red_S,red_X;
poly tran = T+(X^2+X+1)*v0*v1*v3*v5+(X^3+X)*v0*v1*v3+(X+1)*v0*v1*v5
+(X^3+1)*v0*v3*v5+(X^3+X)*v0*v3+(X+1)*v0*v5+(X^2)*v0+(X^2+X+1)*v1*v3*v5+(X^3+X)*v1*v3+(X^2+X+1)*v1*
v5+(X)*v1+(X^3+1)*v3*v5+(X^3+X)*v3+(X^3+1)*v5+X^2;

poly init_S = S+X^2;
poly reached = T+X^2;

ideal I1 = preprocess(def_S, red_all, A_in);
poly unitran = conv_word(tran,I1);
I1 = preprocess(def_X, red_all, X_in);
unitran = conv_word(unitran,I1);

int i = 1;
ideal from_I,to_I,new_I;
from_I[1] = init_S;
while(1)
{
      i++;
      to_I[i] = transition(from_I[i-1],unitran,red_all);
      "Iteration #",i-2;
      "Next State(s): ",to_I[i];
      new_I[i] = redWord(to_I[i]+compl(reached,red_T), red_T);
      "Newly reached states: ",new_I[i];
      if ((redWord(new_I[i],red_T) == 1) or (i>25))
      {
            "*************** TERMINATE! ***************";
            break;
      }
      reached = redWord(reached * new_I[i],red_T);
      "Currently reached states: ",reached;
      from_I[i] = subst(new_I[i],T,S);
}
"BFS depth: ",i-2;
"Final reachable states: ",reached;
```

Figure 5.8: Singular script for executing bit-to-word substitution and traversal loop

```
Iteration # 0
Next State(s):  T^4+(X^3+X^2+X+1)*T^2+(X^2+1)*T
Newly reached states:  T^3+(X^2)*T^2+(X^3+X^2)*T
Currently reached states:
T^4+(X^3+X^2+X+1)*T^2+(X^2+1)*T
Iteration # 1
Next State(s):
T^5+(X^3+X^2+X)*T^4+(X^3+X^2+X+1)*T^3+(X+1)*T
Newly reached states:  T+(X^3+X^2+X)
Currently reached states:
T^5+(X^3+X^2+X)*T^4+(X^3+X^2+X+1)*T^3+(X+1)*T
Iteration # 2
Next State(s):
T^4+(X^3+X^2+X)*T^3+(X^2+X)*T^2+(X^2)*T+(X)
Newly reached states:  T^2+(X^2+X)*T+1
Currently reached states:
T^7+(X^3)*T^6+(X^3+X^2)*T^5+(X^3+X)*T^4+(X^3+X^2)*T^3
+(X^3+X)*T^2+(X+1)*T
Iteration # 3
Next State(s):
T^6+(X^3+X+1)*T^5+(X^2+X+1)*T^4+(X^2+X)*T^3+(X^3+1)*T
^2+(X)*T+(X^2)
Newly reached states:  T^3+T^2+(X^2+1)*T+(X^3)
Currently reached states:
T^9+(X^3+X^2+1)*T^8+(X+1)*T^5+(X^2)*T^4+(X^3+X^2)*T^3
+(X^3)*T^2+(X^2+X)*T
Iteration # 4
Next State(s):
T^8+(X+1)*T^7+T^6+(X^3+X^2+X)*T^5+(X^3)*T^4+(X^3+X^2+
1)*T^3+(X^2+X)*T^2+(X^3+X)*T
Newly reached states:  1
*************** TERMINATE! ***************
BFS depth:  4
Final reachable states:
T^9+(X^3+X^2+1)*T^8+(X+1)*T^5+(X^2)*T^4+(X^3+X^2)*T^3
+(X^3)*T^2+(X^2+X)*T
```

Figure 5.9: The output given by our traversal tool

5.5 Experiment Results

We have implemented our traversal algorithm in 3 parts: the first part implements polynomial reductions (division) of the Gröbner basis computations, under the term order derived from the circuit as Line 2 in Algorithm 8. This is implemented with our customized data structure in C++. The second part implements the bit-level to word-level abstraction to attain transition functions at the word-level using the SINGULAR symbolic algebra computation system [v. 3-1-6] [33], as Line 3 in Algorithm 8; and the third part executes the reachability checking iterations using SINGULAR as well. With our tool implementation, we have performed experiments to analyze reachability of several FSMs. Our experiments run on a desktop with 3.5GHz Intel Core^{TM} i7-4770K Quad-core CPU, 32 GB RAM and 64-bit Ubuntu Linux OS. The experiments are shown in Table 5.2.

There are 2 bottlenecks which restricts the performance of our tool: one bottleneck is that the polynomial reduction engine is slow when the number of gates (especially OR gates) is large; the other one is the high computational complexity of Gröbner basis engine in general. Therefore, we pick 10 FSM benchmarks of reasonable size for testing our tool. Among them “b01, b02, b06” come from ITC’99 benchmarks, “s27, s208, s386” are from ISCAS’89 benchmarks and “bbara, beecount, dk14, donfile” are from MCNC benchmarks. ISCAS benchmarks are given as *bench* format so we can directly read gate information, where ITC/MCNC FSMs are given in unsynthesized *blif* format so we first turn them into gate-level netlists using AIG based synthesizer ABC. Since the number of primary inputs (m) is relatively small, in our experiments we partition primary inputs as m single bit-level variables. To verify the correctness of our techniques and implementations, we compare the number of reachable states obtained from our tool against the results obtained from the VIS tool [37].

Table 5.2: Results of running benchmarks using our tool. Parts I to III denote the time taken by polynomial divisions, bit-level to word-level abstraction and iterative reachability convergence checking part of our approach, respectively.

Benchmark	# Gates	# Latches	# PIs	# States	# iterations	Runtime (sec) I	Runtime (sec) II	Runtime (sec) III	Runtime of VIS (sec)
b01	39	5	2	18	5	< 0.01	0.01	0.02	< 0.01
b02	24	4	1	8	5	< 0.01	0.01	< 0.01	< 0.01
b06	49	9	2	13	4	< 0.01	0.07	5.0	< 0.01
s27	10	3	4	6	2	< 0.01	0.01	0.02	< 0.01
s208	61	8	11	16	16	< 0.01	0.32	2.4	< 0.01
s386	118	6	13	13	3	1.0	7.6	8.2	< 0.01
bbara	82	4	4	10	6	0.04	0.01	0.04	< 0.01
beecount	48	3	3	7	3	< 0.01	0.01	0.01	< 0.01
dk14	120	3	3	7	2	45	< 0.01	0.08	< 0.01
donfile	205	5	2	24	3	12316	0.02	1.7	< 0.01

In Table 5.2, # States denotes the final reachable states starting from given reset state, which given by our tool is the same with the return value of *compute_reach* in VIS. Meanwhile, from observation of the experiment run-times, we find the reduction runtime increases as the number of gates grows. Also, iterative reachability convergence check's runtime reflects both the size of present state/next state words (k) and the number of final reached states, which corresponds to the degree of polynomial *reached* in Algorithm 7. Although the efficiency of our initial implementation fails to compete with the BDD based FSM analyzer VIS, the experiment demonstrates the power of abstraction of algebraic geometry techniques for reachability analysis applications. Currently, we are investigating techniques that can help us overcome the complexity of the GB computation with elimination orders and speed-up our approach.

5.6 Conclusion

This dissertation has presented a new approach to perform reachability analysis of finite state machines at the word-level. This is achieved by modeling the transition relations and sets of states by way of polynomials over finite fields $\mathbb{F}_{2^k}$, where k represents the size of the state register bits. Subsequently using the concepts of elimination ideals, Gröbner bases, and quotients of ideals, we show that the set of reachable states can be encoded, canonically, as the variety of a univariate polynomial. This polynomial is computed using the Gröbner basis algorithm w.r.t. an elimination term order. Experiments are conducted with a few FSMs that validate the concept of word-level FSM traversal using algebraic geometry.

CHAPTER 6

FUNCTIONAL VERIFICATION OF SEQUENTIAL NORMAL BASIS MULTIPLIER

In order to utilize our traversal algorithm, it is necessary to find out a sort of suitable circuit benchmarks which is easy to compute its Gröbner basis (GB). From the work of *Lv* [9], we learn that arithmetic circuits in Galois field (GF) is convertible to an ideal of circuit polynomials, and the ideal generators form a GB themselves when applying reverse topological term order. Furthermore, according to the work of *Pruss et al.* [19], with a limited computation complexity, we can abstract the word-level signature of an arithmetic component working in GF. Thus, we consider the possibility of applying our traversal algorithm on sequential Galois field circuits. In each frame, we can use the techniques from [19] to abstract the word-level signature of the combinational logic, which corresponds to the transition function in our traversal algorithm. As a result, we manage to find a type of sequential GF multiplier which we can apply our traversal algorithm to actually verify its functional correctness.

6.1 Motivation

From the preliminaries about FSMs in Section 5.2, we learn that the Moore machine does not rely on inputs for state transitions. As depicted in Figure 6.1(a), a typical Moore machine implementation consists of combinational logic component and register files, where $r_0, \dots, r_k$ are present state (PS) variables standing for state inputs (SI), and $r'_0, \dots, r'_k$ are next state (NS) variables standing for state outputs (SO). Figure 6.1(b) shows the state transition graph (STG) of a Moore machine with $k + 1$ distinct states. We notice that it forms a simple chain, with k consecutive transitions the machine reaches final state R_k.

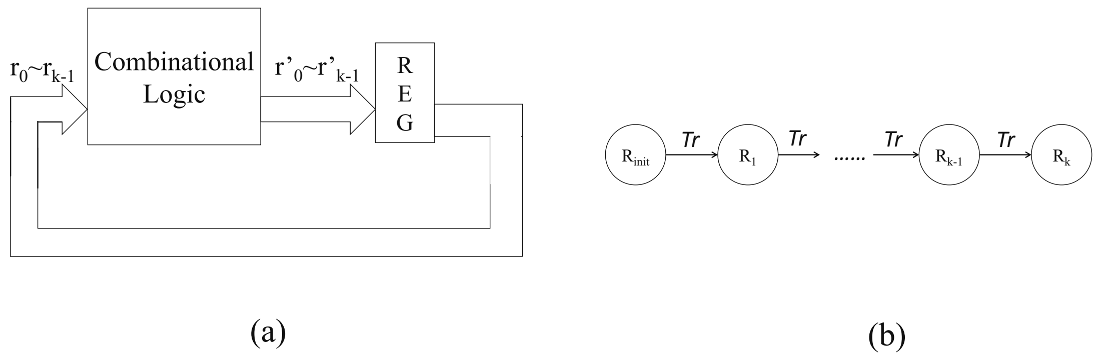


Figure 6.1: A typical Moore machine and its state transition graph

In practice, some arithmetic components are designed in sequential circuits similar to the structure in Figure 6.1(a). Initially the operands are loaded into the registers, then the closed circuit executes without taking any additional information from outside, and store the results in registers after k clock cycles. Its behavior can be described using STG in Figure 6.1(b): state R denotes the bits stored in registers. Concretely, R_{init} is the initial state (usually reset to all zeros), R_1 to R_{k-1} are intermediate results stored as SO of current state and SI for next state, and R_k (or R_{final}) is the final result given by arithmetic circuits (and equals to the answer to arithmetic function when circuit is working functional correctly). This kind of design results in reusing a smaller combinational logic component such that the area cost is greatly optimized. An example of these designs is the sequential GF arithmetic multipliers described in Chapter 3. However, it also brings difficulties in verifying the the circuit functions.

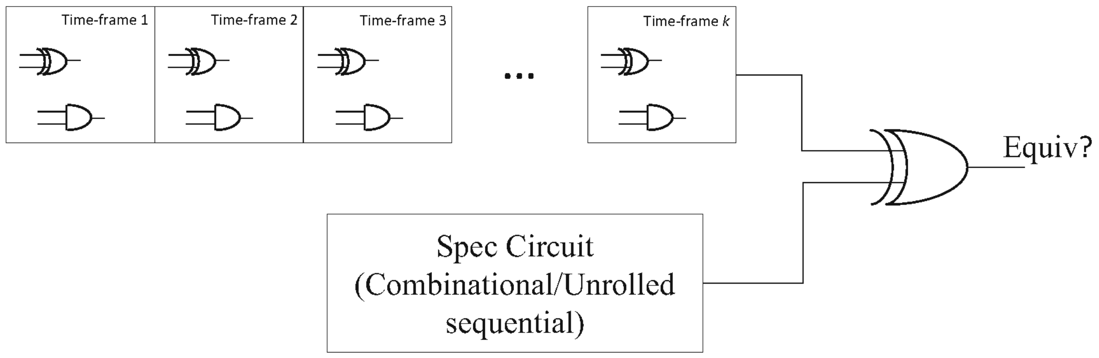


Figure 6.2: Conventional verification techniques based on bit-level unrolling and equivalence checking

Conventional methods to such a sequential circuit may consist of unrolling the circuit for k time-frames, and performing an equivalence checking between the unrolled machine and the specification function. However, the number of gates will grow fast when doing unrolling on bit-level. Meanwhile the structural similarity based equivalence checking techniques will fail when the sequential circuit is highly customized and optimized from the naive specification function. As a result, conventional techniques is grossly inefficient for large circuits. Therefore, a new method based on our proposed word-level FSM traversal technique is worthy to be explored.

6.2 Formal Verification of Normal Basis Multiplier using Gröbner Basis

The gate-level design of a NB multiplier can be generated using appoaches introduced in Section 3.3. The gate-level netlist is ready to be verified using an approach similar to that in Chapter 5. First we introduce the sketch of our approach using abstraction term order (ATO) mentioned previously in Section 4.3, then refine our approach using the concept "RATO", which is previously used in Section 5.3.

6.2.1 Implicit Unrolling based on Abstraction with ATO

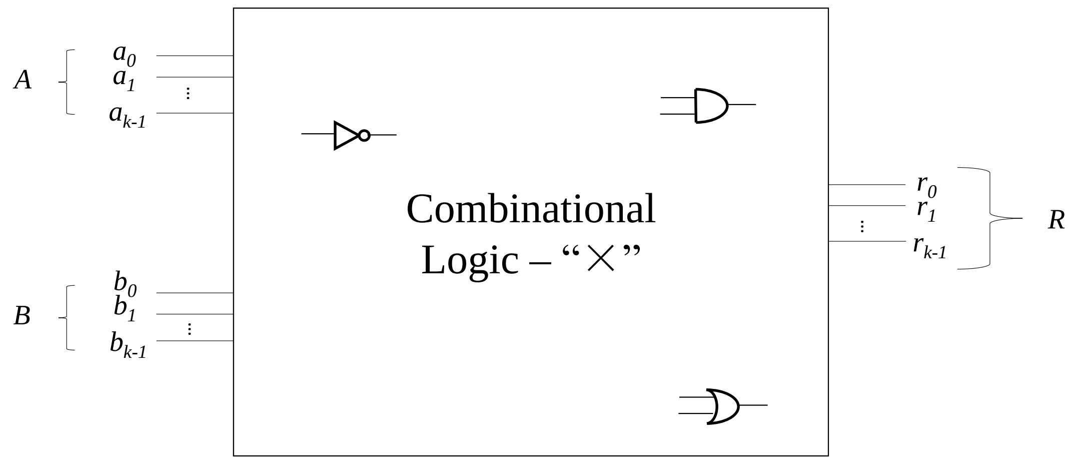


Figure 6.3: Architecture of a combinational GF multiplier

The concept of abstraction was discussed in Section 4.3. If we use an elimination term order with $intermediate\ variables > R > A, B$ for the circuit in Figure 6.3 to compute Gröbner basis, the function of the combinational logic component can be abstracted as

$$R = \mathcal{F}(A, B)$$

To verify the functional correctness of a combinational NB multiplier (e.g. Mastrovito multiplier or Montgomery multiplier), the function given by abstraction will be computed as

$$R = A \cdot B$$

While in the sequential case, the function of combinational logic only fulfills a part of the multiplication. For example, in the RH-SMPO introduced in Section 3.3, the combinational logic actually implements F_{k-1} in Equation 3.16, while computing all of F_i requires k-cycle unrolling of the circuit. Nevertheless, the abstraction still provides a word-level representation which can be more efficient for unrolling than bit-level expressions. In other words, with the assistance of abstraction, we can execute implicit unrolling instead of explicit unrolling and avoid bit-blasting problem.

For 2-input sequential NB multipliers, abstraction is utilized to implement following algorithm:

Algorithm 9: Abstraction via implicit unrolling for Sequential GF circuit verification

```
Input: Circuit polynomial ideal $J$, vanishing ideal $J_0$, initial state ideal for output
       $R(=0)$, inputs $\mathcal{G}(A_{init}), \mathcal{H}(B_{init})$
Output: Circuit function $R_{final}$ after $k$ cycles of $A_{init}, B_{init}$
1 $from_0(R, A, B) = \langle R, \mathcal{G}(A_{init}), \mathcal{H}(B_{init})\rangle$;
  /* $A, B$ are word-level variables, solutions to
     polynomial equations $\mathcal{G}, \mathcal{H}$ denote the initial values
     of preloaded operands $A_{init}, B_{init}$ */
2 $i = 0$;
3 repeat
4     $i \leftarrow i + 1$;
5     $G \leftarrow$GB($\langle J + J_0 + from_{i-1}(R, A, B)\rangle$) with ATO;
6     $to_i(R', A', B') \leftarrow G \cap \mathbb{F}_{2^k}[R', A', B', R, A, B]$;
      /* Using projections of varieties from abstraction
         theory, we get NS variables $R', A', B'$ in terms of PS
         $A, B$ */
7     $from_i \leftarrow to_i(\{R, A, B\} \setminus \{R', A', B'\})$;
      /* By replacing NS variables with PS variables we
         push it to next time-frame */
8 until $i == k$;
9 return $from_k(R_{final})$
```

Notations in the following figure as well as in next example comply with the notations in this algorithm.

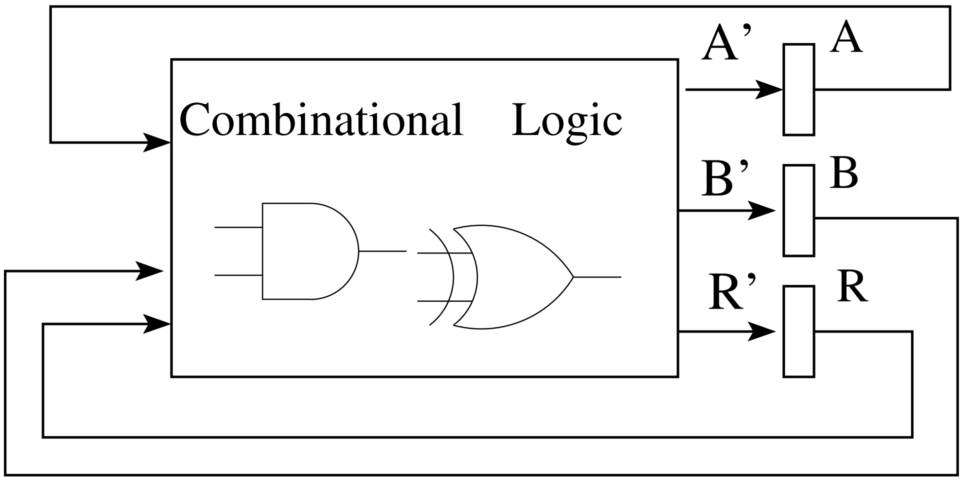


Figure 6.4: A typical normal basis GF sequential circuit model. $A = (a_0, \ldots, a_{k-1})$ and similarly B, R are k-bit registers; A', B', R' denote next-state inputs.

We follow the sequential GF circuit model of Figure 6.4, with word-level variables A, B, R denoting the *present states (PS)* and A', B', R' denoting the *next states (NS)* of the machine; where $A = \sum_{i=0}^{k-1} a_i \beta^{2^i}$ for the PS variables and $A' = \sum_{i=0}^{k-1} a'_i \beta^{2^i}$ for NS variables, and so on. Variables R (R') correspond to those that store the result, and A, B (A', B') store input operands. E.g., for a GF multiplier, A_{init}, B_{init} (and $R_{init} = 0$) are the initial values (operands) loaded into the registers, and $R = \mathcal{F}(A_{init}, B_{init}) = A_{init} \times B_{init}$ is the final result after k-cycles. Our approach aims to find this polynomial representation for R.

Each gate in the combinational logic is represented by a Boolean polynomial. To this set of Boolean polynomials, we append the polynomials that define the word-level to bit-level relations for PS and NS variables ($A = \sum_{i=0}^{k-1} a_i \beta^{2^i}$). We denote this set of polynomials as ideal $J = \langle f_1, \ldots, f_s \rangle \subset \mathbb{F}_{2^k}[x_1, \ldots, x_d, R, R', A, A', B, B']$, where $x_1, \ldots, x_d$ denote the bit-level (Boolean) variables of the circuit. The ideal of vanishing polynomials J_0 is also included, and then the implicit FSM unrolling problem is setup for abstraction.

The configurations of the flip-flops are the states of the machine. Since the set of states is a finite set of points, we can consider it as the variety of an ideal related to the circuit. Moreover, since we are interested in the function encoded by the state variables (over k-time frames), we can project this variety on the word-level state variables, starting from the initial state A_{init}, B_{init}. Projection of varieties (geometry) corresponds to elimination ideals (algebra), and can be analyzed via Gröbner bases. Therefore, we employ a Gröbner basis computation with ATO: we use a *lex term order* with *bit-level variables* $>$ *word-level NS outputs* $>$ *word-level PS inputs*. This allows to eliminate all the bit-level variables and derives a representation only in terms of words. Consequently, k-successive Gröbner basis computations implicitly unroll the machine, and provide word-level algebraic k-cycle abstraction for R' as $R' = \mathcal{F}(A_{init}, B_{init})$.

Algorithm 9 describes our approach. In the algorithm, $from_i$ and to_i are polynomial ideals whose varieties are the valuations of word-level variables R, A, B and R', A', B' in the i-th iteration; and the notation "\" signifies that the NS in iteration (i) becomes the PS in iteration $(i + 1)$. Line 5 computes the Gröbner basis with the abstraction term order. Line 6 computes the elimination ideal, eliminating the bit-level variables and representing the set of reachable states up to iteration i in terms of the elimination ideal. These computations are analogous to those of image computations performed in the FSM reachability algorithm (given in Chapter 5).

In the practice of sequential GF multiplier verification, the combinational logic actually implements function not only related to current operands A and B, but also involved with PS variable (i.e. temporary product) R. Which can be obtained using the abstraction:

$$R' = \mathcal{F}(A, B, R)$$

Using ATO, if we put R ahead of R', A, B in term ordering, R is thus eliminated and the result in single iteration will be $R' = \mathcal{F}(A, B)$. NS operands A' and B' are right-cyclic shift of A and B, which can be directly written. If initial values A_{init}, B_{init} are treated as parameters, the NS ideal contains polynomials with A', B' and $R' = \mathcal{F}(A_{init}, B_{init})$. This is also shown in the following example.

Example 6.1 (Functional verification of 5-bit RH-SMPO) *Figure 3.3 shows the detailed structure of a 5-bit RH-SMPO. The transition function for operands A, B is doing cyclic shift, while transition function for R has to be computed through Gröbner basis* *abstraction approach. Following ideal J_{ckt} from line 5 in Algorithm 9 is the ideal for all gates in combinational logic block and definition of word-level variables.*

$$J_{ckt} = \langle f_1, f_2, \dots, f_{19} \rangle$$

$$\left.\begin{aligned} d_0 + a_4b_4, c_1 + a_0 + a_4, c_2 + b_0 + b_4, d_1 + c_1c_2, c_3 + a_1a_4 \\ c_4 + b_1b_4, d_2 + c_3c_4, e_0 + d_0 + d_1, e_3 + d_1 + d_2, e_4 + d_2 \\ R_0 + r_4 + e_0, R_1 + r_0, R_2 + r_1, R_3 + r_2 + e_3, R_4 + r_3 + e_4 \end{aligned}\right\} \quad \{f_1 \dots f_{15}\}$$

$$\left.\begin{aligned} A + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18} \\ B + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18} \\ R' + r'_0\alpha^5 + r'_1\alpha^{10} + r'_2\alpha^{20} + r'_3\alpha^9 + r'_4\alpha^{18} \\ R + R_0\alpha^5 + R_1\alpha^{10} + R_2\alpha^{20} + R_3\alpha^9 + R_4\alpha^{18} \end{aligned}\right\} \quad \{f_{16} \dots f_{19}\}$$

In our implementation here, since we only focus on the output variable R, evaluations of intermediate input operands A, B are replaced by evaluations with parameters A_{init}, B_{init}. Thus polynomials describing A and B (f_{16} and f_{17}) can be removed from J_{ckt}, and R is directly evaluated by initial operands A_{init} and B_{init}. Those two parameters are associated with PS bit-level inputs $a_0, a_1, \dots, a_4$ and $b_0, b_1, \dots, b_4$ by polynomials given in $from^i$.

According to line 5 of Algorithm 9, we merge J_{ckt}, J_0 and $from^i$, then compute its Gröbner basis with abstraction term order. Concretely, in the first iteration $from^0$ contains three generators. The first one describes PS variable R – temporary (or initial, in first iteration) product, which equals to 0 according to the mechanism of sequential GF multiplier. The polynomial is written as

$$R$$

The other two polynomials describes PS variable A, B – current multiplication operands, and we write them in parameters A_{init}, B_{init}. Since this is the first iteration, they have the same form with f_{16} and f_{17}:

$$A_{init} + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}$$

$$B_{init} + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18}$$

After computing the Gröbner basis of $J_{ckt} + J_0 + from^0$ using ATO

$$\textit{all other bit-level variables} > R > \{R', A_{init}, B_{init}\}$$

there is a polynomial in form of $R' + \mathcal{F}(A_{init}, B_{init})$, which should be included by to^{i+1}. to^{i+1} also exclude next state variable A' and B', instead we redefine A_{init} and B_{init} using next state bit-level variables $\{a'_i, b'_j\}$. Next state Bit-level variables $a'_i = a_{i-1 \pmod k}, b'_j = b_{j-1 \pmod k}$ according to definition of cyclic shift.

Line 7 in Algorithm 9 is implemented by replacing R' with R, $\{a'_i, b'_j\}$ with $\{a_i, b_j\}$.

All intermediate results for each clock cycle are listed below:

- *Clock-cycle 1:* $\mathbf{from^0} = \{R, A_{init} + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}, B_{init} + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18}\}$,
  $\mathbf{to^1} = \{R' + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init} + (\alpha^4+\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{8} + (\alpha^2)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha)A_{init}^{2}B_{init} + (\alpha^4+\alpha^3+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^{4} + (\alpha^4+\alpha)A_{init}B_{init}^{2} + (\alpha^3+\alpha+1)A_{init}B_{init}, A_{init} + a'_4\alpha^5 + a'_0\alpha^{10} + a'_1\alpha^{20} + a'_2\alpha^9 + a'_3\alpha^{18}, B_{init} + b'_4\alpha^5 + b'_0\alpha^{10} + b'_1\alpha^{20} + b'_2\alpha^9 + b'_3\alpha^{18}\}$

- *Clock-cycle 2:* $\mathbf{from^1} = \{R + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init} + (\alpha^4+\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{8} + (\alpha^2)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha)A_{init}^{2}B_{init} + (\alpha^4+\alpha^3+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^{4} + (\alpha^4+\alpha)A_{init}B_{init}^{2} + (\alpha^3+\alpha+1)A_{init}B_{init}, A_{init} + a_4\alpha^5 + a_0\alpha^{10} + a_1\alpha^{20} + a_2\alpha^9 + a_3\alpha^{18}, B_{init} + b_4\alpha^5 + b_0\alpha^{10} + b_1\alpha^{20} + b_2\alpha^9 + b_3\alpha^{18}\}$,

$\mathbf{to^2} = \{R' + (\alpha^3+\alpha+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{8} + (\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{8}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init}^{8} + (\alpha^4)A_{init}^{8}B_{init}^{4} + (\alpha^4+\alpha^3+1)A_{init}^{8}B_{init}^{2} + (\alpha^3+1)A_{init}^{8}B_{init} + (\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4)A_{init}^{4}B_{init}^{8} + (\alpha^4)A_{init}^{4}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^4+\alpha^3+1)A_{init}^{2}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^2)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init} + (\alpha^3+1)A_{init}B_{init}^{8} + (\alpha)A_{init}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}, A_{init} + a_3'\alpha^5 + a_4'\alpha^{10} + a_0'\alpha^{20} + a_1'\alpha^9 + a_2'\alpha^{18}, B_{init} + b_3'\alpha^5 + b_4'\alpha^{10} + b_0'\alpha^{20} + b_1'\alpha^9 + b_2'\alpha^{18}\}$

- *Clock-cycle 3:* $\mathbf{from^2} = \{R + (\alpha^3+\alpha+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{8} + (\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{8}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init}^{8} + (\alpha^4)A_{init}^{8}B_{init}^{4} + (\alpha^4+\alpha^3+1)A_{init}^{8}B_{init}^{2} + (\alpha^3+1)A_{init}^{8}B_{init} + (\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4)A_{init}^{4}B_{init}^{8} + (\alpha^4)A_{init}^{4}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^4+\alpha^3+1)A_{init}^{2}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^2)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init} + (\alpha^3+1)A_{init}B_{init}^{8} + (\alpha)A_{init}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}, A_{init} + a_3\alpha^5 + a_4\alpha^{10} + a_0\alpha^{20} + a_1\alpha^9 + a_2\alpha^{18}, B_{init} + b_3\alpha^5 + b_4\alpha^{10} + b_0\alpha^{20} + b_1\alpha^9 + b_2\alpha^{18}\}$,
  $\mathbf{to^3} = \{R' + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha)A_{init}^{16}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init} + (\alpha)A_{init}^{8}B_{init}^{16} + (\alpha+1)A_{init}^{8}B_{init}^{8} + (\alpha^4)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{4}B_{init}^{16} + (\alpha^4)A_{init}^{4}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha^2+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}^{4} + (\alpha^4+\alpha)A_{init}B_{init}, A_{init} + a_2'\alpha^5 + a_3'\alpha^{10} + a_4'\alpha^{20} + a_0'\alpha^9 + a_1'\alpha^{18}, B_{init} + b_2'\alpha^5 + b_3'\alpha^{10} + b_4'\alpha^{20} + b_0'\alpha^9 + b_1'\alpha^{18}\}$

- *Clock-cycle 4:* $\mathbf{from^3} = \{R + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha)A_{init}^{16}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init} + (\alpha)A_{init}^{8}B_{init}^{16} + (\alpha+1)A_{init}^{8}B_{init}^{8} + (\alpha^4)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{4}B_{init}^{16} + (\alpha^4)A_{init}^{4}B_{init}^{8} +$

$(\alpha^4+\alpha^3+\alpha+1)A_{init}^4B_{init}^4 + (\alpha^3+\alpha+1)A_{init}^4B_{init}^2 + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^4B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^2B_{init}^{16} + (\alpha^3+\alpha^2+1)A_{init}^2B_{init}^8 + (\alpha^3+\alpha+1)A_{init}^2B_{init}^4 + (\alpha^3+\alpha+1)A_{init}^2B_{init}^2 + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^8 + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}^4 + (\alpha^4+\alpha)A_{init}B_{init}, A_{init} + a_2\alpha^5 + a_3\alpha^{10} + a_4\alpha^{20} + a_0\alpha^9 + a_1\alpha^{18}, B_{init} + b_2\alpha^5 + b_3\alpha^{10} + b_4\alpha^{20} + b_0\alpha^9 + b_1\alpha^{18}\}$,

$\mathbf{to^4} = \{R' + (\alpha^3+\alpha+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^8 + (\alpha^4+\alpha)A_{init}^{16}B_{init}^4 + (\alpha^3+1)A_{init}^{16}B_{init}^2 + (\alpha^3+\alpha+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^8B_{init}^{16} + (\alpha^3+1)A_{init}^8B_{init}^8 + (\alpha^4+\alpha^2+\alpha)A_{init}^8B_{init}^4 + (\alpha^2+\alpha)A_{init}^8B_{init}^2 + (\alpha^3+\alpha^2+1)A_{init}^8B_{init} + (\alpha^4+\alpha)A_{init}^4B_{init}^{16} + (\alpha^4+\alpha^2+\alpha)A_{init}^4B_{init}^8 + (\alpha^4+\alpha^2+\alpha)A_{init}^4B_{init}^4 + (\alpha^2+\alpha)A_{init}^4B_{init} + (\alpha^3+1)A_{init}^2B_{init}^{16} + (\alpha^2+\alpha)A_{init}^2B_{init}^8 + (\alpha^4+\alpha^2)A_{init}^2B_{init}^2 + (\alpha^3+\alpha^2+1)A_{init}^2B_{init} + (\alpha^3+\alpha+1)A_{init}B_{init}^{16} + (\alpha^3+\alpha^2+1)A_{init}B_{init}^8 + (\alpha^2+\alpha)A_{init}B_{init}^4 + (\alpha^3+\alpha^2+1)A_{init}B_{init}^2 + (\alpha)A_{init}B_{init}, A_{init} + a_1'\alpha^5 + a_2'\alpha^{10} + a_3'\alpha^{20} + a_4'\alpha^9 + a_0'\alpha^{18}, B_{init} + b_1'\alpha^5 + b_2'\alpha^{10} + b_3'\alpha^{20} + b_4'\alpha^9 + b_0'\alpha^{18}\}$

- *Clock-cycle 5:* $\mathbf{from^4} = \{R + (\alpha^3+\alpha+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^8 + (\alpha^4+\alpha)A_{init}^{16}B_{init}^4 + (\alpha^3+1)A_{init}^{16}B_{init}^2 + (\alpha^3+\alpha+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^8B_{init}^{16} + (\alpha^3+1)A_{init}^8B_{init}^8 + (\alpha^4+\alpha^2+\alpha)A_{init}^8B_{init}^4 + (\alpha^2+\alpha)A_{init}^8B_{init}^2 + (\alpha^3+\alpha^2+1)A_{init}^8B_{init} + (\alpha^4+\alpha)A_{init}^4B_{init}^{16} + (\alpha^4+\alpha^2+\alpha)A_{init}^4B_{init}^8 + (\alpha^4+\alpha^2+\alpha)A_{init}^4B_{init}^4 + (\alpha^2+\alpha)A_{init}^4B_{init} + (\alpha^3+1)A_{init}^2B_{init}^{16} + (\alpha^2+\alpha)A_{init}^2B_{init}^8 + (\alpha^4+\alpha^2)A_{init}^2B_{init}^2 + (\alpha^3+\alpha^2+1)A_{init}^2B_{init} + (\alpha^3+\alpha+1)A_{init}B_{init}^{16} + (\alpha^3+\alpha^2+1)A_{init}B_{init}^8 + (\alpha^2+\alpha)A_{init}B_{init}^4 + (\alpha^3+\alpha^2+1)A_{init}B_{init}^2 + (\alpha)A_{init}B_{init}, A_{init} + a_1\alpha^5 + a_2\alpha^{10} + a_3\alpha^{20} + a_4\alpha^9 + a_0\alpha^{18}, B_{init} + b_1\alpha^5 + b_2\alpha^{10} + b_3\alpha^{20} + b_4\alpha^9 + b_0\alpha^{18}\}$,

  $\mathbf{to^5} = \{\mathbf{R' + A_{init}B_{init}}, A_{init} + a_0'\alpha^5 + a_1'\alpha^{10} + a_2'\alpha^{20} + a_3'\alpha^9 + a_4'\alpha^{18}, B_{init} + b_0'\alpha^5 + b_1'\alpha^{10} + b_2'\alpha^{20} + b_3'\alpha^9 + b_4'\alpha^{18}\}$

The algorithm returns

$$from^5(R_{final}) = R_{final} + A_{init}B_{init}$$

which is the function of the multiplier: $R_{final} = A_{init} \cdot B_{init}$

6.2.2 Overcome Computational Complexity using RATO

We implement Algorithm 9 in SINGULAR, with the same setup and environment with the experiment in Section 5.5. The result on verifying Agnew's SMPO is shown in following table:

Table 6.1: (Dummy)Runtime of Gröbner Basis Computation of Mastrovito Multipliers in Singular using ATO $>$.

Word Size (k)	Number of Polynomials (d)	Computation Time (minutes)
16	$1,871$	2.4
24	$3,135$	12
32	$5,549$	22.6
40	$8,587$	266
48	$12,327$	NA (Out of Memory)

It indicates that our approach based on ATO cannot verify sequential GF multiplier with size larger than 40(dummy). Similar to our improvements in Section 5.3, RATO [38] is also available to accelerate the GB computation here. More specifically, we obviate the GB computation by turning the Buchberger's algorithm into a single-step multivariate polynomial division.

- First, a set of polynomials is constructed by polynomials translated from each logic gates, and they generate ideal J_{gates}. Similarly, ideal J_{W-B} is generated by polynomials representing the word-bit correspondences. They are merged as an ideal describing the combinational logic of the circuit: $J_{ckt} = J_{gates} + J_{W-B}$.
- Second, we impose RATO on ideal J_{ckt}: all bit-level variables are sorted using reverse topological order in circuit structure, followed by word-level NS output (e.g. R' in Figure 6.4), then word-level PS inputs (e.g. A, B, R in Figure 6.4).
- When computing the Gröbner basis of $J_{ckt} + J_0$ (adding vanishing polynomials), because of RATO, there only exist one pair of polynomials (f_w, f_g) such that $LCM(lm(f_w), lm(f_g)) \neq lm(f_w) \cdot lm(f_g)$.

- Then we only need to compute the S-poly for (f_w, f_g), the reduce the S-poly by $J_{ckt} + J_0$.

After executing the reduction, the remainder only contains a limited number of variables. It can be further transformed to a canonical polynomial function of the circuit. We illustrate the whole improved procedure by applying RATO on 5-bit RH-SMPO described in Example 6.1 and Figure 3.3.

Example 6.2 *From the circuit topological structure in Figure 3.3, the term order under RATO is:*

$$\begin{aligned}&\{r'_0, r'_1, r'_2, r'_3, r'_4\} > \{r_0, r_1, r_2, r_3, r_4\}\\&> \{e_0, e_3, e_4\}, \{d_0, d_1, d_2\}, \{c_1, c_2, c_3, c_4\}\\&> \{a_0, a_1, a_2, a_3, a_4, b_0, b_1, b_2, b_3, b_4\} > R' > R > \{A, B\}\end{aligned}$$

The variables in braces are at the same topological level, they are arranged in LEX order. We search among all generators of J_{ckt} from Example 6.1 using RATO, and find a pair of polynomials whose leading monomials are not relatively prime: $(f_w, f_g), f_w = r'_0 + r_4 + e_0, f_g = r'_0\alpha^5 + r'_1\alpha^{10} + r'_2\alpha^{20} + r'_3\alpha^9 + r'_4\alpha^{18} + R'$. We calculate Spoly can reduce it by $J_{ckt} + J_0$:

$$\begin{aligned}&Spoly(f_w, f_g) \xrightarrow{J_{ckt}+J_0}_+\\&(\alpha^3+\alpha^2+\alpha)r_1 + (\alpha^4+\alpha^3+\alpha^2)r_2 + (\alpha^2+\alpha)r_3 + (\alpha)r_4\\&+ (\alpha^3+\alpha^2)a_1b_1 + (\alpha^4+\alpha^3+\alpha^2+\alpha)a_1b_2 + (\alpha^2+\alpha)a_1b_3\\&+ (\alpha^2+1)a_1b_4 + (\alpha^4+1)a_1B + (\alpha^4+\alpha)a_2b_1 + (\alpha^4+\alpha^3+\alpha)a_2b_2\\&+ (\alpha^3+1)a_2b_3 + (\alpha^3+\alpha^2+1)a_2b_4 + (\alpha^3+\alpha^2)a_2B + (\alpha^2+\alpha)a_3b_1\\&+ (\alpha^3+1)a_3b_2 + (\alpha+1)a_3b_3 + (\alpha^4+\alpha^2+\alpha)a_3b_4\\&+ (\alpha^4+\alpha^3+\alpha)a_3B + (\alpha^3+1)a_4b_1 + a_4b_2 + (\alpha^4+\alpha^2+\alpha)a_4b_3\\&+ (\alpha^4+\alpha^3+1)a_4b_4 + (\alpha^2+\alpha)a_4B + (\alpha^4+1)b_1A + (\alpha^3+\alpha^2)b_2A\\&+ (\alpha^4+\alpha^3+\alpha)b_3A + (\alpha^2+\alpha)b_4A + (\alpha^4+\alpha^2+\alpha+1)R' + R + AB\end{aligned}$$

Above example indicates that RATO based abstraction on 5-bit RH-SMPO will result a remainder containing both bit-level variables and word-level variables. Moreover, the number of variables is still large such that Gröbner basis computation will be inefficient.

Since the remainder from *Spoly* reduction contains some bit-level variables, our objective is to compute a polynomial that contains only word-level variables (such as $R' + \mathcal{F}(A, B)$). One possible solution to this problem is to replace bit-level variables by equivalent polynomials that only contain word-level variables, e.g. $a_i = \mathcal{G}(A), r_j = \mathcal{H}(R)$. In this section a Gaussian-elimination approach is introduced to compute corresponding $\mathcal{G}(A), \mathcal{H}(R)$ efficiently.

Example 6.3 **Objective**: *Compute polynomial $a_i + \mathcal{G}_i(A)$ from $f_0 = a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18} + A = g_0 + A$.*

First, compute $f_0^2 = a_0\alpha^{10} + a_1\alpha^{20} + a_2\alpha^9 + a_3\alpha^{18} + a_4\alpha^5 + A^2 = g_0^2 + A^2$; then f_0^4, f_0^8, f_0^{16}. By repeating squaring we get a system of equations:

$$\begin{cases} f_0 &= 0 \\ f_0^2 &= 0 \\ f_0^4 &= 0 \\ f_0^8 &= 0 \\ f_0^{16} &= 0 \end{cases} \iff \begin{cases} g_0 &= A \\ g_0^2 &= A^2 \\ g_0^4 &= A^4 \\ g_0^8 &= A^8 \\ g_0^{16} &= A^{16} \end{cases}$$

Following is the coefficients matrix form of this system of equations:

$$\begin{pmatrix} \alpha^5 & \alpha^{10} & \alpha^{20} & \alpha^9 & \alpha^{18} \\ \alpha^{10} & \alpha^{20} & \alpha^9 & \alpha^{18} & \alpha^5 \\ \alpha^{20} & \alpha^9 & \alpha^{18} & \alpha^5 & \alpha^{10} \\ \alpha^9 & \alpha^{18} & \alpha^5 & \alpha^{10} & \alpha^{20} \\ \alpha^{18} & \alpha^5 & \alpha^{10} & \alpha^{20} & \alpha^9 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} A \\ A^2 \\ A^4 \\ A^8 \\ A^{16} \end{pmatrix}$$

Then we use Gaussian elimination on coefficients matrix to recursively eliminate a_1 from third row, a_2 from fourth row, etc. The final solution to this system of equations is

$$\begin{cases} a_0 &= (\alpha+1)A^{16} + (\alpha^4+\alpha^3+\alpha)A^8 + (\alpha^3+\alpha^2)A^4 \\ & \quad +(\alpha^4+1)A^2 + (\alpha^2+1)A \\ a_1 &= (\alpha^2+1)A^{16} + (\alpha+1)A^8 + (\alpha^4+\alpha^3+\alpha)A^4 \\ & \quad +(\alpha^3+\alpha^2)A^2 + (\alpha^4+1)A \\ a_2 &= (\alpha^4+1)A^{16} + (\alpha^2+1)A^8 + (\alpha+1)A^4 \\ & \quad +(\alpha^4+\alpha^3+\alpha)A^2 + (\alpha^3+\alpha^2)A \\ a_3 &= (\alpha^3+\alpha^2)A^{16} + (\alpha^4+1)A^8 + (\alpha^2+1)A^4 \\ & \quad +(\alpha+1)A^2 + (\alpha^4+\alpha^3+\alpha)A \\ a_4 &= (\alpha^4+\alpha^3+\alpha)A^{16} + (\alpha^3+\alpha^2)A^8 + (\alpha^4+1)A^4 \\ & \quad +(\alpha^2+1)A^2 + (\alpha+1)A \end{cases}$$

Similarly we can compute equivalent polynomials $\mathcal{H}_i(R)$ for r_i and $\mathcal{T}_j(B)$ for b_j, respectively. Using those polynomial equations, it is sufficient to translate all bit-level inputs in the remainder polynomial because of following lemma:

Proposition 6.1 *Remainder of S-poly reduction will only contain primary inputs (bit-level) and word-level output; furthermore, there will be one and only one term containing word-level output whose monomial is word-level output itself rather than higher order form.*

Proof. First proposition is easy to prove by contradiction: assume there exists an intermediate bit-level variable v in the remainder, then this remainder must be divided further by a polynomial with leading term v. Since the remainder cannot be divided by any other polynomials in J_{ckt}, the assumption does not hold.

Second part, the candidate pair of polynomials only have one term of single word-level output variable (say it is R) and this term is the last term under RATO, which means there is only one term with R in Spoly. Meanwhile in other polynomials from $J_{ckt} + J_0$ there is no such term containing R, so this term will be kept to remainder r, with exponent equals to 1. ■

By replacing all bit-level variables by corresponding word-level variable polynomials, we transform the remainder of $Spoly$ reduction to the form of $R' + R + \mathcal{F}'(A, B)$. Note R is present state notion of output, which equals to initial value $R = 0$ in first clock cycle, or value of R' from last clock cycle. By substituting R with its corresponding value (0 or a polynomial only about A and B), we get the desired polynomial function $R' + \mathcal{F}(A, B)$.

6.2.3 Solving a Linear System for Bit-to-Word Substitution

In Example 6.3 we use a Gaussian-elimination method to solve the system of polynomial equations. We describe another formal method to solve following system of equations

$$\begin{bmatrix} S \\ S^2 \\ S^{2^2} \\ \vdots \\ S^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} \beta & \beta^2 & \beta^{2^2} & \cdots & \beta^{2^{k-1}} \\ \beta^2 & \beta^{2^2} & \beta^{2^3} & \cdots & \beta \\ \beta^{2^2} & \beta^{2^3} & \beta^{2^4} & \cdots & \beta^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \beta^{2^{k-1}} & \beta & \beta^2 & \cdots & \beta^{2^{k-2}} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{k-1} \end{bmatrix} \tag{6.1}$$

Let s be a vector of k unknowns $s_0, \ldots, s_{k-1}$, then Equation 6.1 can be solved by using Cramer's rule:

$$s_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|}, \;\; 0 \le i \le k-1, |\mathbf{M}| \ne 0 \tag{6.2}$$

where $\mathbf{M_i}$ denotes a coefficient matrix replacing i-th column in $\mathbf{M}$ with vector $\mathbf{S} = [S \;\, S^2 \;\, \cdots \;\, S^{2^{k-1}}]^T$.

Notice that $\mathbf{M}$ is constructed by squaring a row and assigning it to next row, therefore its determinant exhibits certain special properties:

Definition 6.1 *Let $\{\alpha_0, \alpha_1, \ldots, \alpha_{k-1}\}$ be a set of k elements of $\mathbb{F}_{p^k}$. Then the determinant*

$$\det M(\alpha_0, \ldots, \alpha_{k-1}) = \begin{vmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{k-1} \\ \alpha_0^p & \alpha_1^p & \cdots & \alpha_{k-1}^p \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{p^{k-1}} & \alpha_1^{p^{k-1}} & \cdots & \alpha_{k-1}^{p^{k-1}} \end{vmatrix} \tag{6.3}$$

is called the **Moore determinant** *of set $\{\alpha_0, \ldots, \alpha_{k-1}\}$.*

Moore determinant can be written as an explicit expression

$$\det M(\alpha_0, \ldots, \alpha_{k-1}) = \alpha_0 \prod_{i=1}^{k-1} \left(\prod_{c_0, \ldots, c_{i-1} \in \mathbb{F}_p} (\alpha_i - \sum_{j=0}^{i-1} c_j \alpha_j) \right) \tag{6.4}$$

We use an example to help understanding the notations in Equation 6.4:

Example 6.4 *Let $\{\alpha_0, \alpha_1, \alpha_2\}$ be a set of elements of $\mathbb{F}_{2^3}$. Then*

$$\det M(\alpha_0, \alpha_1, \alpha_2) = \begin{vmatrix} \alpha_0 & \alpha_1 & \alpha_2 \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 \\ \alpha_0^4 & \alpha_1^4 & \alpha_2^4 \end{vmatrix} \tag{6.5}$$
$$= \alpha_0 \prod_{i=1}^{2} \prod_{c_0,\dots,c_{i-1} \in \mathbb{F}_2} (\alpha_i - \sum_{j=0}^{i-1} c_j \alpha_j)$$

First, let $i = 1$, we obtain $c_0 \in \mathbb{F}_2$. When $c_0 = 0$, the product term equals to α_1; when $c_0 = 1$ it equals to $(\alpha_1 - \alpha_0)$. Then let $i = 2$, we obtain $c_0, c_1 \in \mathbb{F}_2$, they can take value from $\{0, 0\}, \{0, 1\}, \{1, 0\}$ and $\{1, 1\}$. We add 4 more product terms $\alpha_2, (\alpha_2 - \alpha_1), (\alpha_2 - \alpha_0), (\alpha_2 - \alpha_0 - \alpha_1)$, respectively.

Thus, the result is

$$\det M(\alpha_0, \alpha_1, \alpha_2) = \alpha_0 \alpha_1 (\alpha_1 - \alpha_0) \alpha_2 (\alpha_2 - \alpha_1)(\alpha_2 - \alpha_0)(\alpha_2 - \alpha_0 - \alpha_1) \tag{6.6}$$

We discover through investigation that $|\mathbf{M}|$ has a special property when the set of elements forms a basis. The proof is given below:

Lemma 6.1 *Let $\{\alpha_0, \alpha_1, \dots, \alpha_{k-1}\}$ be a normal basis of $\mathbb{F}_{p^k}$ over $\mathbb{F}_p$. Then*

$$\det M(\alpha_0, \alpha_1, \dots, \alpha_{k-1}) = 1 \tag{6.7}$$

Proof. According to the definition Equation 6.4, the Moore determinant consists of all possible linear combinations of $\{\alpha_0, \alpha_1, \dots, \alpha_{k-1}\}$ with coefficients over $\mathbb{F}_p$. If $\{\alpha_0, \alpha_1, \dots, \alpha_{k-1}\}$ is a (normal) basis, then all product terms are distinct and represents all elements in the field $\mathbb{F}_{p^k}$. Since the product of all elements of a field equals to 1, the Moore determinant $|\mathbf{M}| = 1$. ■

Applying Lemma 6.1 to Equation 6.2 gives

$$s_i = |\mathbf{M_i}|, \ \ 0 \le i \le k - 1 \tag{6.8}$$

where $|\mathbf{M_i}|$ can be easily computed using Laplace expansion method, with complexity $O(k!)$.

6.2.4 The Overall Verification Approach

Based on above concepts and improvements, the functional verification for sequential GF multiplier on word-level with k-bit input operands A, B and k-bit output R is described as follows:

1) Given a sequential GF multiplier S, with word-level k-bit inputs A, B and output R.

2) Choose a primitive polynomial $P(x)$ of degree k and construct $\mathbb{F}_{2^k}$, and let $P(\alpha) = 0$.

3) Perform a reverse-topological traversal of S to derive RATO: LEX with $\{x_1 > x_2 > \cdots > x_d > R' > R > A > B\}$, where $\{x_1, \dots, x_d\}$ are bit-level variables of S, R' is the NS output value.

4) Derive the set of bit-level polynomials $\{f_1, \dots, f_s\}$ from each gate in S, and represent them using RATO. These polynomials are in the form $x_i + tail(f_i)$ where x_i is the output of corresponding gate.

5) Compose the bit-level to word-level correspondences polynomials such as $r_0\beta + r_1\beta^2 + \cdots + r_{k-1}\beta^{2^{k-1}} + R$. Add them to set $\{f_1, \dots, f_s\}$ and generate ideal J_{ckt}. Compose the ideal of vanishing polynomials $J_0 = \langle x_i^2 + x_i, \dots, R^{2^k} + R, \dots \rangle$.

6) Select the only critical pair (f_w, f_g) that does not have relatively prime leading terms. Compute $Spoly(f_w, f_g) \xrightarrow{J_{ckt}+J_0}_{+} r$.

7) Construct matrices $M_0, \dots, M_{k-1}$, where M_i is M with the i-column replaced by vector $[RR^2 \cdots R^{2^{k-1}}]^T$, and M is the reverse-circulant matrix generated by vector $(\beta, \beta^2, \dots, \beta^{2^{k-1}})$.

8) Symbolically compute the determinants $|M_i|$ to find set F_R, where $f_{R_i} : R_i + |M_i|$, for $0 \le i \le k-1$. Obtain F_A, F_B from F_R since they have the same form.

9) Compute $r \xrightarrow{F_A \cup F_B \cup F_R}_{+} r_w$, and iterate r_w as to^i in Algorithm 9. **Then r_w after iteration k is of the form $R' + \mathcal{F}(A, B)$ and it is the unique, canonical word-level abstraction of S over $\mathbb{F}_{2^k}$ after k clock-cycles**.

6.3 Software Implementation of Implicit Unrolling Approach

Our experiment on different size of SMPO designs is performed with both SINGULAR [33] symbolic algebra computation system and our customized toolset developed using C++. The SMPO designs are given as gate-level netlists with registers, then translated to polynomials to compose elimination ideal for Gröbner basis calculation. The experiment is conducted on desktop with 3.5GHz Intel Core™ i7 Quad-core CPU, 16 GB RAM and running 64-bit Linux OS.

The SINGULAR tool can read in scripts written in its own format similar to ANSI-C. The input file format for our C++ tool is also designed to compatible with Singular ring and polynomial definitions. For SMPO experiments, the main loop of our script file performs the same function as Algorithm 9 describes, while Gröbner basis computation in main loop can be divided into 4 different function parts:

1) Pre-process:

This step is executed only once before main loop starts. The function of pre-process is to compute following system of equations for bit-level inputs $a_0 \dots a_{k-1}$:

$$\begin{cases} a_0 & = f_0(A) \\ a_1 & = f_1(A) \\ \vdots & \\ a_{k-1} & = f_{k-1}(A) \end{cases}$$

The methodology has been discussed in this chapter. For 5-bit SMPO example, we start from bit-word correspondence polynomial

$$A + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}$$

and the result is

$$\begin{cases} a_0 & = & (\alpha+1)A^{16} + (\alpha^4+\alpha^3+\alpha)A^8 + (\alpha^3+\alpha^2)A^4 \\ & & +(\alpha^4+1)A^2 + (\alpha^2+1)A \\ a_1 & = & (\alpha^2+1)A^{16} + (\alpha+1)A^8 + (\alpha^4+\alpha^3+\alpha)A^4 \\ & & +(\alpha^3+\alpha^2)A^2 + (\alpha^4+1)A \\ a_2 & = & (\alpha^4+1)A^{16} + (\alpha^2+1)A^8 + (\alpha+1)A^4 \\ & & +(\alpha^4+\alpha^3+\alpha)A^2 + (\alpha^3+\alpha^2)A \\ a_3 & = & (\alpha^3+\alpha^2)A^{16} + (\alpha^4+1)A^8 + (\alpha^2+1)A^4 \\ & & +(\alpha+1)A^2 + (\alpha^4+\alpha^3+\alpha)A \\ a_4 & = & (\alpha^4+\alpha^3+\alpha)A^{16} + (\alpha^3+\alpha^2)A^8 + (\alpha^4+1)A^4 \\ & & +(\alpha^2+1)A^2 + (\alpha+1)A \end{cases}$$

By replacing bit-level variable a_i with b_i, r_i or r'_i, and word-level variable A with B, R, R' respectively, we can directly get bit-word relation functions for another operand input, pseudo input and pseudo output.

One limitation to SINGULAR tool is the degree of a term cannot exceed 2^{63}, so when conducting experiments for SMPO circuits larger than 62 bits, we rewrite the degree using smaller integers (the feasibility of this rewriting can also be verified in following steps). Our C++ tool does not have data size limit, but it accumulates exponent when computing the power of a variable. Thus if the exponent is as large as 2^{63}, its runtime is very long. As a result, our C++ tool also adopts and benefits from this rewriting technique. Since the bit-to-word substitution procedure only requires squaring of equations each time, the exponent of word A can only be in the form 2^{i-1}, i.e. $A^{2^0}, A^{2^1}, \ldots, A^{2^{k-1}}$. To minimize the exponents presenting in both SINGULAR and our tool, we rewrite exponent 2^{i-1} to exponent i, i.e. $(A^{2^0}, A^{2^1}, \ldots, A^{2^{k-1}}) \rightarrow (A, A^2, \ldots, A^k)$. In this way result is rewritten to be

$$a_0 = (\alpha+1)A^5 + (\alpha^4+\alpha^3+\alpha)A^4 + (\alpha^3+\alpha^2)A^3 + (\alpha^4+1)A^2 + (\alpha^2+1)A$$

Thus the exponents do not exceed the SINGULAR data size limit.

This step requires limited substitution operations, so although we use the naive Gaussian elimination method (whose time complexity is $O(k^3)$), the time cost is trivial comparing to following steps.

2) S-poly reduction:

First, S-poly is calculated based on RATO, then reduced with the ideal composed by circuit description polynomials (J_{ckt}). For already finished experiments, naive reduction (multi-division) is adopted, and this step takes largest portion of total time consumption.

For SMPO experiments, reduced Spoly has following generic form (all coefficients are omitted):

$$redSpoly = R' + \mathcal{F}(a_i, b_j, r_l, A, B, R) = \sum r_i + \sum a_i b_i + \sum a_i B + \sum b_i A + R' + R$$

From observation, there is no cross-product terms of bit-by-bit or bit-by-word variables from the same input/output such as $a_i a_j, a_i A$, etc. Consider the necessary condition of our trick, this property of reduced Spoly guarantees the word level variable can only exist in the form $A^{2^{i-1}}$, after substituting bit-level variables with corresponding word-level variable.

3) Substitute bit-level variables in reduced S-poly, i.e.

$$\mathcal{F}(a_i, b_j, r_l, A, B, R) \xrightarrow{a_i+\mathcal{G}_i(A), b_j+\mathcal{G}_j(B), r_l+\mathcal{G}_l(R)} \mathcal{H}(A, B, R)$$

Use the result from pre-process, get rid of r_i, a_i and b_i through substitution. This step yields following polynomial (consider the trick we used):

$$R' + \mathcal{H}(A, B, R) = R' + \sum R^i + \sum A^i B^j \tag{6.9}$$

all coefficients omitted.

4) Substitute PS word-level variable R with inputs A and B, i.e.

$$\mathcal{F}(A, B, R) \xrightarrow{R+\mathcal{F}'(A,B)} \mathcal{F}(A, B)$$

In Equation 6.9, there are still terms containing R in the ideal we want to compute Gröbner basis. For PS variable R we can use relation $R + \mathcal{F}'(A, B)$, which is the last clock-cycle's output ($R' + \mathcal{F}'(A, B)$) with only leading term replaced in step "$from^i \leftarrow to^i$" in Algorithm 9. Basically this step has nothing different from the last one, however, the degree of its terms are not actual degree because of the exponent rewriting technique. For example, a power of r – r^m is originally $r^{2^{m-1}}$. So if $R + \mathcal{F}'(A, B)$ contains terms $A^i B^j$, the result after exponent rewriting is

$$(A^{2^{i-1}} B^{2^{j-1}})^{2^{m-1}} = A^{2^{((i+m-2) \bmod k)+1}} B^{2^{((j+m-2) \bmod k)+1}}$$

Thus the actual exponent for A and B in $(A^i B^j)^m$ should be $((i + m - 2) \bmod k) + 1$ and $((j + m - 2) \bmod k) + 1$, respectively.

Within one iteration, after finishing steps 2) to 4), the output should be intermediate multiplication result (temporary product) $R' + \mathcal{F}(A, B)$. After k iterations, the output is $R + A \cdot B$ when SMPO circuit is bug-free.

6.4 Experimental Results

We have implemented our approach within the SINGULAR symbolic algebra computation system [v. 3-1-6] [33] as well as C++/GCC. Using our implementation, we have performed experiments to verify two SMPO architectures — Agnew-SMPO [14] and the RH-SMPO [15] — over $\mathbb{F}_{2^k}$, for various datapath/field sizes. Bugs are also introduced into the SMPO designs by modifying a few gates in the combinational logic block. Experiments using SAT-, BDD-, and AIG-based solvers are also conducted and results are compared against our approach. Our experiments run on a desktop with 3.5GHz Intel Core$^{\text{TM}}$ i7 Quad-core CPU, 16 GB RAM and 64-bit Linux.

Evaluation of SAT/ABC/BDD based methods: To verify circuit S against the polynomial $\mathcal{F}$, we unroll the SMPO over k time-frames, and construct a miter against a combinational implementation of $\mathcal{F}$. A (pre-verified) $\mathbb{F}_{2^k}$ Mastrovito multiplier [4] is used as the *spec* model. This miter is checked for SAT using the *Lingeling* [39] solver. We also experiment with the Combinational Equivalence Checking (CEC) engine of the ABC tool [40], which uses AIG-based reductions to identify internal AIG equivalences within the miter to efficiently solve verification. The BDD-based VIS tool [37] is also used for equivalence check. The run-times for verification of (unrolled) RH-SMPO against Mastrovito *spec* are given in Table 6.2 – which shows that the techniques fail beyond 23 bit fields.

Table 6.2: Run-time for verification of bug-free RH-SMPO circuits for SAT, ABC and BDD based methods. *TO* = timeout 14 hrs

	Word size of the operands k-bits			
Solver	11	18	23	33
Lingeling	593	*TO*	*TO*	*TO*
ABC	6.24	*TO*	*TO*	*TO*
BDD	0.1	11.7	1002.4	*TO*

CEC between unrolled RH-SMPO and Agnew-SMPO also suffers the same fate (results omitted). In fact, both SMPO designs are based on slightly different mathematical concepts and their computations in all clock-cycles, except for the k^{th} one, are also different. These designs have no internal logical/structural equivalencies, and verification with SAT/BDDs/ABC is infeasible. Their dissimilarity is depicted in Table 6.3, where N_1 depicts the number of AIG nodes in the miter prior to *fraig_sweep*, and the nodes after *fraiging* are recorded as N_2; so $\frac{N_1-N_2}{N_1}$ reflects the proportion of equivalent nodes in original miter, which emphasizes the (lack of) *similarity* between two designs.

Table 6.3: Similarity between RH-SMPO and Agnew's SMPO

Size k	11	18	23	33
N_1	734	2011	3285	6723
N_2	529	1450	2347	4852
Similarity	27.9%	27.9%	28.6%	27.8%

Evaluation of Our Approach: Our algorithm inputs the circuit given in BLIF format, derives RATO, and constructs the polynomial ideal from the logic gates and the register/data-word description. We perform one $Spoly$ reduction, followed by the bit-level to word-level substitution, in each clock cycle. After k iterations, the final result polynomial R is compared against the spec polynomial. The run-times for verifying bug-free and buggy RH-SMPO and Agnew-SMPO are shown in Table 6.4 and Table 6.5, respectively. We can verify, as well as catch bugs in, up to 100-bit multipliers. Beyond 100-bit fields, our approach is infeasible – mostly due to the fact that the intermediate abstraction polynomial R is very dense and contains high-degree terms, which can be infeasible to compute. However, it should be noted that if we do not use the proposed bit-level to word-level substitution, and compute reduced Gröbner bases with RATO, then our approach does not scale beyond 33-bit datapaths.

Table 6.4: Run-time (seconds) for verification of bug-free and buggy RH-SMPO using our approach

Operand size k	33	51	65	81	89	99
#variables	4785	11424	18265	28512	34354	42372
#polynomials	3630	8721	13910	21789	26255	32373
#terms	13629	32793	52845	82539	99591	122958
Runtime(bug-free)	112.6	1129	5243	20724	36096	67021
Runtime(buggy)	112.7	1129	5256	20684	36120	66929

Table 6.5: Run-time (seconds) for verification of bug-free and buggy Agnew's SMPO our approach

Operand size k	36	66	82	89	100
#variables	6588	21978	33866	39872	50300
#polynomials	2700	8910	13694	16109	20300
#terms	12996	43626	67322	79299	100100
Runtime(bug-free)	113	3673	15117	28986	50692
Runtime(buggy)	118	4320	15226	31571	58861

6.5 Conclusions and Further Work

This proposal has described a method to verify sequential Galois field multipliers over $\mathbb{F}_{2^k}$ using computer algebra and algebraic geometry based approach. As sequential Galois field circuits perform the computations over k clock-cycles, verification requires an efficient approach to unroll the computation, and represent it as a canonical word-level multi-variate polynomial. Using algebraic geometry, we show that the unrolling of the computation at word-level can be performed by Gröbner bases and elimination term orders. Subsequently, we show that the complex Gröbner basis computation can be eliminated by means of a bit-level to word-level substitution, which is implemented using the binomial expansion over Galois fields and Gaussian elimination. Our approach is able to verify up to 100-bit sequential circuits, whereas contemporary techniques fail beyond 23-bit datapaths.

Our approach still has following limitations: first, it can only be applied on XOR-rich circuits, while most industrial designs are AND-OR gates dominant; second, it only uses naive bit-word abstraction based on functions of arithmetic circuits, which will be inefficient when the function does not have a straightforward expression (such as an implicit function); last but not least, Gröbner basis computation in our improved approach still requires a very long time. To overcome these limitations, further explorations are needed for my research.

One way to further boost the efficiency is to adopt techniques from sparse linear algebra. Analysis on experiment results shows major time consumption is on "multi-division" part. A matrix-based technique named as "F-4 style reduction" [41] can speed up the procedure dividing a low-degree polynomial with term-sparse polynomial ideal.

CHAPTER 7

FINDING UNSATISFIABLE CORES OF A SET OF POLYNOMIALS USING THE GRÖBNER BASIS ALGORITHM

Gröbner basis can find further applications to a vast field of other techniques in formal verification. Satisfiability (SAT) problem is the basis of most decidable decision problems, as well as the basis of many formal verification techniques. In this chapter, we discuss a special topic branching out from SAT theory. They are about a situation when SAT problems give negative answer, which are called unsatisfiability (UNSAT) problems. Within a set of constrains (e.g. clauses, formulas or polynomials) which is unsatisfiable, sometimes it is worthy to explore the reasons for UNSAT. From the execution of the GB algorithm, an auxiliary structure can be obtained to help explore the reasons causing UNSAT. This chapter introduces the details about the motivation, mechanism and implementation.

7.1 Motivation

In this section, we introduce the motivation of our UNSAT core extraction research. We start with defining an UNSAT problem, then review the previous work and applications to abstraction refinement. Moreover, we then present the need of finding a word-level analog to contemporary UNSAT reasoning techniques.

7.1.1 Preliminaries of SAT/UNSAT Theory

Boolean satisfiability (SAT) problem is one of most basic problems in computer science. In the following part we define the terminology related to SAT problem [42].

Definition 7.1 *A* **literal** *l is defined as a variable v, or its negation $\overline{v}$. A disjunction (OR relation) of literals forms a* **clause**, *i.e. $c = l_1 \vee l_2 \vee \cdots \vee l_k$. A Boolean formula can* *always be written as* **conjunctive normal form** *(CNF), which is the conjunction (AND relation) of clauses:* $F = c_1 \wedge c_2 \wedge \cdots \wedge c_k$.

Using above concepts, the SAT/UNSAT problem can also be formally defined:

Definition 7.2 *A* **satisfiability (SAT) problem** *is a decision problem that takes a CNF formula and returns that the formula is satisfiable (SAT), when there is an assignment of the variables making the formula evaluate to true. Otherwise, the formula is* **unsatisfiable (UNSAT)**.

Figure 7.1 shows a simple example of SAT problem on circuits verification: we need to verify whether subcircuit A and subcircuit B has the same function, so we build a miter circuit for their outputs X and Y, and the equivalence checking problem is turned into a SAT problem as follows:

Is subcircuit A functionally equivalent to subcircuit B?

$\Longleftrightarrow$ *Is it true that no Boolean vector assignment to primary inputs a,b,c such that Z=1?*

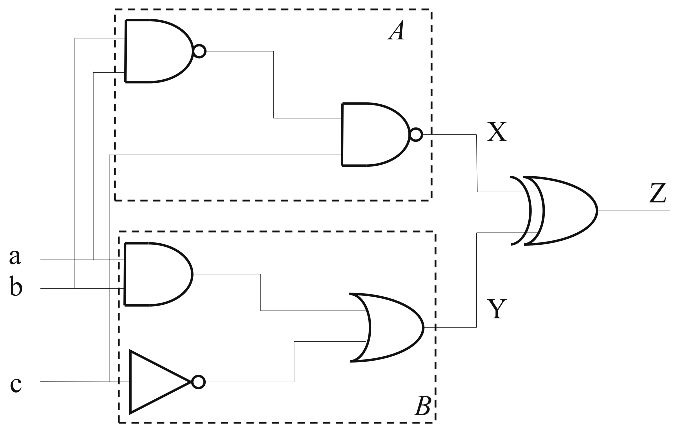


Figure 7.1: An example of Boolean satisfiability problem on circuits

For an UNSAT problem, the cause of UNSAT may include a part of clauses.

Definition 7.3 *Assume a CNF formula F is UNSAT. A subformula $M \subseteq F$ is an* **UNSAT core** *if M is also UNSAT. Further, if $\forall c \in M$, $M \backslash \{c\}$ is SAT, then M is called a* **minimal UNSAT core** *of F.*

7.1.2 Previous Work on CNF-UNSAT

In most cases, SAT problems are modeled as conjunctive normal form (CNF) formulas, and solved by procedures based on Davis-Putnam and Davis-Logemann-Loveland (DPLL) algorithm. The main idea of DPLL algorithm is recursive branching and backtrack searching. To improve the efficiency of DPLL-based SAT solver, efforts are made on minimizing number of branches, accelerating unit propagation and modifying backtrack algorithm. Recently state-of-art SAT solvers developed conflict-driven clause learning (CDCL) technique to prune the search space, which is effective to reduce result search time.

When a SAT solver fails to give a SAT assignment, it will provide a *UNSAT proof* or *refutation proof* to prove the problem is UNSAT. By analyzing clauses involved to the UNSAT proof, we can generate a subformula which remains to be UNSAT. A näive method is to collect all leaf clauses in an UNSAT proof as an UNSAT core. In practice, a minimal UNSAT core is more valuable. There are mainly 2 kinds of method to find minimal UNSAT cores. The one is insertion-based method, which is achieved by adding clauses to the smallest subset until the subset turns to be UNSAT. The other method is deletion-based method, which is realized by deleting clauses from a larger subset until the subset turns to be SAT. Recently heuristics such as clause-set refinement [43] and model rotation [44] are applied as improvements on the deletion/insertion based methods. These methods are also expanded to satisfiability modulo theories (SMT) [45].

On the other hand, researchers are seeking an alternative solution for SAT problem using a totally different method from "old-fashioned" DPLL algorithm. One promising option is polynomial calculus (PC), mapping Boolean variables and connectors in CNF formulas to variables and operators in Galois fields. In this way clauses are transformed to monomials/polynomials, thus theorems and concepts in computer algebra such as Hilbert's Nullstellensatz and Gröbner basis can be employed to assist finding valid assignments or proof of unsatisfiability. Basic concepts about PC will be formally introduced with definitions from computer algebra.

Boolean operator	Function over $\mathbb{F}_{2^k}$
$a \wedge b$	$a \cdot b$
$a \oplus b$	$a + b$
$\bar{a}$	$1 + a$
$a \vee b$	$a + b + a \cdot b$

Table 7.1: Mapping Boolean operators to functions over $\mathbb{F}_{2^k}$

The inspiration that use PC to solve SAT problems first comes from [46]. Using PC and its variations, researchers develop many SAT solvers [47–49]. Besides, researchers borrow concepts from PC, combine them with traditional DPLL and clause learning techniques to build hybrid SAT solvers [50, 51].

7.1.3 Exploiting UNSAT cores for abstraction refinement

Finding small UNSAT cores attracts interest for decades because of the need for all kinds of applications. In solving MaxSAT problems, small UNSAT proof provides lower bound for the branch-and-bound searching algorithm [52]. It is applied to solving Boolean algebra problems, such as Boolean function decomposition [53]. Small explanation generation in general constraint programming problems also rely on small UNSAT cores [54].

UNSAT cores can find wide range of applications in circuit verification as well. Many abstraction refinement techniques require information mining from UNSAT proofs of intermediate abstractions. Here we use an abstraction refinement algorithm from [55] to explain how an UNSAT proof is utilized in such techniques.

Bounded model checking (BMC) is a model checking technique which set upper bound to the length of all paths. It can be solved by solving a SAT problem. Given a model M, property p and bound k, The k-BMC unrolls M by k clock-cycles and generates Boolean formula F including violation check of p. Then F is fed to a SAT solver, if the SAT solver returns SAT, then p is violated in some paths shorter than k. Otherwise, UNSAT indicates p is not violated in all paths shorter than k. Algorithm 10 makes an improvement on BMC using abstraction refinement.

Algorithm 10: Abstraction refinement using k-BMC

```
    Input: $M$ – original machine, $p$ – property to check, $k$ – # of steps in $k$-BMC
    Output: If $p$ is violated, return error trace; otherwise $p$ is valid on $M$
 1  $k$ = InitValue;
 2  if $k$-BMC$(M, p, k)$ is SAT then
 3      return "Found error trace"
 4  else
 5      Extract UNSAT proof $\mathcal{P}$ of $k$-BMC;
 6      $M'$ = ABSTRACT$(M, \mathcal{P})$;
 7  end
 8  if MODEL-CHECK$(M', p)$ returns PASS then
 9      return "Passing property"
10  else
11      Increase bound $k$;
12      goto Line 2;
13  end
```

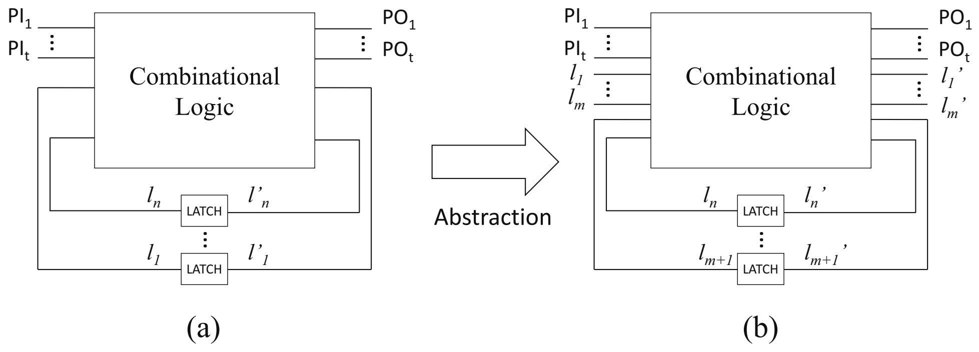


Figure 7.2: Abstraction by reducing latches

Assume that we are given a sequential circuit with n latches as shown in Figure 7.5(a). This circuit can be modeled as a Mealy machine M and the states s can be explicitly encoded by bit-level latch variables $l_1, \ldots, l_n$. Algorithm 10 describes an approach to check if machine M violates property p. This algorithm relies on k-BMC technique, which works on the basis of CNF-SAT solving. The k-BMC represents the initial states I, the transition relation T and property p as CNF formulas.

The first "if-else" branch in Algorithm 10 can be explained as: we check if the conjunction of formulas

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p$$

is SAT or not, where s_i denotes the set of reached states in i-th time-frame. If the result is SAT, then a counterexample is found that violates property p. If the result is UNSAT, we cannot assert that p is satisfied for the original machine M because we only unrolled M for a given specific number of time-frames without any fixpoint detection. In this algorithm, we analyze the UNSAT core composed by a set of clauses whose conjunction is UNSAT. If there are some latch variables (L_{abs}) not included in this UNSAT core, then we can assert that the evaluations of these variables will not affect the unsatisfiability of original formula. Therefore, we can ignore them in the abstracted model. In practice, we turn these latches into primary inputs/outputs as shown in Figure 7.5(b) ($L_{abs} = \{l_1, \ldots, l_m\}$).

The second "if-else" branch means: if we model check the abstracted machine M' and find no error trace, we can assert that property p also holds on the original machine M. The reason of this assertion is that the abstracted states represented using abstracted latches cover the original states, which means M' is an **over-approximation** of M, such that

$$(M' \implies p) \implies (M \implies p)$$

If we find a violation on abstracted machine, then this abstracted model is not a suitable model to check p, so we have to increase the bound k to find a finer abstraction.

It is clear that UNSAT cores play an important role in abstraction refinement approaches. In [55] the UNSAT core is extracted using a conventional CNF-SAT solver, which encounters the "bit-blasting" problem when the size of datapath (number of latches in Figure 7.5) is very large. **Here we propose an altogether new method based on Gröbner basis computation to extract UNSAT cores, and we believe it may become an efficient method according to the following observation:**

While the complexity of computing a GB over finite fields is exponential in the number of variables, the GB computation is observed to be more efficient for UNSAT problems. The reason is as follows:

Theorem 7.1 Weak Nullstellensatz: *Given ideal $J \subset \mathbb{F}[x_1, \dots, x_n]$, its variety over algebraic closure of field $\mathbb{F}$ is empty if and only if its reduced Gröbner basis contains only one generator "1".*

$$\mathbf{V}_{\overline{\mathbb{F}}}(J) = \emptyset \Longleftrightarrow \textit{reduced GB}(J) = \{1\}$$

It is well known that using Buchberger's algorithm and its variations to compute a GB has a very high space and time complexity and is usually not practical. One reason is that the size of the GB may explode even if the term ordering is carefully chosen. However if the reduced GB is $\{1\}$, which means every term in the original polynomials will be canceled, the degree of remainders when computing GB with Buchberger's algorithm will be limited. Thus the number of polynomials in non-reduced GB is much smaller than usual. Instead of applying polynomial calculus to SAT solving, it may be more efficient to try it for UNSAT problems.

Moreover, conventional techniques are limited to bit-level variables (literals). Algebraic geometry methods allow the use of word-level variables, which provides a strong potential for all applications which can be modeled as polynomials in finite fields extensions.

7.2 UNSAT Cores of Polynomial Ideals

In this section we provide a solution to the problem that how to use GB to extract the UNSAT core.

Problem statement: Let $F = \{f_1, \dots f_s\}$ be a set of multivariate polynomials in the ring $R = \mathbb{F}[x_1, \dots, x_d]$ that generate ideal $J = \langle f_1, \dots, f_s \rangle \subset R$. Suppose that it is known that $V(J) = \emptyset$, or it is determined to be so by applying the Gröbner basis algorithm. Identify a subset of polynomials $F_c \subseteq F$, $J_c = \langle F_c \rangle$, such that $V(J_c) = \emptyset$ too. Borrowing the terminology from the Boolean SAT domain, we call F_c the infeasible core or the unsat core of F.

It is not hard to motivate that an unsat core should be identifiable using the Gröbner basis algorithm: Assume that $F_c = F - \{f_j\}$. If $GB(F) = GB(F_c) = \{1\}$, then it implies that f_j is a member of the ideal generated by $(F - \{f_j\})$, i.e. $f_j \in \langle F - \{f_j\}\rangle$. Thus f_j can be composed of the other polynomials of F_c, so f_j is not a part of the unsat core, and it can be safely discarded from F_c. This can be identified by means of the GB algorithm for this ideal membership test.

A näive way (and inefficient way) to identify *a minimal core* using the GB computation is as follows: select a polynomial f_i and see if $V(F_c - \{f_i\}) = \emptyset$ (i.e. if reduced $GB(F_c - \{f_i\}) = \{1\}$). If so, discard f_i from the core; otherwise retain f_i in F_c. Select a different f_i and continue until all polynomials f_i are visited for inclusion in F_c. This approach will produce a minimal core, as we would have tested each polynomial f_i for inclusion in the core. This requires $O(|F|)$ calls to the GB engine, which is really impractical.

7.2.1 An Example Motivating our Approach

Buchberger's algorithm picks pairs of polynomials from a given set, computes their S-poly, then reduces this S-poly with the given set of polynomials. If the remainder is non-zero, it is added to the set of polynomials. By tracking S-poly computations and multivariate divisions that lead to remainder 1, we can obtain an UNSAT core. Moreover, we can identify a minimal UNSAT core with one-time execution of Buchberger's algorithm.

Example 7.1 *A SAT problem is described with 8 CNF clauses:*

$$\begin{array}{ll} c_1 : \bar{a} \vee \bar{b} & c_5 : x \vee y \\ c_2 : a \vee \bar{b} & c_6 : y \vee z \\ c_3 : \bar{a} \vee b & c_7 : b \vee \neg y \\ c_4 : a \vee b & c_8 : a \vee x \vee \neg z \end{array}$$

Using Boolean to polynomial mappings given in Table 7.1, we can transform them to a set of polynomials $F = \{f_1, \dots, f_8\}$ over ring $\mathbb{F}_2[a, b, x, y, z]$:

$f_1 : ab$	$f_5 : xy + y + x + 1$
$f_2 : ab + a$	$f_6 : yz + y + z + 1$
$f_3 : ab + b$	$f_7 : by + y$
$f_4 : ab + a + b + 1$	$f_8 : axz + az + xz + z$

We compute its GB using Buchberger's algorithm with lexicographic term ordering $a > b > x > y > z$. Since this problem is UNSAT, we will stop when "1" is added to GB.

1) First we compute $Spoly(f_1, f_2) \xrightarrow{F}_+ r_1$, remainder r_1 equals to a;

2) Update $F = F \cup r_1$;

3) Next we compute $Spoly(f_1, f_3) \xrightarrow{F}_+ r_2$, remainder r_2 equals to b;

4) Update $F = F \cup r_2$;

5) We can use a directed acyclic graph (DAG) to represent the process to get r_1, r_2, as Figure 7.3(a) shows;

6) Then we compute $Spoly(f_1, f_4) = s_3 = a + b + 1$, obviously $a + b + 1$ can be reduced (multivariate divided) by r_1 , the intermediate remainder $r_3 = b + 1$. It can be immediately divided by r_2, and the remainder is "1", we terminate the Buchberger's algorithm;

7) We draw a DAG depicting the process through which we obtain remainder "1" as shown in Figure 7.3(b). From leaf "1" we backtrace the graph to roots f_1, f_2, f_3, f_4. They constitute an UNSAT core for this problem as these polynomials are the "causes" of unsatisfiability of original set of polynomials.

We conclude our approach as a conjecture Algorithm 11.

7.2.2 The Refutation Tree of the GB Algorithm: Find F_c from F

We investigate if it is possible to identify a core by analyzing the $Spoly(f_i, f_j) \xrightarrow{F}_+ g_{ij}$ reductions in Buchberger's algorithm. Since F is itself an unsat core, definitely *there*

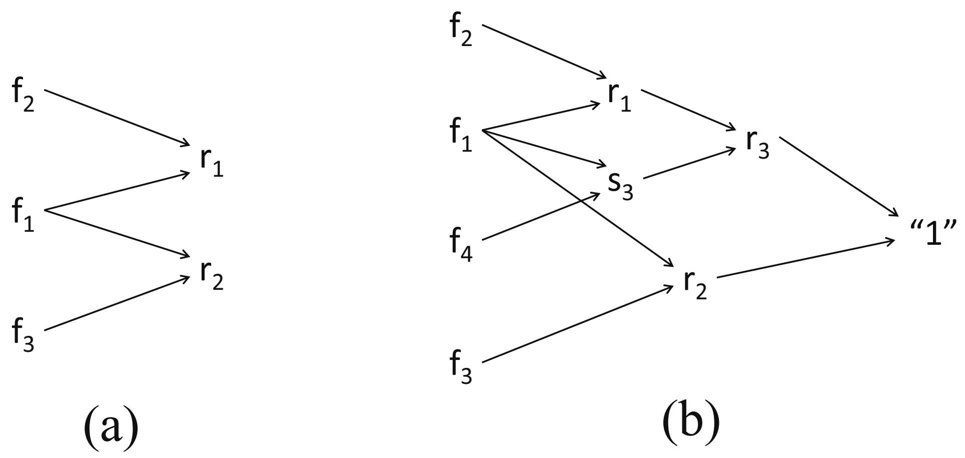


Figure 7.3: DAG representing Spoly computations and multivariate divisions

Algorithm 11: Extract UNSAT core using a variation of Buchberger's algorithm

```
Input: A set of polynomials $F = \{f_1, f_2, \dots, f_s\}$
Output: An UNSAT core $\{f_{m_1}, f_{m_2}, \dots, f_{m_t}\}$
repeat
    Pick a pair $f_i, f_j \in F$ that has never been computed S-poly;
    if $Spoly(f_i, f_j) \xrightarrow{F}_+ r_l \neq 0$ then
        $F = F \cup r_l$;
        Create a DAG $G_l$ with $f_i, f_j$ as roots, $r_l$ as leaf, recording the S-poly, all
        intermediate remainders and $f_k \in F$ that cancel monomial terms in the S-poly;
    end
until $r_l == 1$;
Backward traverse the DAG for remainder "1", replace $r_l$ with corresponding DAG
$G_l$;
return All roots
```

exists a sequence of Spoly reductions in Buchberger's algorithm where $Spoly(f_i, f_j) \xrightarrow{F}_+ 1$ *is achieved.* Moreover, polynomial reduction algorithms can be suitably modified to record which polynomials from F are used in the division leading to $Spoly(f_i, f_j) \xrightarrow{F}_+ 1$. This suggests that we should be able to identify a core by recording the *data* generated by Buchberger's algorithm — namely, the critical pairs(f_i, f_j) used in the Spoly computations, and the polynomials from F used to cancel terms in the reduction $Spoly(f_i, f_j) \xrightarrow{F}_+ 1$. The following example motivates our approach to identify $F_c \subseteq F$ using this data:

Example 7.2 *Consider the following set of polynomials $F = \{f_1, \ldots, f_9\}$:*

$f_1 : abc + ab + ac + bc$	$f_5 : bc + c$
$\quad + a + b + c + 1$	$f_6 : abd + ad + bd + d$
$f_2 : b$	$f_7 : cd$
$f_3 : ac$	$f_8 : abd + ab + ad + bd + a + b + d + 1$
$f_4 : ac + a$	$f_9 : abd + ab + bd + b$

Assume $>_{DEGLEX}$ monomial ordering with $a > b > c > d$. Let $F = \{f_1, \ldots, f_9\}$ and $J = \langle F \rangle \subset \mathbb{F}_2[a, b, c, d]$ where $\mathbb{F}_2 = \{0, 1\}$ is the finite field of 2 elements. Then $V(J) = \emptyset$ as $GB(J) = 1$. The set F consists of 4 minimal cores: $F_{c1} = \{f_1, f_2, f_3, f_4, f_7, f_8\}, F_{c2} = \{f_2, f_4, f_5, f_6, f_8\}, F_{c3} = \{f_2, f_3, f_4, f_6, f_8\}$, and $F_{c4} = \{f_1, f_2, f_4, f_5\}$.

Buchberger's algorithm terminates to a unique reduced GB, irrespective of the order in which the critical pairs (f_i, f_j) are selected and reduced by operation $Spoly(f_i, f_j) \xrightarrow{F}_+ g_{ij}$. Let us suppose that in the GB computation corresponding to Example 7.2, the first 3 critical *Spoly* pairs analyzed are $(f_1, f_2), (f_3, f_4)$ and (f_2, f_5). It turns out that the Spoly-reductions corresponding to these 3 pairs lead to the unit ideal. Recording the data corresponding to this sequence of reductions is depicted by means of a graph in Figure 7.4. We call this graph a *refutation tree*.

In the figure, the nodes of the graph correspond to the polynomials utilized in Buchberger's algorithm. The leaf nodes always correspond to polynomials from the given generating set. An edge e_{ij} from node i to node j signifies that the polynomial at node j results from polynomial at node i. For example, consider the computation $Spoly(f_1, f_2) \xrightarrow{F}_+ f_{10}$, where $f_{10} = a + c + 1$. Since $Spoly(f_1, f_2) = f_1 - ac \cdot f_2$, the leaves corresponding to f_1 and $ac \cdot f_2$ are created. The reduction $Spoly(f_1, f_2) \xrightarrow{F}_+ f_{10}$ is carried out as the following sequence of 1-step divisions: $Spoly(f_1, f_2) \xrightarrow{a \cdot f_2} \xrightarrow{f_3} \xrightarrow{c \cdot f_2} \xrightarrow{f_2} f_{10}$. This is depicted as the bottom subtree in the figure, terminating at polynomial f_{10}. Moreover, the multiplication $a \cdot f_2$ implies that division by f_2 resulted in the quotient a. The

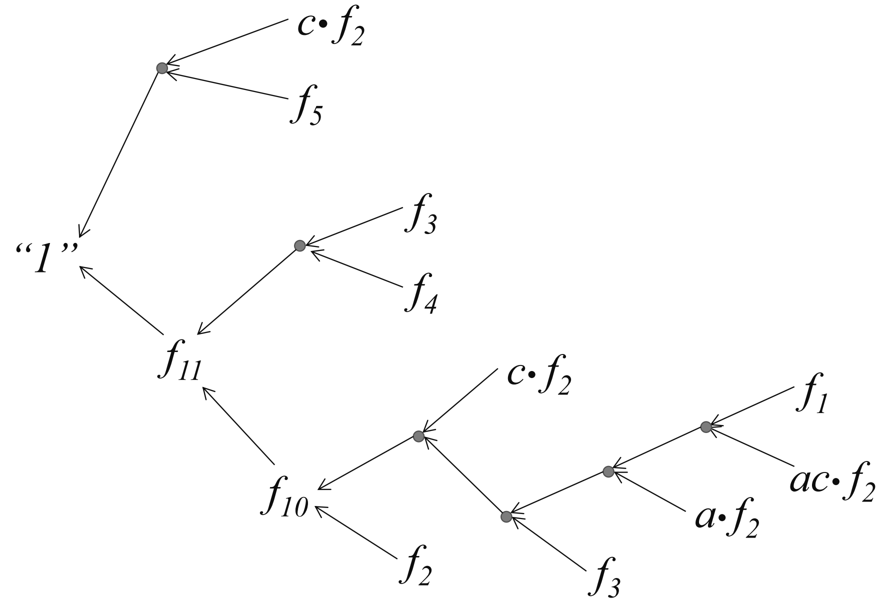


Figure 7.4: Generating refutation trees to record unsat cores

refutation tree of Figure 7.4 shows further that $Spoly(f_3, f_4) \xrightarrow{f_{10}} f_{11} = c + 1$ and, finally, $Spoly(f_5, f_2) \xrightarrow{f_{11}} 1$.

To identify an $F_c \subset F$, we start from the refutation node "1", and traverse the graph in reverse, all the way up to the leaves. Then, all the leaves in the transitive fanin of "1" constitute an unsat core. The polynomials (nodes) that do not lie in the transitive fanin of "1" can be safely discarded from F_c. From Figure 7.4, $F_c = \{f_1, f_2, f_3, f_4, f_5\}$ is identified as an unsat core of F.

7.3 Reducing the Size of the Infeasible Core F_c

The core F_c obtained from the aforementioned procedure may contain redundant elements which could be discarded. For example, consider the core $F_c = \{f_1, \ldots, f_5\}$ generated in the previous section. While F_c is a smaller infeasible core of F, it is not minimal. In fact, Example 1 shows that $F_{c4} = \{f_1, f_2, f_4, f_5\}$ is the minimal core, where $F_{c4} \subset F_c$. Clearly, the polynomial $f_3 \in F_c$ is a redundant element of the core and can be discarded. We will now describe techniques to further reduce the size of the unsat core by identifying such redundant elements. For this purpose, we perform a more systematic book-keeping of the data generated during the execution of Buchberger's algorithm and the refutation tree.

7.3.1 Identifying redundant polynomials from the refutation tree

We record the S-polynomial reduction $Spoly(f_i, f_j) \xrightarrow{F}_+ g_{ij}$ that give a non-zero remainder when divided by the system of polynomials F at that moment. The remainder g_{ij} is a polynomial combination of $Spoly(f_i, f_j)$ and the current basis F; thus, it can be represented as:

$$g_{ij} = S(f_i, f_j) + \sum_{k=1}^{m} c_k f_k, \tag{7.1}$$

where $0 \neq c_k \in \mathbb{F}[x_1, \ldots, x_d]$ and $\{f_1, \ldots, f_m\}$ is the "current" system of polynomials F. For each non-zero g_{ij}, we will record the following data:

$$((g_{ij})(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \ldots, (c_{kl}, f_{kl})) \tag{7.2}$$

In Equation 7.1 and Equation 7.2, g_{ij} denotes the remainder of the S-polynomial $Spoly(f_i, f_j)$ modulo the current system of polynomials $f_1, \ldots, f_m$, and we denote by

$$h_{ij} := \frac{LCM(lm(f_i), lm(f_j))}{lt(f_i)}, h_{ji} := -\frac{LCM(lm(f_i), lm(f_j))}{lt(f_j)}$$

the coefficients of f_i, respectively f_j, in the S-polynomial $Spoly(f_i, f_j)$. Furthermore, in Equation 7.2, $(c_{k1}, \ldots c_{kl})$ are the respective quotients of division by polynomials $(f_{k1}, \ldots, f_{kl})$, generated during the $Spoly$ reduction.

Example 7.3 *Revisiting Example 7.2, and Figure 7.4, the data corresponding to $Spoly(f_1, f_2)$ $\xrightarrow{F}_+ g_{12} = f_{10}$ reduction is obtained as the following sequence of computations:*

$$f_{10} = g_{12} = f_1 - acf_2 - af_2 - f_3 - cf_2 - f_2.$$

As the coefficient field is $\mathbb{F}_2$ in this example, $-1 = +1$, so:

$$f_{10} = g_{12} = f_1 + acf_2 + af_2 + f_3 + cf_2 + f_2$$

is obtained. The data is recorded according to Equation 7.2:

$$((f_{10} = g_{12}), (f_1, 1)(f_2, ac)|(a, f_2), (1, f_3), (c, f_2), (1, f_2))$$

Our approach and the book-keeping terminates when we obtain "1" as the remainder of some S-polynomial modulo the current system of generators. As an output of the Buchberger's algorithm, we can obtain not only the Gröbner basis $G = \{g_1, \dots, g_t\}$, but also a matrix M of polynomials such that:

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_t \end{bmatrix} = M \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} \tag{7.3}$$

Each element g_i of G is a polynomial combination of $\{f_1, \dots, f_s\}$. Moreover, this matrix M is constructed precisely using the data that is recorded in the form of Equation 7.2. We now give a condition when the matrix M may identify some redundant elements.

Theorem 7.2 *With the notations above, we have that a core for the system of generators $F = \{f_1, \dots, f_s\}$ of the ideal J is given by the union of those f_i's from F that appear in the data recorded above and correspond to the nonzero entries in the matrix M.*

Proof. In our case, since the variety is empty, and hence the ideal is unit, we have that $G = \{g_1 = 1\}$ and $t = 1$. Therefore $M = [a_1, \dots, a_s]$ is a vector. Then the output of the algorithm gives: $1 = a_1 f_1 + \cdots + a_s f_s$. Clearly, if $a_i = 0$ for some i then f_i does not appear in this equation and should not be included in the infeasible core of F.

■

Example 7.4 *Corresponding to Example 7.2 and the refutation tree shown in Figure 7.4, we discover that the polynomial f_3 is used only twice in the division process. In both occasions, the quotient of the division is 1. From Figure 7.4, it follows that:*

$$1 = (f_2 + f_5) + \cdots + \mathbf{1} \cdot \mathbf{f_3} + \cdots + \mathbf{1} \cdot \mathbf{f_3} + \cdots + (f_1 + f_2) \tag{7.4}$$

Since $1 + 1 = 0$ over $\mathbb{F}_2$, we have that the entry in M corresponding to f_3 is 0, and so f_3 can be discarded from the core.

7.3.2 The GB-Core Algorithm Outline

The following steps describe an algorithm (GB-Core) that allows us to compute a refutation tree of the polynomial set and corresponding matrix M.

Inputs: Given a system of polynomials $F = \{f_1, \ldots, f_s\}$, a monomial order $>$ on $\mathbb{F}[x_1, \ldots, x_d]$.

S-polynomial reduction: We start computing the S-polynomials of the system of generators $\{f_1, \ldots, f_s\}$, then divide each of them by the current basis $G = \{f_1, \ldots, f_s, \ldots, f_m\}$, which is the intermediate result of Buchberger's algorithm. In this way, we obtain expressions of the following type:

$$g_{ij} = \underbrace{h_{ij}f_i + h_{ji}f_j}_{Spoly(f_i,f_j)} + \sum_{k=1}^{m} c_k f_k \tag{7.5}$$

If the remainder g_{ij} is non-zero, we denote it by f_{m+1} and add it to the current set of generators G. We also record the data as in Equation 7.2:

$$((f_{m+1} = g_{ij})(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \ldots, (c_{kl}, f_{kl}))$$

This data forms a part of the refutation tree rooted at node f_{m+1}.

Recording the coefficients: In Equation 7.5 we obtain a vector of polynomial coefficients c_k where $k > s$. These coefficients are associated with new elements (remainders) in the Gröbner basis, that are not a part of the unsat core. Since each polynomial f_k, ($k > s$) is generated by $\{f_1, \ldots, f_s\}$, we can re-express f_k in terms of $\{f_1, \ldots, f_s\}$. Thus, each $f_k, k > s$ can be written as $f_k = d_1 f_1 + \cdots + d_s f_s$. This process adds a new row $(d_1, \ldots, d_s)$ to the coefficient matrix M.

Termination and refutation tree construction: We perform S-polynomial reductions and record these coefficients generated during the division until the remainder $f_m = 1$ is encountered. The corresponding data is stored in a data-structure D corresponding to Equation 7.2. The matrix M is also constructed. From this recorded data the refutation tree can be easily derived.

We start with the refutation node "$f_m = 1$":

$$((f_m = 1)(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \ldots, (c_{kl}, f_{kl}))$$

and recursively substitute the expressions for the polynomials f_k ($k > s$) until we obtain the tree with all the leaf nodes corresponding to the original set of polynomials $\{f_1, \dots, f_s\}$. Algorithm 12 describes this data recording through which the refutation tree T and the matrix M is derived.

Algorithm 12: GB-core algorithm (based on Buchberger's algorithm)

```
Input: $F = \{f_1, \dots, f_s\} \in \mathbb{F}[x_1, \dots, x_d], f_i \neq 0$
Output: Refutation tree $T$ and coefficients matrix $M$
 1: Initialize: list $G \leftarrow F$; Dataset $D \leftarrow \emptyset$; $M \leftarrow s \times s$ unit matrix
 2: for each pair $(f_i, f_j) \in G$ do
 3:    $f_{sp}, (f_i, h_{ij})(f_j, h_{ji}) \leftarrow$ Spoly($f_i, f_j$) {$f_{sp}$ is the S-polynomial}
 4:    $g_{ij}|(c_{k1}, f_{k1}), \dots, (c_{kl}, f_{kl}) \leftarrow (f_{sp} \xrightarrow{G}_+ g_{ij})$
 5:    if $g_{ij} \neq 0$ then
 6:       $G \leftarrow G \cup g_{ij}$
 7:       $D \leftarrow D \cup ((g_{ij})(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \dots, (c_{kl}, f_{kl}))$
 8:       Update matrix $M$
 9:    end if
10:    if $g_{ij} = 1$ then
11:       Construct $T$ from $D$
12:       Return($T, M$)
13:    end if
14: end for
```

Notice that the core can actually be derived directly from the matrix M. However, we also construct the refutation tree T as it facilitates an iterative refinement of the core, which is described in the next section.

7.4 Iterative Refinement of the Unsat Core

As with most other unsat core extractors, our algorithm also cannot generate a minimal core in one execution. To obtain a smaller core, we re-execute our algorithm with the core obtained in the current iteration. We describe two heuristics that are applied to our algorithm to increase the likelihood of generating a smaller core in the next iteration.

An effective heuristic should increase the chances that the refutation "1" is composed of fewer polynomials. In our GB-core algorithm, we use a strategy to pick critical pairs such that polynomials with larger indexes get paired *later* in the order:

$$(f_1, f_2) \rightarrow (f_1, f_3) \rightarrow (f_2, f_3) \rightarrow (f_1, f_4) \rightarrow (f_2, f_4) \rightarrow \cdots$$

Moreover, for the reduction process $Spoly(f_i, f_j) \xrightarrow{F}_{+} g_{ij}$, we pick divisor polynomials from F following the increasing order of polynomial indexes. Therefore, by relabeling the polynomial indexes, we can affect their chances of being selected in the unsat core. We use two criteria to to affect the polynomial selection in the unsat core. One corresponds to the *refutation distance*, whereas the other corresponds to the *frequency* with which a polynomial appears in the refutation tree.

Definition 7.4 (Refutation Distance) *Refutation distance of a polynomial f_i in a refutation tree corresponds to the number of edges on the shortest path from refutation node "1" to any leaf node that represents polynomial f_i.*

On a given refutation tree, polynomials with shorter refutation distances are used as divisors in later stages of polynomial reductions; which implies that they may generally have lower-degree leading terms. This is because we impose a degree-lexicographic term order, and successive divisions (term cancellations) reduce the degree of the remainders. However, what is more desirable is to use these polynomials with lower-degree leading terms earlier in the reduction, as they can cancel more terms. This may prohibit other (higher-degree) polynomials from being present in the unsat core.

Similarly, we can define the concept of another heuristic:

Definition 7.5 (Frequency of Occurrence) *Frequency of occurrence of a polynomial f_i in a refutation tree corresponds to the number it appears in the refutation tree.*

The motivation for using the *frequency of occurrence* of f_i in the refutation tree is as follows: polynomials that appear frequently in the refutation tree may imply that they have certain properties (leading terms) that give them a higher likelihood of being present in the unsat core.

We apply both heuristics: after the first iteration of the GB-core algorithm, we analyze the refutation tree T and sort the polynomials in the core by the refutation distance criterion, and use the frequency criterion as the tie-breaker. The following example illustrates our heuristic.

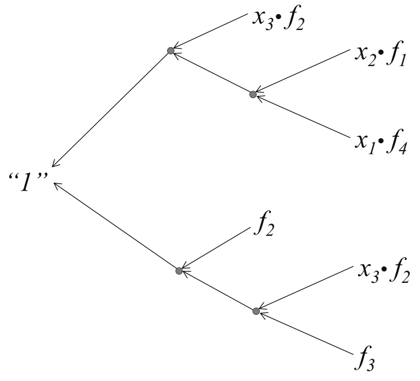
Example 7.5 *Consider a set of 6 polynomials over $\mathbb{F}_2$ of an infeasible instance.*

$$f_1 : x_1x_3 + x_3; \quad f_2 : x_2 + 1$$

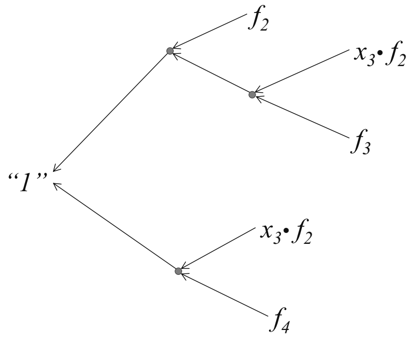
$$f_3 : x_2x_3 + x_2; \quad f_4 : x_2x_3$$

$$f_5 : x_2x_3 + x_2 + x_3 + 1; \quad f_6 : x_1x_2x_3 + x_1x_3$$

After the first iteration of the GB-core algorithm, the core is identified as $\{f_1, f_2, f_3, f_4\}$, and we obtain a refutation tree as shown in Figure 7.5(a).



(a)



(b)

Figure 7.5: Refutation trees of core refinement example

The refutation distance corresponding to polynomial f_2 is equal to 2 levels. Note that while three leaf nodes in Figure 7.5 (a) correspond to f_2, the shortest distance from "1" to any f_2 node is 2 levels. The refutation distance and frequency measures of other polynomials are identical – equal to 3 and 1, respectively – so their relative ordering is unchanged. We reorder f_2 to be the polynomial with the smallest index. We re-index the polynomial set $f_1' = f_2, f_2' = f_1, f_3' = f_3, f_4' = f_4$ and apply our GB-core algorithm on the core $\{f_1', f_2', f_3', f_4'\}$. The result is shown in Figure 7.5(b) with the core identified as $\{f_1', f_3', f_4'\} = \{f_2, f_3, f_4\}$. Further iterations do not refine the core – i.e. a fix point is reached.

7.5 Refining the Unsat core using Syzygies

The unsat core obtained through our GB-core algorithm is by nature a refutation polynomial that equals to 1:

$$1 = \sum_{i=1}^{s} c_i \cdot f_i$$

where $0 \neq c_i \in \mathbb{F}[x_1, \ldots, x_d]$ and the polynomials $F = \{f_1, \ldots, f_s\}$ form a core. Suppose that a polynomial $f_k \in F$ can be represented using a combination of the rest of the polynomials of the core, e.g.:

$$f_k = \sum_{j \neq k} c'_j f_j.$$

Then we can substitute f_k in terms of the other polynomials in the refutation. Thus, f_k can be dropped from the core as it is redundant. One of the limitations of the GB-core algorithm and the re-labeling/refinement strategy is that they cannot easily identify such polynomials f_k in the generating set F that can be composed of the other polynomials in the basis, i.e. $f_k \in \langle F - \{f_k\} \rangle$. We present an approach targeted to identify such combinations to further refine the core.

During the execution of Buchberger's algorithm, many critical pairs (f_i, f_j) do not add any new polynomials in the basis when $Spoly(f_i, f_j) \xrightarrow{F}_+ 0$ gives zero remainder. Naturally, for the purpose of the GB computation, this data is discarded. However, our objective is to gather more information from each GB iteration so as to refine the core. Therefore, we further record the quotient-divisor data from S-polynomial reductions that result in the remainder 0. Every $Spoly(f_i, f_j) \xrightarrow{F}_+ 0$ implies that some polynomial combination of $\{f_1, \ldots, f_s\}$ vanishes: i.e. $c_1 f_1 + c_2 f_2 + \cdots + c_s f_s = 0$, for some $c_1, \ldots, c_s$. These elements $(c_1, \ldots, c_s)$ form a syzygy on $f_1, \ldots, f_s$.

Definition 7.6 (Syzygy [17]) *Let $F = \{f_1, \ldots, f_s\}$. A syzygy on $f_1, \ldots, f_s$ is an s-tuple of polynomials $(c_1, \ldots, c_s) \in (\mathbb{F}[x_1, \ldots, x_d])^s$ such that $\sum_{i=1}^{s} c_i \cdot f_i = 0$.*

For each $Spoly(f_i, f_j) \xrightarrow{F}_+ 0$ reduction, we record the information on corresponding syzygies as in Equation 7.6, also represented in matrix form in Equation 7.7:

$$\begin{cases} c_1^1 f_1 + c_2^1 f_2 + \cdots + c_s^1 f_s = 0 \\ c_1^2 f_1 + c_2^2 f_2 + \cdots + c_s^2 f_s = 0 \\ \quad \vdots \\ c_1^m f_1 + c_2^m f_2 + \cdots + c_s^m f_s = 0 \end{cases} \tag{7.6}$$

$$\begin{bmatrix} c_1^1 & c_2^1 & \cdots & c_s^1 \\ c_1^2 & c_2^2 & \cdots & c_s^2 \\ \vdots & \vdots & \ddots & \vdots \\ c_1^m & c_2^m & \cdots & c_s^m \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} = 0 \tag{7.7}$$

Here $\{f_1, f_2, \ldots, f_s\}$ is the given core. Take one column of the syzygy matrix (e.g. the set of polynomials in j-th column $c_j^1, c_j^2, \ldots, c_j^m$) and compute its reduced Gröbner basis G_r. If $G_r = \{1\}$, then it means that there exists some polynomial vector $[r_1, r_2, \ldots, r_m]$ such that $1 = r_1 c_j^1 + r_2 c_j^2 + \cdots + r_m c_j^m = \sum_{i=1}^m r_i c_j^i$. If we multiply each row i in the matrix of Equation 7.7 with r_i, and sum up all the rows, we will obtain the following equation:

$$\begin{bmatrix} \sum_{i=1}^m r_i c_1^i & \cdots & 1 & \cdots & \sum_{i=1}^m r_i c_s^i \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} = 0 \tag{7.8}$$

This implies that

$$\sum_{i=1}^m r_i c_1^i f_1 + \cdots + f_j + \cdots + \sum_{i=1}^m r_i c_s^i f_s = 0,$$

or that f_j is a polynomial combination of $f_1, \ldots, f_s$ (excluding f_j). Subsequently, we can deduce that f_j can be discarded from the core. By repeating this procedure, some redundant polynomials can be identified and size of unsat core can be reduced further.

Example 7.6 *Revisiting Example 7.2, execute the GB-core algorithm and record the syzygies on $f_1, \ldots, f_s$ corresponding to the S-polynomials that give 0 remainder. The coefficients can be represented as entries in matrix shown below. For example, the first row in the matrix corresponds to the syzygies generated by $Spoly(f_1, f_3) \xrightarrow{F}_+ 0$.*

$$\begin{array}{r} \\ Spoly(f_1,f_3) \\ Spoly(f_2,f_3) \\ Spoly(f_1,f_4) \\ Spoly(f_2,f_4) \\ Spoly(f_1,f_5) \end{array} \begin{array}{c} \begin{array}{cccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} \end{array} \\ \begin{pmatrix} 1 & a+c+1 & b+1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & c+1 & 1 & b & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & a+c+1 & 0 & 0 & a & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{7.9}$$

Usually, we need to generate extra columns compared to the syzygy matrix of Equation 7.7. In this example, we need to add an extra column for the coefficient of f_{10}. This is because f_{10} is not among the original generating set; however, some S-polynomial pairs require this new remainder f_{10} as a divisor during reduction. In order to remove this extra column, we need to turn the non-zero entries in this column to 0 through standard matrix manipulations.

Recall that we record f_{10} in M as a nonzero remainder when reducing S-polynomial pair $Spoly(f_1, f_2) \xrightarrow{F}_+ f_{10}$. We extract this information from the coefficient matrix M:

$$(1 \;\; ac+a+c+1 \;\; 1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0)$$

It represents f_{10} is a combination of f_1 to f_9:

$$f_{10} = f_1 + (ac + a + c + 1)f_2 + f_3$$

It can be written in the same syzygy matrix form (with column f_{10} present) as follows:

$$\begin{array}{rc} & \begin{array}{cccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} \end{array} \\ Spoly(f_1,f_2) & \begin{pmatrix} 1 & ac+a+c+1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{7.10}$$

By adding this row vector (Equation 7.10) to the rows in Equation 7.9 corresponding to the non-zero entries in the column for f_{10}, we obtain the syzygy matrix only for the polynomials in the core:

$$\begin{array}{r} \\ Spoly(f_1,f_3) \\ Spoly(f_2,f_3) \\ Spoly(f_1,f_4) \\ Spoly(f_2,f_4) \\ Spoly(f_1,f_5) \end{array} \begin{array}{c} \begin{array}{ccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \end{array} \\ \begin{pmatrix} 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & ac & 1 & 0 & a & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

We find out there is a "1" entry in the f_3 column. The last row implies that f_3 is a combination of f_2, f_5 ($f_3 = acf_2 + af_5$), so f_3 can be discarded from the core.

The syzygy heuristic gathers extra information from the GB computation, it is still not sufficient to derive all polynomial dependencies. In Buchberger's algorithm, many S-polynomials reduce to zero, so the number of rows of the syzygy matrix can be much larger than the size of original generating set. Full GB computation on each column of the syzygy matrix can become prohibitive to apply iteratively. For this reason, we only apply the syzygy heuristic on the smaller reduced core given by our iterative refinement algorithm.

Our Overall Approach for Unsat Core Extraction: i) Given the set $F = \{f_1, \ldots, f_s\}$, we apply the GB-core algorithm, record the data D, M (Section 4) and the syzygies S on $f_1, \ldots, f_s$. ii) From M, we obtain a core $F_c \subseteq F$. iii) Iteratively refine F_c (Section 5) until $|F_c|$ cannot be reduced further. iv) Apply the syzygy-heuristic (Section 6) to identify if some $f_k \in F_c$ is a combination of other polynomials in F_c; all such f_k are discarded from F_c. This gives us the final unsat core F_c.

7.6 Application to Abstraction Refinement

In this section we apply our UNSAT core extraction approach to the k-BMC in Algorithm 8. This algorithm utilize UNSAT core to remove irrelevant latches (state variables) to reduce the state space. Since those state variables contribute nothing to the violation of property p, the abstracted model is a reasonable over-approximation for checking p without loss of accuracy. In the following example, we apply the UNSAT core extraction approach to a FSM which is complicated for bounded model checking and show the power of abstraction refinement to reduce the state space for refined k-BMC.

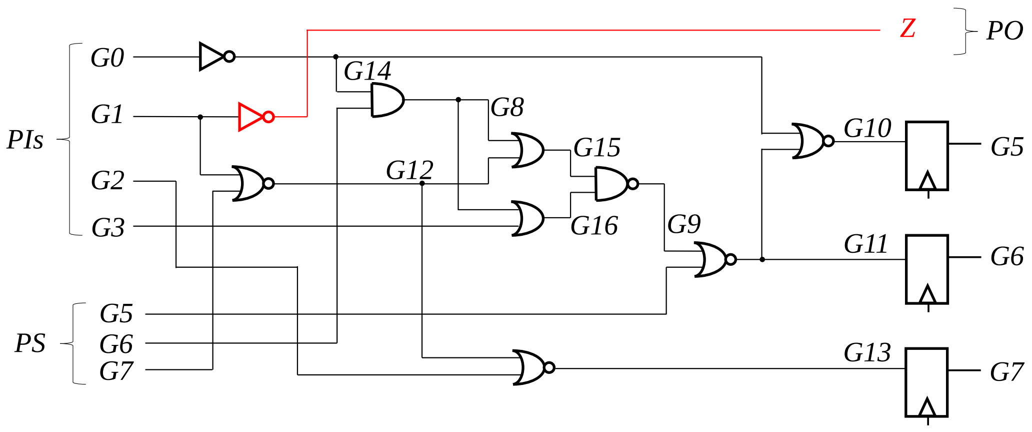


Figure 7.6: Gate-level Schematic of Example Circuit

Example 7.7 *Figure 7.6 shows a sequential circuit ("s27" from the ISCAS benchmark set) with 3-bit state registers PS* = $\{G7, G6, G5\}$ *and NS* = $\{G13, G11, G10\}$. *Its underlying FSM contains 8 states.*

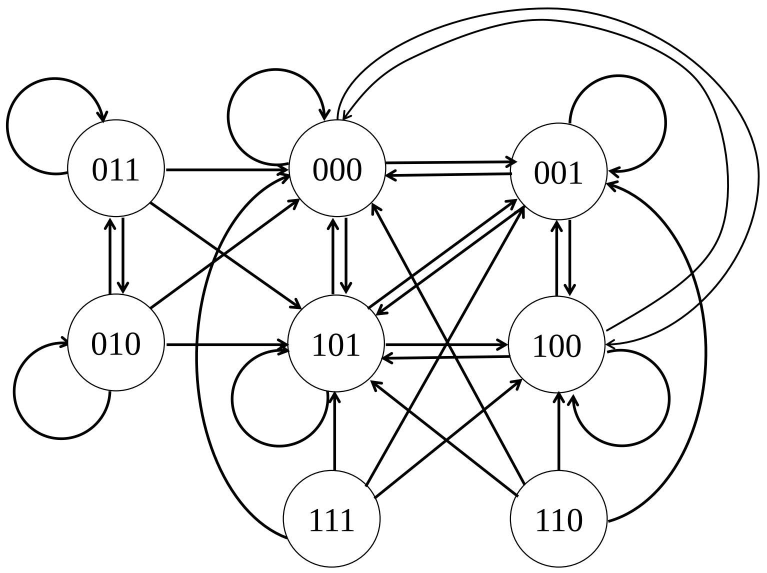


Figure 7.7: State transition graph of example circuit

On this FSM, define a linear temporal logic (LTL) property

$$p = \mathbf{AG}((\neg G13)\mathbf{U}(\neg Z))$$

with given initial state $\{000\}$*. Model checking on this machine requires traversal on the STG to search for an accepting trace. If we use* k*-BMC without abstraction refinement, we need to unroll the machine to iteration* $k = 3$*. We show how abstraction will be applied using our setup.*

Because the circuit has 3 latches, we model the problem over the field $\mathbb{F}_{2^3}$*. First, let the bound* $k = 0$*. Generate the polynomial constraints for initial state ideal* I *and check* $SAT(I \wedge \neg p)$ *using Gröbner bases. The ideal*

$$\begin{aligned} I = \langle & G14 + 1 + G0, G8 + G14 \cdot G6, G15 + G12 + G8 + G12 \cdot G8, \\ & G16 + G3 + G8 + G3 \cdot G8, G9 + 1 + G16 \cdot G15, \\ & G10 + 1 + G14 + G11 + G14 \cdot G11, G11 + 1 + G5 + G9 + G5 \cdot G9, \\ & G12 + 1 + G1 + G7 + G1 \cdot G7, G13 + 1 + G2 + G12 + G2 \cdot G12, \\ & Z + 1 + G1, \\ & \textit{(Initial state 000)} G5, G6, G7 \rangle; \end{aligned}$$

Property $\neg p$ is also written as a polynomial in the first time-frame:

$$\neg p = Z \cdot G13 + 1$$

As $I \wedge \neg p$ is UNSAT by the weak Nullstellensatz, we extract an UNSAT core using our approach:

$$\begin{aligned} Core(I \wedge \neg p) = & G12 + 1 + G1 + G7 + G1 \cdot G7, G13 + 1 + G2 + G12 + G2 \cdot G12, \\ & Z + 1 + G1, G7; \end{aligned}$$

The result shows that state variables $\{G5, G6, G10, G11\}$ are irrelevant when considering the violation of p. Thus, we can remove the latches $G10, G11$, and make $G5, G6$ primary inputs. In this way, the number of state variables is reduced to 1, such that the new machine only contains 2 states.

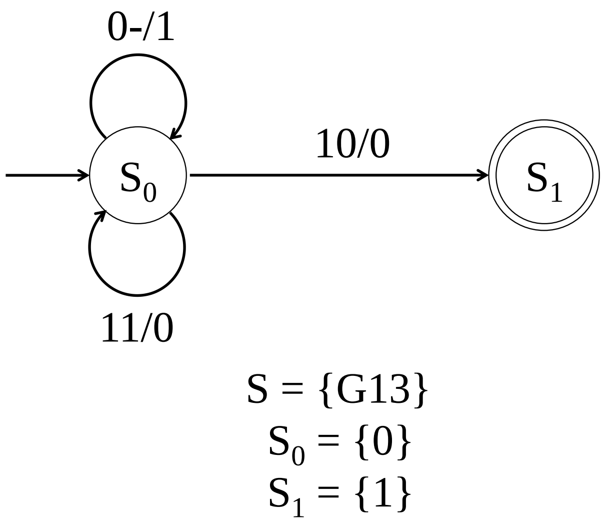


Figure 7.8: State transition graph of abstracted machine

Since the state space is greatly reduced, we can execute unbounded model checking on this abstracted machine with less cost. As a result, property p is not violated on the abstracted machine. Therefore, p is also a passing property of the original machine and the refined k-BMC algorithm terminates!

7.7 Experiment results

We have implemented our core extraction approach (the GB-Core and the refinement algorithms) using the SINGULAR symbolic algebra computation system [v. 3-1-6] [33]. With our tool implementation, we have performed experiments to extract a minimal unsat core from a given set of polynomials. Our experiments run on a desktop with 3.5GHz Intel Core$^{\text{TM}}$ i7-4770K Quad-core CPU, 16 GB RAM and 64-bit Ubuntu Linux OS. The experiments are shown in Table 7.2.

Gröbner basis is not an efficient engine for solving contemporary industry-size CNF-SAT benchmarks, as the translation from CNF introduces too many variables and clauses for GB engines to handle. On the other hand, although our approach is totally compatible with any constraints which can be written as polynomials in GF extensions, there is no such benchmark libraries clearly identifying a minimal core within to test our tool.

In order to validate our approach, we make a compromise and create a somewhat customized benchmark library by modifying SAT benchmarks and translating from circuit benchmarks: 1) ”aim-100” is a modified version of the random 3-SAT benchmark ”aim-50/100”, modified by adding some redundant clauses; 2) The ”subset” series are generated for random subset sum problems; 3) ”cocktail” is similarly revised from a combination of factorization and a random 3-SAT benchmark; 4) and ”phole4/5” are generated by adding redundant clauses to pigeon hole benchmarks; 5) Moreover, “SMPO” and “RH” benchmarks correspond to hardware equivalence checking instances of Agnew’s SMPO and RH-SMPO circuits [14, 15], compared against a golden model *spec*. Similarly, the ”MasVMon” benchmarks are the equivalence checking circuits corresponding to Mastrovito multipliers compared against Montgomery multipliers [56]. Some of these are available as CNF formulae, whereas others were available directly as polynomials over finite fields. The CNF formulae are translated as polynomial constraints over $\mathbb{F}_2$ (as shown in [57]), and the GB-Core algorithm and the refinement approach is applied.

In Table 7.2, #Polys denotes the given number of polynomials from which a core is to be extracted. #MUS is the *minimal* core either extracted by PicoMUS (for CNF benchmarks) or exhaustive deletion method (for non-CNF bencmarks). #GB-core iterations corresponds to the number of calls to the GB-core engine to arrive at the reduced unsat core. The second last column shows the improvement in the minimal core size by applying the syzygy heuristic on those cases which cannot be iteratively refined further. We choose PicoMUS as a comparison to our tool because it is a state-of-art MUS extractor, and the results it returned for our set of benchmarks are proved to be minimal. The data shows that in most of these cases, our tool can produce a minimal core. For the subset-3 benchmark, we obtain another core with even smaller size than the one from PicoMUS. The results demonstrate the power of the Gröbner basis technique to identify the causes of unsatisfiability.

Table 7.2: Results of running benchmarks using our tool. Asterisk(*) denotes that the benchmark was not translated from CNF. Our tool is composed by 3 parts: part I runs a single GB-core algorithm, part II applies the iterative refinement heuristic to run the GB-core algorithm iteratively, part III applies the syzygy heuristic.

Benchmark	# Polys	# MUS	Size of core			# GB-core iterations	Runtime (sec)			Runtime of PicoMUS (sec)
			I	II	III		I	II	III	
5×5 SMPO	240	137	169	137	137	8	1222	1938	1698	< 0.1
4×4 SMPO*	84	21	21	21	21	1	125	0.3	29	-
3×3 SMPO*	45	15	15	15	15	1	6.6	0.2	5.7	-
3×3 SMPO	17	2	2	2	2	1	0.07	0.01	0.01	< 0.1
4×4 MasVMont*	148	83	83	83	83	1	23	139	12	-
3×3 MasVMont*	84	53	53	53	53	1	4.3	4.6	0.9	-
2×2 MasVMont	27	23	24	23	23	2	1.3	1.0	80	< 0.1
5×5 RH*	142	34	48	35	35	4	997	1.0	80	-
4×4 RH*	104	35	43	36	36	3	96	5.7	0.6	-
3×3 RH*	50	20	20	20	20	1	2.9	3.5	10	-
aim-100	79	22	22	22	22	1	43	0.7	0.2	< 0.1
cocktail	135	4	6	4	4	2	51	0.01	0.01	< 0.1
subset-1	100	6	6	6	6	1	2.4	0.01	0.01	< 0.1
subset-2	141	19	37	23	21	2	12	1.6	1.1	< 0.1
subset-3	118	16	13	12	11	2	8.6	0.2	0.07	< 0.1
phole4	104	10	16	16	10	1	4.3	0.2	0.5	< 0.1
phole5	169	19	30	25	19	3	12	3.2	2.7	< 0.1

7.8 Conclusions

This paper addresses the problem of identifying an infeasible core of a set of multivariate polynomials, with coefficients from a field, that have no common zeros. The problem is posed in the context of computational algebraic geometry and solved using the Gröbner basis algorithm. We show that by recording the data produced by the Buchberger's algorithm – the $Spoly(f_i, f_j)$ pairs, as well as the polynomials of F used in the division process $Spoly(f_i, f_j) \xrightarrow{F}_+ 1$ – we can identify certain conditions under which a polynomial can be discarded from a core. An algorithm was implemented within the Singular computer algebra tool and some experiments were conducted to validate the approach. While the use of GB engines for SAT solving has a rich history, the problem of unsat core identification using GB-engines has not been addressed by the SAT community. We hope that this paper will kindle some interest in this topic which is worthy of attention from the SAT community – particularly when there seems to be a renewal of interest in the use of Gröbner bases for formal verification [22, 56, 58, 59].

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Future Work

8.1.1 Multivariate polynomial ideal based FSM Traversal

8.1.2 New Diagram Structure accelerating Polynomial Reduction

8.1.3 Interpolation extraction using GB Algorithm

8.1.4 Verification of Integer Arithmetic Circuits

APPENDIX

REFERENCES

[1] Robert J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, 1987.

[2] S. Roman, *Field Theory*, Springer, 2006.

[3] R. Lidl and H. Niederreiter, *Finite Fields*, Cambridge University Press, 1997.

[4] E. Mastrovito, “VLSI Designs for Multiplication Over Finite Fields GF(2^m)”, *Lecture Notes in Computer Science*, vol. 357, pp. 297–309, 1989.

[5] P. Montgomery, “Modular Multiplication Without Trial Division”, *Mathematics of Computation*, vol. 44, pp. 519–521, Apr. 1985.

[6] C. Koc and T. Acar, “Montgomery Multiplication in GF(2^k)”, *Designs, Codes and Cryptography*, vol. 14, pp. 57–69, Apr. 1998.

[7] H. Wu, “Montgomery Multiplier and Squarer for a Class of Finite Fields”, *IEEE Transactions On Computers*, vol. 51, May 2002.

[8] M. Knežević, K. Sakiyama, J. Fan, and I. Verbauwhede, “Modular Reduction in GF(2^n) Without Pre-Computational Phase”, *in Proceedings of the International Workshop on Arithmetic of Finite Fields*, pp. 77–87, 2008.

[9] J. Lv, *Scalable Formal Verification of Finite Field Arithmetic Circuits using Computer Algebra Techniques*, PhD thesis, Univ. of Utah, Aug. 2012.

[10] Alfred J Menezes, Ian F Blake, XuHong Gao, Ronald C Mullin, Scott A Vanstone, and Tomik Yaghoobian, *Applications of finite fields*, vol. 199, Springer Science & Business Media, 2013.

[11] S. Gao, *Normal Basis over Finite Fields*, PhD thesis, University of Waterloo, 1993.

[12] Ronald C Mullin, Ivan M Onyszchuk, Scott A Vanstone, and Richard M Wilson, “Optimal normal bases in gf (p n)”, *Discrete Applied Mathematics*, vol. 22, pp. 149–161, 1989.

[13] Jimmy K Omura and James L Massey, “Computational method and apparatus for finite field arithmetic”, May 6 1986, US Patent 4,587,627.

[14] Gordon B. Agnew, Ronald C. Mullin, IM Onyszchuk, and Scott A. Vanstone, “An implementation for a fast public-key cryptosystem”, *Journal of CRYPTOLOGY*, vol. 3, pp. 63–79, 1991.

[15] Arash Reyhani-Masoleh and M Anwar Hasan, "Low complexity word-level sequential normal basis multipliers", *Computers, IEEE Transactions on*, vol. 54, pp. 98–110, 2005.

[16] Alper Halbutogullari and Çetin K Koç, "Mastrovito multiplier for general irreducible polynomials", *IEEE Transactions on Computers*, vol. 49, pp. 503–518, 2000.

[17] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, Springer, 2007.

[18] W. W. Adams and P. Loustaunau, *An Introduction to Gröbner Bases*, American Mathematical Society, 1994.

[19] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction from Combinational Circuits for Verification over Galois Fields", *IEEE Transactions on CAD (in review)*, 2016.

[20] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, PhD thesis, University of Innsbruck, 1965.

[21] T Dubé, Bud Mishra, and Chee-Keng Yap, "Complexity of buchbergers algorithm for gröbner bases", *Extended Abstract, New York University, New York*, 1986.

[22] S. Gao, A. Platzer, and E. Clarke, "Quantifier Elimination over Finite Fields with Gröbner Bases", *in Intl. Conf. Algebraic Informatics*, 2011.

[23] S. Gao, "Counting Zeros over Finite Fields with Gröbner Bases", Master's thesis, Carnegie Mellon University, 2009.

[24] David Hilbert, "Über die Theorie der algebraischen Formen", *Math. Annalen*, vol. 36, pp. 473–534, 1890.

[25] Charles E Leiserson and James B Saxe, "Retiming synchronous circuitry", *Algorithmica*, vol. 6, pp. 5–35, 1991.

[26] J-Y Jou and K-T Cheng, "Timing-driven partial scan", *in Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pp. 404–407. IEEE, 1991.

[27] Anmol Mathur and Qi Wang, "Power reduction techniques and flows at rtl and system level", *in 2009 22nd International Conference on VLSI Design*, pp. 28–29. IEEE, 2009.

[28] Jitesh Shinde and SS Salankar, "Clock gatinga power optimizing technique for vlsi circuits", *in 2011 Annual IEEE India Conference*, pp. 1–4. IEEE, 2011.

[29] Olivier Coudert and Jean Christophe Madre, "A unified framework for the formal verification of sequential circuits", *in The Best of ICCAD*, pp. 39–50. Springer, 2003.

[30] Hyunwoo Cho, Gary D Hachtel, and Fabio Somenzi, "Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 935–945, 1993.

[31] Priyank Kalla and Maciej Ciesielski, "A comprehensive approach to the partial scan problem using implicit state enumeration", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 810–826, 2002.

[32] Alex Wee, "Difference between iphone 7 and iphone 6s in a nutshell", *Image*, 2016.

[33] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 3-1-3 — A computer algebra system for polynomial computations", 2011, http://www.singular.uni-kl.de.

[34] UC Berkeley, "Berkeley logic interchange format (blif)", *Oct Tools Distribution*, vol. 2, pp. 197–247, 1992.

[35] Robert Brayton and Alan Mishchenko, "Abc: An academic industrial-strength verification tool", *in Computer Aided Verification*, pp. 24–40. Springer, 2010.

[36] Ellen M Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R Stephan, Robert K Brayton, and Alberto Sangiovanni-Vincentelli, "Sis: A system for sequential circuit synthesis", 1992.

[37] Robert K Brayton, Gary D Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, et al., "Vis: A system for verification and synthesis", *in Computer Aided Verification*, pp. 428–432. Springer, 1996.

[38] Tim Pruss, Priyank Kalla, and Florian Enescu, "Equivalence verification of large galois field arithmetic circuits using word-level abstraction via gröbner bases", *in Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pp. 1–6. ACM, 2014.

[39] Armin Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013", *Proceedings of SAT Competition 2013; Solver and*, p. 51, 2013.

[40] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool", *in Computer Aided Verification*, vol. 6174, pp. 24–40. Springer, 2010.

[41] J-C. Faugẽre, "A New Efficient Algorithm for Computing Gröbner Bases (F_4)", *Journal of Pure and Applied Algebra*, vol. 139, pp. 61–88, June 1999.

[42] Ana Petkovska, David Novo, Alan Mishchenko, and Paolo Ienne, "Constrained interpolation for guided logic synthesis", *in 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 462–469. IEEE, 2014.

[43] Joao Marques-Silva and Ines Lynce, "On improving mus extraction algorithms", *in International Conference on Theory and Applications of Satisfiability Testing*, pp. 159–173. Springer, 2011.

[44] Anton Belov and Joao Marques-Silva, "Accelerating mus extraction with recursive model rotation", *in Formal Methods in Computer-Aided Design (FMCAD), 2011*, pp. 37–40. IEEE, 2011.

[45] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani, "A simple and flexible way of computing small unsatisfiable cores in sat modulo theories", *in International Conference on Theory and Applications of Satisfiability Testing*, pp. 334–339. Springer, 2007.

[46] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo, "Using the groebner basis algorithm to find proofs of unsatisfiability", *in Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 174–183. ACM, 1996.

[47] Evgeny Pavlenko, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, Alexander Dreyer, Frank Seelisch, and G Greuel, "Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra", *in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6. IEEE, 2011.

[48] Alexey Lvov, Luis Alfonso Lastras-Montano, Viresh Paruthi, Robert Shadowen, and Ali El-Zein, "Formal verification of error correcting circuits using computational algebraic geometry", *in Formal Methods in Computer-Aided Design (FMCAD), 2012*, pp. 141–148. IEEE, 2012.

[49] Michael Brickenstein and Alexander Dreyer, "Polybori: A framework for gröbner-basis computations with boolean polynomials", *Journal of Symbolic Computation*, vol. 44, pp. 1326–1345, 2009.

[50] Christopher Condrat and Priyank Kalla, "A gröbner basis approach to cnf-formulae preprocessing", *in Tools and Algorithms for the Construction and Analysis of Systems*, pp. 618–631. Springer, 2007.

[51] Christoph Zengler and Wolfgang Küchlin, "Extending clause learning of sat solvers with boolean gröbner bases", *in Computer Algebra in Scientific Computing*, pp. 293–302. Springer, 2010.

[52] Chu Min Li and Felip Manya, "Maxsat, hard and soft constraints.", *Handbook of satisfiability*, vol. 185, pp. 613–631, 2009.

[53] Ruei-Rung Lee, Jie-Hong R Jiang, and Wei-Lun Hung, "Bi-decomposing large boolean functions via interpolation and satisfiability solving", *in Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 636–641. IEEE, 2008.

[54] Hadrien Cambazard and Barry OSullivan, "Reformulating table constraints using functional dependenciesan application to explanation generation", *Constraints*, vol. 13, pp. 385–406, 2008.

[55] Liang Zhang, *Design Verification for Sequential Systems at Various Abstraction Levels*, PhD thesis, Citeseer, 2005.

[56] J. Lv, P. Kalla, and F. Enescu, “Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits”, *IEEE Transactions CAD*, vol. 32, pp. 1409–1420, Sept. 2013.

[57] C. Condrat and P. Kalla, “A Gröbner Basis Approach to CNF formulae Preprocessing”, *in International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 618–631, 2007.

[58] Priyank Kalla, “Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation”, *in Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, pp. 2–2. FMCAD Inc, 2015.

[59] Amr Sayed-Ahmed, Daniel Gro, Mathias Soeken, Rolf Drechsler, et al., “Formal verification of integer multipliers by combining gr?? bner basis with logic reduction”, *in 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1048–1053. IEEE, 2016.