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

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, $\mathbb{Q}$. Note that while both of these examples are rings with an infinite number of elements, the number of elements in a ring can also be finite.

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.

The main concept of field theory is **Field Extensions**. 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 common 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, and the construction of their field extensions.

3.1.2 Galois Field

Galois fields, also known as 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 Elliptic Curve Cryptography as Galois field arithmetic circuits. 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 in 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 $\mathbb{F}_2$:*

$+$	0	1
0	0	1
1	1	0

Addition over $\mathbb{F}_2$

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

Multiplication over $\mathbb{F}_2$

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.*

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 $P(x) \in \mathbb{F}_2[x]$ 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 $\mathbb{R}$. 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* $a_3, \ldots, a_0$ *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.*

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	$\alpha^3 + \alpha^2 + \alpha + 1$

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

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* $\mathbb{F}_{2^k}$*, where* α *is 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 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, Fig **??** shows the containment of the fields $\mathbb{F}_2 \subset \mathbb{F}_4 \subset \mathbb{F}_{16}$. It's easy to see that since $\mathbb{F}_4 = \mathbb{F}_{2^2}$, it contains $\mathbb{F}_2$. Likewise $\mathbb{F}_{16} = \mathbb{F}_{4^2} = \mathbb{F}_{2^4}$ 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 $k \mid n$.*

CHAPTER 4

WORD-LEVEL TRAVERSAL OF FINITE STATE MACHINES USING ALGEBRAIC GEOMETRY

Reachability analysis is a basic component of sequential circuit verification, esp. 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, some major refactoring and attachments are made on original designs. Those modifications may introduce malfunctions or glitches to the whole logic. In localized simulation or formal verification (esp. equivalence checking), the modifications may be denied since the malfunctions or glitches are considered as "faults" in this circuit. However, if the circuit behavior is carefully investigated, it may come to a verdict 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**, 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 constrains upon the machine to make it start from designated initial states and take in 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 sit in 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 bring 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.

4.1 Motivation

The motivation of this part of my research consists of many inspirations and observations. In this dissertation, we will summarize those major ones into 3 topics. First, we start from a sequential circuit partial scan paper to discuss the inspirations of our work as well as some 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. Thus we cover both the necessity and sufficiency of developing such a word-level traversal technique in our work.

4.1.1 Limitations of Full Scan Algorithms

The inspiration of this research mainly comes from a journal paper [16]. In that paper, the author proposed an traversal algorithm using concept “implicit state enumeration”. Concretely, the algorithm is written as follows:

Algorithm 1: SIMPSON: Scan Register Selection using Implicit State Enumeration [16]

Input: Sequential circuit, number of registers to scan
Output: Scan registers listed in decreasing order of their non-controllability

```
1  from^0 = reached = S^0;
2  i = 0;
3  while TRUE do
4      i++;
5      to^i = IMAGE(Δ, from^{i-1});
6      new^i = to^i · ¬reached;
7      for each state variable r_j do
8          record_if_transitions_present_or_missing(r_j, new^i);
9          compute_degree_of_unsettability(r_j, new^i);
10     end
11     if new^i == 0 then
12         break;
13     end
14     reached = reached + new^i;
15     from^i = new^i;
16 end
17 /*                  FSM traversal completed                */
18 for each state variable r_j do
19     if missing transition for r_j then
20         scan state variable r_j;
21     end
22 end
23 illegal_states = bdd_complement(reached);
24 for each state variable r_j do
25     compute_degree_of_unateness(r_j, illegal_states);
26     non-controllability(r_j) = degree_of_unsettability(r_j)+degree_of_unateness(r_j);
27 end
28 order state variables in terms of their non-controllabilities;/*  Sorting   */
29 output the required scan registers;
```

Note that the former part of this algorithm describes a breath-first-search (BFS) traversal in state space. The traversal algorithm is a simple variation of BFS algorithm in graph theory 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 an functionally independent combinational logic component, the traversal algorithm analyzes the combinational logic and

derive the transition function for one-step reachability within current time-frame, and extend the result to complete execution paths through unrolling. If each state encoding (i.e. exact values in the selected registers) is explicitly exposed and counted during the unrolling procedure, this unrolling is called **explicit unrolling**. In the SIMPSON 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 SIMPSON 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 is easy to explode when the formulas become too long and too complicated. In the paper, the author make a compromise between accuracy and cost, and turn to a partial scan technique rather than a full scan. In my research work, the aim is to explore a word-level technique which can make a full scan available.

4.1.2 Word-level Data Flow on Modern Datapath Designs

The integrity 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 [17]. Such high integrity make the silicon implementation of large datapaths possible. In recent decades, leading by top-notch central processing units (CPUs) and high bandwidth memory (HBM), 64-bit or even larger datapaths frequently appear in digital ICs. Meanwhile, with the development of electrical design automation (EDA) tools, data flow on those datapaths are directly 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 executes faster by orders of magnitude.

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.

Example 4.1 (Verification of sequential multiplication hardware) *Figure* **??** *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 shift right by 1 bit. Finally when the most significant bit (MSB) of preloaded multiplicand arrives the MSB of product registers, the registers contains the result – 128 bits product. The behavior can be concluded as following algorithm:

Algorithm 2: 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

$$S = P_{1/2} + B \tag{4.1}$$

The state space will be greatly reduced.

On the other hand, the user of conventional model checker will have to write all these clauses. These clauses contain 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 inconveniences and confusions. However, if word-level techniques allow specification like equation 4.1, the verification tool will be very user-friendly and straightforward.

4.1.3 Prerequisites of Word-level Techniques

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.

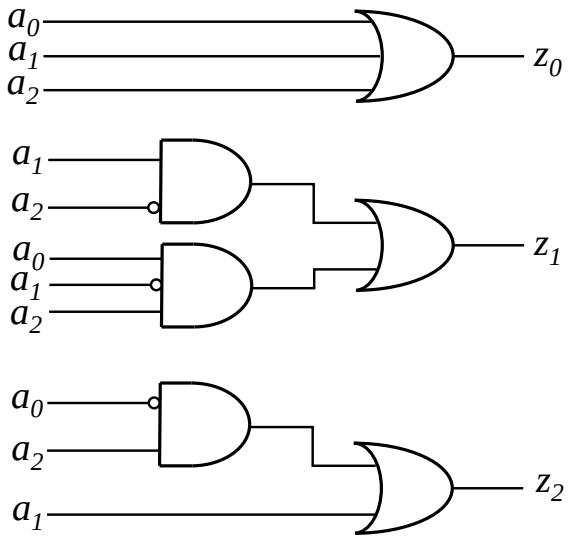


Figure 4.1: Gate-level netlist for Lagrange's interpolation example

Example 4.2 (Lagrange's interpolation) *Assume we are given gate-level netlist shown in figure 4.1. 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:

$\{a_2 a_1 a_0\} \in \mathbb{B}^3$	$A \in \mathbb{F}_{2^3}$	$\to$	$\{z_2 z_1 z_0\} \in \mathbb{B}^3$	$Z \in \mathbb{F}_{2^3}$
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	$\alpha^2 + 1$
101	$\alpha^2 + 1$	$\to$	011	$\alpha + 1$
110	$\alpha^2 + \alpha$	$\to$	101	$\alpha^2 + 1$
111	$\alpha^2 + \alpha + 1$	$\to$	101	$\alpha^2 + 1$

Table 4.1: 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:

$$\mathcal{F}(x) = \sum_{k=1}^{N} \left[\prod_{(0 \le j \le k-1),(j \ne i)} \frac{x - x_j}{x_i - x_j} \cdot y_k \right] \tag{4.2}$$

The geometry 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 4.2. In

this example, although defined in Galois field instead of real number field, the essential concept of Lagrange's interpolation keeps the same. We can get 8 "dots":

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

Substitute 8 (x_i, y_i) pairs in equation 4.2 with these 8 "dots" 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 too weak on scalability. Our approach uses abstraction based on Gröbner basis with abstraction term order (ATO), which is briefly introduced in preliminaries.

4.2 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 **??**.

4.2.1 Conventional Traversal Method

Conceptually, the state-space of a FSM is traversed in a breadth-first manner, as shown in Algorithm 3. 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 3, 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:

$$T(s, x, t) = \prod_{i=1}^{n} (t_i \overline{\oplus} \Delta_i) \tag{4.3}$$

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:

$$to = \text{Img}(\Delta, from) = \exists_s \exists_x [T(s, x, t) \cdot from] \tag{4.4}$$

Here, $\exists x(f)$ represents the *existential quantification of f w.r.t. variable x.*

Algorithm 3: BFS Traversal for FSM Reachability

Input: Transition functions Δ, 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 $reached$

Let us describe the application of the algorithm on the FSM circuit of figure 4.2. *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.*

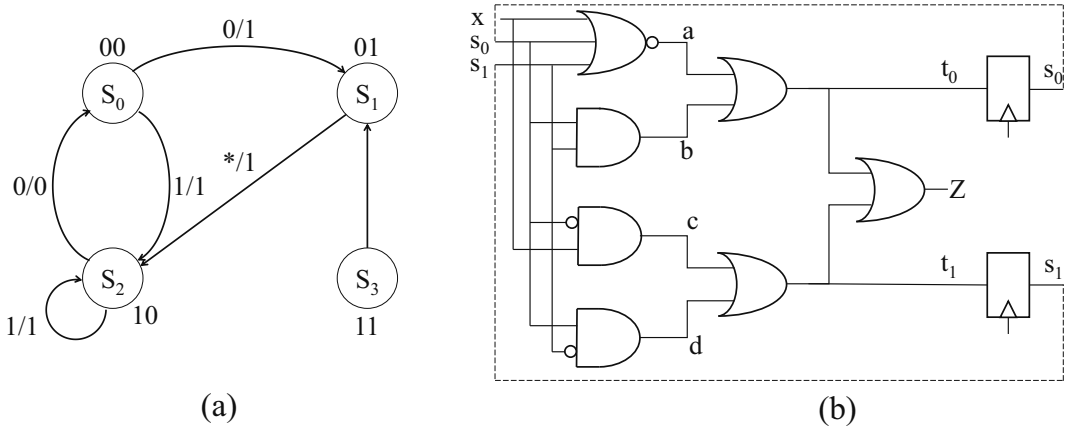


Figure 4.2: The example FSM and the gate-level implementation.

In Line 1 of the BFS algorithm, assume that the initial state is S_3 in figure 4.2(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 4.3 *For the circuit in figure 4.2 (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 Eqn. (4.4) 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* [18]. Finally, the set union, intersection and complement operations are also to be implemented in algebraic geometry.

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

The state transition graph (STG) shown in figure 4.2(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 3, 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 3, we need **union**, **intersection** and **complement** of varieties over $\mathbb{F}_{2^k}$, for which we again use algebraic geometry concepts.

Definition 4.1 *(***Sum/Product of Ideals** *[19]) If $I = \langle f_1, \ldots, f_r \rangle$ and $J = \langle g_1, \ldots, g_s \rangle$ are ideals in R, then the* **sum** *of I and J is defined as $I + J = \langle f_1, \ldots, f_r, g_1, \ldots, g_s \rangle$. Similarly, the* **product** *of I and J is $I \cdot J = \langle f_i g_j \mid 1 \leq i \leq r, 1 \leq j \leq s \rangle$.*

Theorem 4.1 *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 3, 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 4.2 *(**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 4.4 *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 4.1 *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 Def. 4.2, $hg \in \langle f\rangle$, which means $hg = f \cdot r$ with $r \in \mathbb{F}_{2^k}[x]$. Therefore, $hdg_1 = df_1 r$ and $hg_1 = f_1 r$. 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 4.2 *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 4.1.

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 4.2, the **absolute complement** $V(J')$ of a variety $V(J)$ can be computed as:

Corollary 4.1 *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 4.1, 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 4. 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.

4.2.3 Word-level FSM Traversal Example

Example 4.5 *We apply Algorithm 4 to the example shown in figure 4.2 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*

Algorithm 4: 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 $\langle reached \rangle$

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 4.1*

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

$V(\langle T^3 + 1 \rangle) = \{1, \alpha, \alpha+1\}$ *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.

Significance of using GB: A reduced GB is a unique, minimal and *canonical* representation of the circuit's function. Starting from a certain initial state and using a reduced GB to represent the transition function, reachable states can be computed and represented canonically. Then it becomes possible to identify when a fixpoint is reached (termination of the algorithm) by performing an equality check of polynomial ideals. Moreover, the GB computation is also used as a quantification procedure. As the GB is computed w.r.t. an elimination term order with "bit-level variables" $>$ "present-state word" $S >$ "next-state word" T, the set of reachable states are encoded, canonically, using a univariate polynomial in T, quantifying away the rest of the variables.

4.3 Improving our Approach

Using ATO on computing GB for large set of polynomial is time-consuming, and usually intractable. 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.

4.3.1 Simplifying the Gröbner Basis Computation

In algorithm 4, 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 [**?**]. According to proposition 2 in [**?**], 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 4.6 *After adding intermediate bit-level signal a, b, c, d, the elimination ideal for example circuit (example 4.5) can be rewritten LEX order with RATO:*

$$\begin{aligned}(t_0, t_1) &> (a, b, c, d)\\ &> T > (x, s_0, s_1)\end{aligned}$$

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

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

$$f_2 : b + s_0 s_1 \qquad f_3 : c + x + x s_0$$
$$f_4 : d + s_0 s_1 + s_0 \qquad f_5 : t_0 + ab + a + 1$$
$$f_6 : t_1 + cd + c + d \quad f_7 : t_0 + t_1 \alpha + T$$

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:

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

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.

$$s_i = \mathcal{G}(S) \tag{4.5}$$

Then we can substitute all bit-level variables with the word-level variable and obtain a word-level expression. The authors of [**?**] propose a method to construct a system of equations and solution to the system consists of equation 4.5. One way to obtain is to apply a Gaussian-elimination-fashion approach, which could compute corresponding $\mathcal{G}(S)$ efficiently with time complexity $O(k^3)$.

Example 4.7 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:

$$\begin{bmatrix} S \\ S^2 \\ S^{2^2} \\ \vdots \\ S^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \cdots & \alpha^{k-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(k-1)} \\ 1 & \alpha^4 & \alpha^8 & \cdots & \alpha^{4(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{2^{k-1}} & \alpha^{2\cdot 2^{k-1}} & \cdots & \alpha^{(k-1)\cdot 2^{k-1}} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{k-1} \end{bmatrix} \tag{4.6}$$

Treat $\mathbf{s}$ as a vector of k unknowns $s_0, \ldots, s_{k-1}$, then equation 5.14 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 5.14 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 $J_0^{PI} = \langle x_1^2 - x_1, \ldots, x_n^2 - x_n \rangle$. 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 Alg. 5.

4.3.2 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.

Algorithm 5: Refined Algebraic Geometry based FSM Traversal

Input: Input-output circuit characteristic polynomial ideal J_{ckt}, initial state polynomial $\mathbb{F}(S)$

$from^0 = reached = \mathbb{F}(S)$;
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 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 $\langle reached \rangle$

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

Proposition 4.1 *Divide n-bit PIs to partitions $n_1, n_2, \ldots, n_s$ and let n_1-bit, n_2-bit, ..., n_s-bit word-level variables represent their evaluations in $\mathbb{F}_{2^k}$ respectively. Then $n_1, n_2, \ldots, n_s \mid k$.*

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 4.3 *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*

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

Example 4.8 *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*

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

Using theorem 4.3 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.

4.4 Implementation of Word-level FSM Traversal Algorithm

In this section, we will introduce the tool implementation of our approach. The architecture of our tool consists of 4 functional components. First, given circuit is synthesized to gate-level using state-of-art synthesizers. Secondly, the synthesized netlist is translated to polynomial form and variables are sorted in RATO. This part is implemented using scripting language such as *Perl*. Thirdly, the polynomial reduction is executed using our customized reduction engine, which is dumped in C++. Finally, we utilize symbolic computation engine – SINGULAR [20] to code algorithm 5 and execute the BFS traversal. The result is given in a univariate polynomial about NS word T, denoting the set of reachable states from given initial state.

The following example illustrates the full-blown reachability analysis approach based on C++ and SINGULAR implementation.

Example 4.9 (Full Blown Reachability Analysis Walk-through) *Usually FSM designs can be described in behavior/structural hybrid languages. One of these languages is Berkeley logic interchange format (BLIF) [21], it allows state behavior representation and logic component representation.*

In this example, we use benchmark FSM "lion9" from MCNC benchmark library. Figure 4.3 is the truth table based structural BLIF representation of this FSM.

In order to compose the polynomial set in elimination ideal, we need to synthesize it to gate-level netlist. Modern synthesizers including ABC [22] and SIS [23] can complete this work. Figure 4.4 shows a synthesized FSM given in BLIF format. Note that for simplifying purposes in this example, we only use 2-input logic gates.

Using our interpreter, the synthesized BLIF file turns to a polynomial file customized for our polynomial reduction engine as figure 4.5 depicts. Note the first line includes all variables in RATO, the second line denotes the word-level variable. The third line is the irreducible polynomial of current GF ($\mathbb{F}_{2^4}$), the fourth line is the Spoly need to be reduced. The rest part has one polynomial in the elimination ideal in each line.

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 4.6, "tran" is the transition*

```
.model lion9
.inputs x0 x1
.outputs v6_4
.latch    v6_0  s0   0
.latch    v6_1  s1   0
.latch    v6_2  s2   1
.latch    v6_3  s3   0
.names [17] v6_4
0 1
.names [9] v6_0
0 1
.names [11] v6_1
0 1
.names [13] v6_2
0 1
.names [15] v6_3
0 1
.names x0 x1 s1 s3 [0]
0001 1
…
<omitted truth tables>
…
.names [1] [2] [6] [7] [11]
0000 1
.names [0] [4] [5] [7] [13]
0000 1
.names [2] [5] [6] [8] [15]
0000 1
.names [3] [5] [6] [7] [8] [17]
00000 1
.end
```

Figure 4.3: Sample input BLIF file

```
# synthesized BLIF
.model lion9_syn
.inputs x0 x1
.outputs v6_4
.latch    v6_0  s0   0
.latch    v6_1  s1   0
.latch    v6_2  s2   1
.latch    v6_3  s3   0

.gate INVX1   A=s1 Y=n16
.gate AND2X1  A=s3 B=n16_Y=n17
.gate NOR2X1  A=n16_B=x0 Y=n18_1
.gate AND2X1  A=n18_1 B=s3 Y=n19
.gate INVX1   A=x1 Y=n20
.gate AND2X1  A=n20 B=x0 Y=n21
.gate AND2X1  A=n21 B=s3 Y=n22
.gate NOR2X1  A=s3 B=x0 Y=n38
…
<omitted gate descriptions>
…
.gate OR2X1   A=n38 B=n8 Y=n39
.gate OR2X1   A=n39 B=n25 Y=n40
.gate OR2X1   A=n40 B=n19 Y=n18
.gate OR2X1   A=n34 B=n25 Y=n42
.gate OR2X1   A=n42 B=n22 Y=n43
.gate OR2X1   A=n43 B=n17 Y=n23
.end
```

Figure 4.4: Synthesized BLIF file

1. n23,n43,n42,n18,n40,n39,n38,n8,...<omitted bit-level intermediate vars>...,T,x0,x1,s0,s1,s2,s3;
2. T;
3. X^4+X+1;
4. T+n8+n13*X+n18*X^2+n23*X^3;
5. n17+s3*n16,
6. n19+n18_1*s3,
7. n21+n20*x0,
8. n22+n21*s3,
9. n23_1+s2*s1,
10. n24+x1*x0,
11. ...
12. <omitted polynomials>
13. ...
14. n27+n26*n22+n26+n22,
15. n28+n27*n19+n27+n19,
16. s4_4+n28*n17+n28+n17,
17. n35+n34*n33+n34+n33,
18. n36+n35*n22+n35+n22,
19. n13+n36*n19+n36+n19,
20. n39+n38*n8+n38+n8,
21. n40+n39*n25+n39+n25,
22. n18+n40*n19+n40+n19,
23. n42+n34*n25+n34+n25,
24. n43+n42*n22+n42+n22,
25. n23+n43*n17+n43+n17,
26. n16+s1+1,
27. n20+x1+1;

Figure 4.5: Polynomial file prepared for polynomial reduction engine

function we just obtained. "init_S" is the initial state, note it equals to "0100" register reset values in figure 4.3. Moreover, 2 bits PIs x_0, x_1 are combined to a word-level PI variable P using our conclusion in example 4.8: "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.

```
// ring var: RATO,  all bit-level inputs (PS and PI) followed by P, S and T
ring rr = (2,X), (n23,n43,n42,n18,n40,n39,...<omitted bit-level
intermediate vars>...,x0,x1,s0,s1,s2,s3,P,S,T), lp;
minpoly = X^4+X+1;

ideal A_in = s0,s1,s2,s3;
poly def_S = s0+s1*X+s2*X^2+s3*X^3+S;
ideal X_in = x0,x1;
poly def_X = x0 + x1*X^5+P;
poly red_S = S^16+S;  // GF(2^4)
poly red_T = T^16+T;
poly red_X = P^4+P;  // GF(2^2) as a subset field of GF(2^4)
// red_all: vanishing polys
ideal red_all = x0^2+x0, x1^2+x1, s0^2+s0, s1^2+s1, s2^2+s2,
s3^2+s3,red_S,red_X;
poly tran =
T+(X^2+X+1)*x0*x1*s1*s3+(X^3+X)*x0*x1*s1+...<omitted>...;

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

// Bit-word substitution
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);

// Iterative BFS traversal
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>MAX_iter))
        {
                "*************** 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 4.6: 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 4.7: The output given by our traversal tool

4.5 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 Alg. 5. 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] [20], as Line 3 in Alg. 5; 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 4.2.

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 [24].

Table 4.2: 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)			Runtime of VIS (sec)
						I	II	III	
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 4.2, # 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 Alg.4. 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.

4.6 Conclusion

This paper 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.

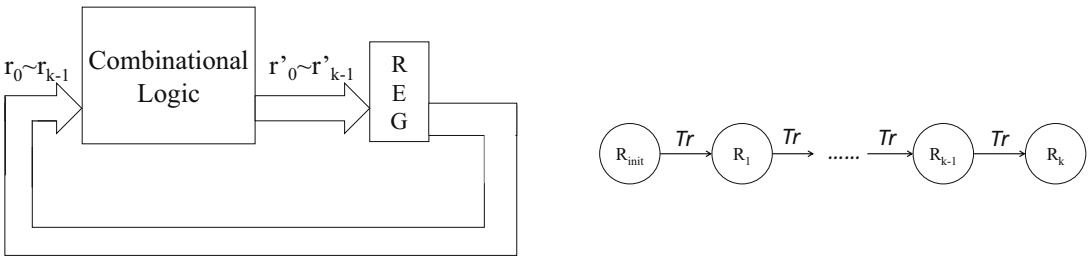
CHAPTER 5

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 et al. [**?**], 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. [**?**], 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 [**?**] 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.

5.1 Motivation

From the preliminaries (Chapter **??**) about FSMs, we learn that the Moore machine does not rely on inputs for state transitions. As depicted in Figure 5.1(a), a typical Moore machine implementation consists of combinational logic component and register files, where $r_0, \ldots, r_k$ are present state (PS) variables standing for state inputs (SI), and $r'_0, \ldots, r'_k$ are next state (NS) variables standing for state outputs (SO). Figure 5.1(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.



(a) (b)

Figure 5.1: 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 5.1(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 5.1(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. However, it also brings difficulties in verifying the the circuit functions.

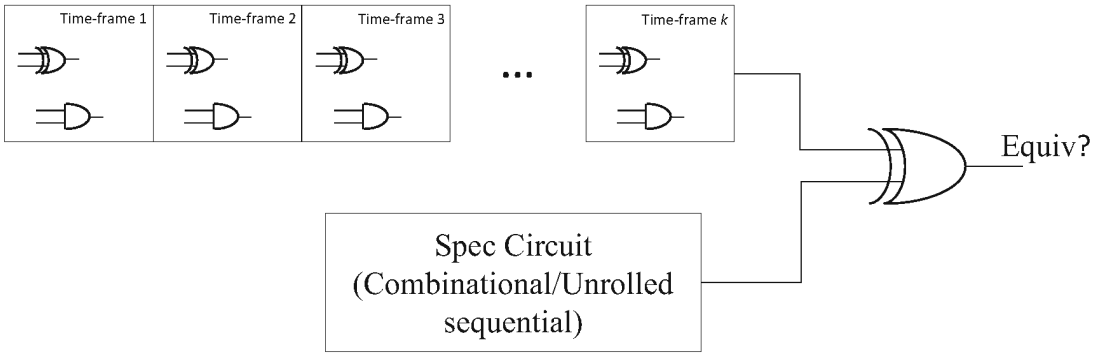


Figure 5.2: Conventional verification techniques based on bit-level unrolling and equivalence checking

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.

5.2 Normal Basis Multiplier over Galois Field

From algebraic view, a field is a space, and field elements are dots 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.

5.2.1 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,

Table 5.1: Bit-vector, Exponential and Polynomial representation of elements in $\mathbb{F}_{2^4} = \mathbb{F}_2[x] \pmod{x^4 + x^3 + 1}$

$a_3a_2a_1a_0$	Polynomial	$a_3a_2a_1a_0$	Polynomial
0000	0	1000	α^3
0001	1	1001	$\alpha^3 + 1$
0010	α	1010	$\alpha^3 + \alpha$
0011	$\alpha + 1$	1011	$\alpha^3 + \alpha + 1$
0100	α^2	1100	$\alpha^3 + \alpha^2$
0101	$\alpha^2 + 1$	1101	$\alpha^3 + \alpha^2 + 1$
0110	$\alpha^2 + \alpha$	1110	$\alpha^3 + \alpha^2 + \alpha$
0111	$\alpha^2 + \alpha + 1$	1111	$\alpha^3 + \alpha^2 + \alpha + 1$

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

$$\begin{aligned} A + B = A - B &= (a_0, a_1, \ldots, a_{k-1})_{StdB} \bigoplus (b_0, b_1, \ldots, b_{k-1})_{StdB} \\ &= (a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{k-1} \oplus b_{k-1})_{StdB} \end{aligned} \tag{5.1}$$

5.2.2 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

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

Respectively, a field element in NB representation is actually

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

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*. **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:

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

The addition and subtraction of elements in NB representation are similar to equation 5.1. However, what makes NB powerful is its property when doing multiplications and exponentiations. The following lemmas and examples illustrate this fabulous property very well.

Lemma 5.1 (Square of NB) *In $\mathbb{F}_{2^k}$, equation*

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

*has been proved. According to the **binomial theorem**, it can be extended to*

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

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 5.1 (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:

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

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 5.1 can be well utilized.

Example 5.2 (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}:

$$c_{n-1} = f_{mult}(a_0, a_1, \ldots, a_{n-1}; b_0, b_1, \ldots, b_{n-1}) \tag{5.2}$$

Before discussing the details of the function f_{mult}, we can take a square on both side of equation 5.2, i.e. $C^2 = A^2 \times B^2$. Obviously, using the property in lemma 5.1,

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 keeps the same. As a result, the new MSB can be written as

$$c_{k-2} = f_{mult}(a_{k-1}, a_0, a_1, \ldots, a_{k-2}; b_{k-1}, b_0, b_1, \ldots, b_{k-2}) \tag{5.3}$$

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 proved that a universal structure that implements f_{mult} can be reused for k times in NB multiplication over $\mathbb{F}_{2^k}$. Comparing to StdB, which requires distinct design for every bit of multiplication, NB is less costly if we can prove f_{mult} will not result in a structure with $O(k^2)$ complexity. So our next mission is to explore the details of f_{mult} to prove it will be a relatively simple design with complexity lower than $O(k^2)$.

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 (which is also the reason it is used as basic model to simulate the behavior of a neuron in neural network machine learning algorithms). 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

$$c_l = A \times C \times B$$

where C is a $k \times k$ square matrix.

Definition 5.1 (λ-Matrix) *A binary $k \times k$ matrix M is used to describe the bit-wise normal basis multiplication function f_{mult} where*

$$c_l = f_{mult}(A, B) = A \times M \times B^T \tag{5.4}$$

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 5.4, 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 5.3 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 5.3 (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

$$Product\ vector\ C = (\sum_{i=0}^{k-1} a_i \beta^{2^i})(\sum_{j=0}^{k-1} b_j \beta^{2^j}) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j \beta^{2^i} \beta^{2^j} \tag{5.5}$$

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

$$\beta^{2^i}\beta^{2^j} = \sum_{l=0}^{k-1} \lambda_{ij}^{(l)} \beta^{2^l}, \quad \lambda_{ij}^{(l)} \in \mathbb{F}_2. \tag{5.6}$$

Substitution yields, result is an expression for l-th digit of product as showed in equation 5.2:

$$c_l = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \lambda_{ij}^{(l)} a_i b_j \tag{5.7}$$

$\lambda_{ij}^{(l)}$ 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 $C_N \sim o(k)$ (symbol o denotes "strictly lower than bound"). 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 minimized 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.

5.2.3 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 5.4 (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.*

$$
\begin{array}{r}
101001 \\
\hline
111010111 \,\big)\, 11010110001110 \\
\underline{111010111} \\
111101101 \\
\underline{111010111} \\
111010110 \\
\underline{111010111} \\
1
\end{array}
$$

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.

From the intuition of humans, StdB multiplication is straightforward and easier to understand while NB is difficult to comprehend. However, if we implement both multiplications to hardware multipliers, it will be clear which side a circuit designer prefers.

Mastrovito multiplier and Montgomery multiplier 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 plus $k^2-\Delta$ XOR gates [25]. 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 plus $k^2 - k/2$ XOR gates [26]. Concretely, 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.

5.3 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.

5.3.1 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.

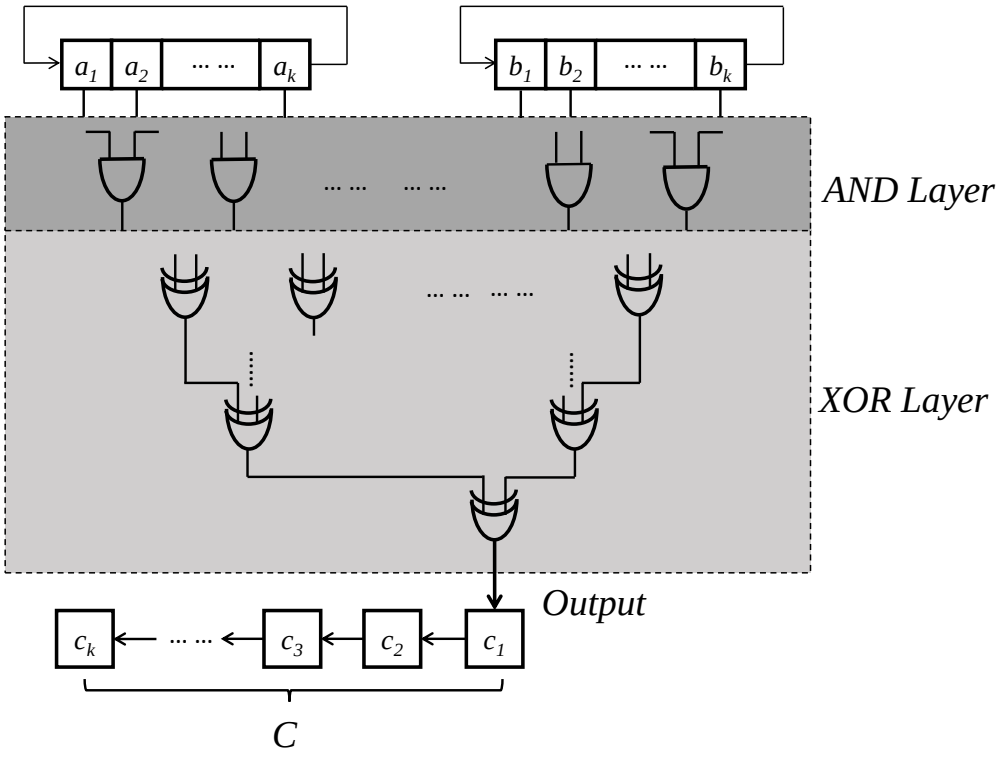


Figure 5.3: A typical SMSO structure of Massey-Omura multiplier

Figure 5.3 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 $a_i b_j$ and each XOR gate corresponds to addition $a_i b_j + a_{i'} b_{j'}$. 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:

$$c_l = row_1(A \times M^{(l)}) \times B + row_2(A \times M^{(l)}) \times B + \cdots + row_k(A \times M^{(l)}) \times B \quad (5.8)$$

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].

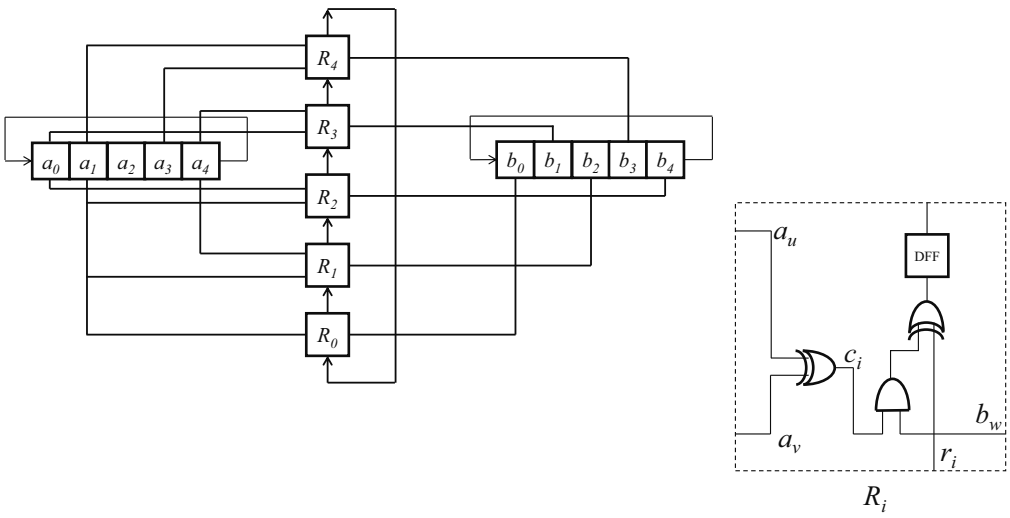


Figure 5.4: 5-bit Agnew's SMPO. Index i satisfies $0 < i < 4$, indices u, v are determined by column # of nonzero entries in i-th row of λ-Matrix $M^{(0)}$, i.e. if entry $M^{(0)}_{ij}$ is a nonzero entry, u or v equals to $i + j \pmod 5$. Index $w = 2i \pmod 5$

Example 5.5 (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 5.4.

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

$$c_0 = a_1b_0 + (a_0 + a_3)b_1 + (a_3 + a_4)b_2 + (a_1 + a_2)b_3 + (a_2 + a_4)b_4 \tag{5.9}$$

Note it is written in the form of equation 5.8. In next clock cycles we can obtain remaining bits of the product, which can be written in following general form polynomial:

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

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:

$$R_0^{(1)} = a_1b_0$$

It equals to the first term of equation 5.9. 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:

$$R_1^{(2)} = R_0^{(1)} + (a_0 + a_3)b_1 = a_1b_0 + (a_0 + a_3)b_1$$

which forms first 2 terms of equation 5.9. Similarly, we track the signal on R_2 in third clock cycle, signal on R_3 in fourth clock cycle, finally we can get

$$R_4^{(5)} = a_1b_0 + (a_0 + a_3)b_1 + (a_3 + a_4)b_2 + (a_1 + a_2)b_3 + (a_2 + a_4)b_4$$

which equals to c_0 in equation 5.9. 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 $R_1 \to R_2 \to R_3 \to \cdots \to R_0$, 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.

5.3.2 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.

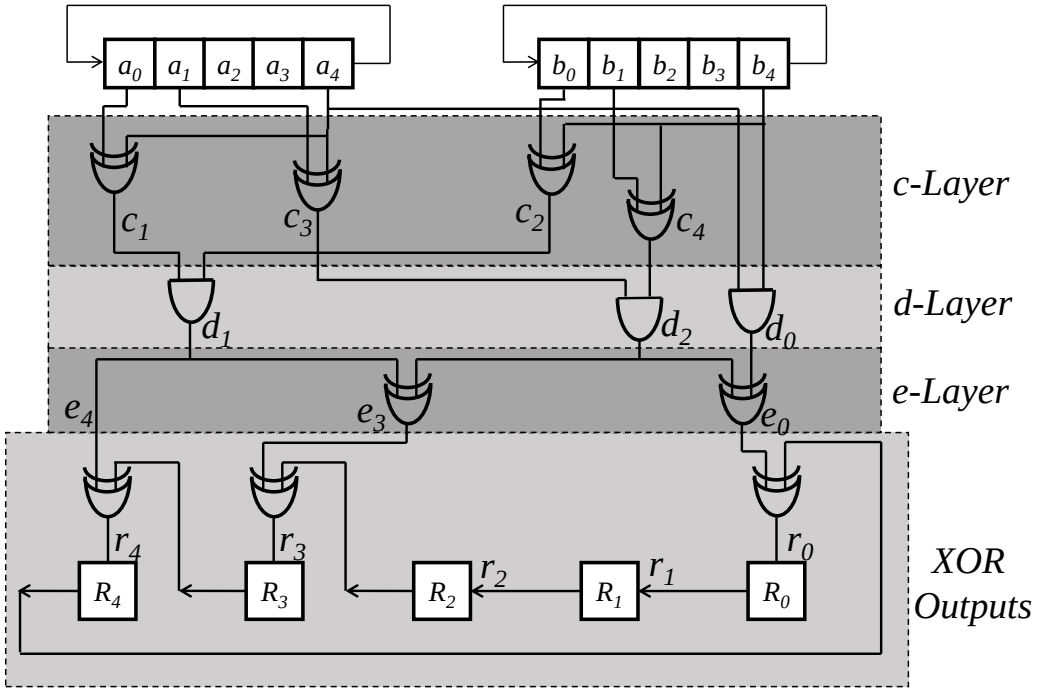


Figure 5.5: 5-bit RH-SMPO

Figure 5.5 is a 5-bit RH-SMPO which is functionally equivalent to 5-bit Agnew's SMPO in figure 5.4. A brief proof is as follows:

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

$$F_i(A, B) = a_i b_i \beta + \sum_{j=1}^{v} d_{i,j} \beta^{1+2^j} \tag{5.10}$$

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

$$d_{i,j} = c_{a,i} c_{b,i} = (a_i + a_{i+j})(b_i + b_{i+j}), \;\; 1 \leq j \leq v \tag{5.11}$$

$i+j$ 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 5.1 *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 6: 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 Eq.5.10*/;`
    $A \leftarrow A^2,\ B \leftarrow B^2$ `/*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 5.6 (Designing a 5-bit RH-SMPO) *From equation 5.10 we can deploy AND gates in d-layer according to $d_{i,j}$, and XOR gates in c-layer according to equation 5.11. Concretely, as algorithm 6 describes, we implement auxiliary function F_{k-1} in the logic:*

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

$$F_4(A, B) = a_4 b_4 \beta + \sum_{j=1}^{2} d_{4,j} \beta^{1+2^j} = d_0 \beta + \sum_{j=1}^{2} d_{4,j} \beta^{1+2^j} \tag{5.12}$$

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

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

$$c_3 = a_1 + a_4,\ c_4 = b_1 + b_4,\ d_2 = d_{4,2} = c_3c_4 = (a_4 + a_1)(b_4 + b_1)$$

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 $\mathbb{F}_2$:*

$$\begin{bmatrix} \beta^{1+2^0} \\ \beta^{1+2^1} \\ \beta^{1+2^2} \\ \vdots \\ \beta^{1+2^{k-1}} \end{bmatrix} = \beta \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} T_{0,0} & T_{0,1} & \cdots & T_{0,k-1} \\ T_{1,0} & T_{1,1} & \cdots & T_{1,k-1} \\ T_{2,0} & T_{2,1} & \cdots & T_{2,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ T_{k-1,0} & T_{k-1,1} & \cdots & T_{k-1,k-1} \end{bmatrix} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \mathbf{T} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} \tag{5.13}$$

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 refers to appendix A.2). Thus we can write down the M-table of $\mathbb{F}_{2^5}$ with current NB N:

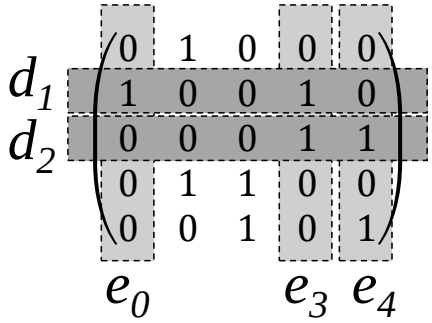


Figure 5.6: A 5×5 multiplication table

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 5.13:*

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

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 5.12, 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 5.6, 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 $2k + \lfloor \frac{k}{2} \rfloor$.

5.4 Full-Blown Verification Procedure for Normal Basis Multiplier Functional Correctness Checking

Once a gate-level design of a NB multiplier is generated, it is ready to be verified using similar approach appear in section **??**. The following part borrows contents from my own conference paper [27].

5.4.1 Implicit Unrolling based on Abstraction with ATO

In preliminaries we talk about the abstraction basics. If we use elimination term order $intermediate\ variables > R > A, B$, the function of the combinational logic component can be abstracted as

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

If we are verifying the functional correctness of a combinational NB multiplier (e.g. Mastrovito multiplier or Montgomery multiplier), the function given by abstraction will be $R = AB$. While in the sequential case, the function of combinational logic only fulfills a part of the multiplication, such as F_{k-1} in equation 5.10. Nevertheless, the abstraction still provides a word-level representation which works definitely better on 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 7: Abstraction via implicit unrolling for Sequential GF circuit verification

Input: Circuit polynomial ideal J, vanishing ideal J_0, initial state ideal $R(=0), \mathcal{G}(A_{init}), \mathcal{H}(B_{init})$

$from_0(R, A, B) = \langle R, \mathcal{G}(A_{init}), \mathcal{H}(B_{init})\rangle$;
$i = 0$;
repeat
  $i \leftarrow i + 1$;
  $G \leftarrow$GB($\langle J + J_0 + from_{i-1}(R, A, B)\rangle$) with ATO;
  $to_i(R', A', B') \leftarrow G \cap \mathbb{F}_{2^k}[R', A', B', R, A, B]$;
  $from_i \leftarrow to_i(\{R, A, B\} \setminus \{R', A', B'\})$;
until $i == k$;
return $from_k(R_{final})$

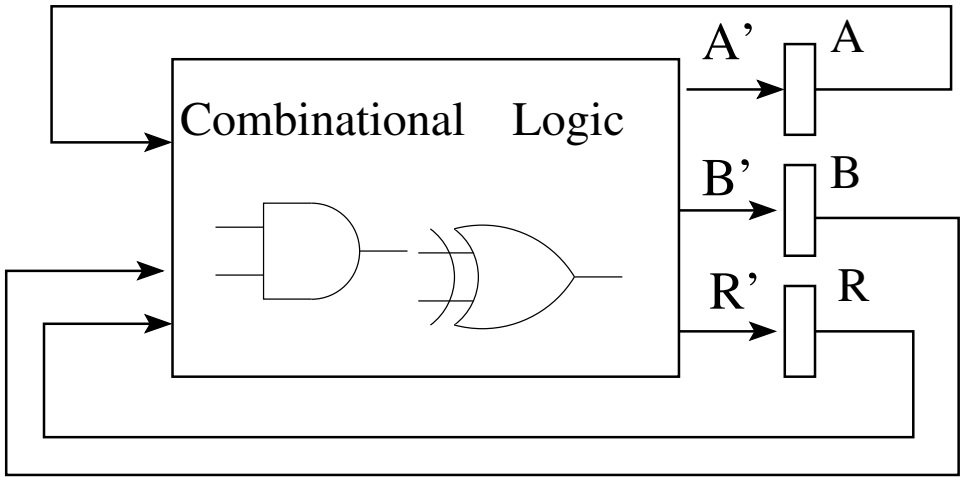


Figure 5.7: A typical normal basis GF sequential circuit model. $A = (a_0, \ldots, a_{k-1})$ and similarly B, R are k-bit registers; A', B', R' denote next-state inputs.

We follow the sequential GF circuit model of figure 5.7, with word-level variables A, B, R denoting *present states (PS)* and A', B', R' denoting *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 $R' = \mathbb{F}(A_{init}, B_{init})$.

Algorithm 7 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 FSM reachability.

Example 5.7 (Functional verification of 5-bit RH-SMPO) *Figure 5.5 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 7 is the ideal for all*

gates in combinational logic block and definition of word-level variables.

$$\begin{aligned} J_{ckt} =& d_0 + a_4b_4, c_1 + a_0 + a_4, c_2 + b_0 + b_4, d_1 + c_1c_2, c_3 + a_1a_4, \\ & c_4 + b_1b_4, d_2 + c_3c_4, e_0 + d_0 + d_1, e_3 + d_1 + d_2, e_4 + d_2, \\ & R_0 + r_4 + e_0, R_1 + r_0, R_2 + r_1, R_3 + r_2 + e_3, R_4 + r_3 + e_4, \\ & A + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}, \\ & B + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18}, \\ & R' + r'_0\alpha^5 + r'_1\alpha^{10} + r'_2\alpha^{20} + r'_3\alpha^9 + r'_4\alpha^{18}, \\ & R + R_0\alpha^5 + R_1\alpha^{10} + R_2\alpha^{20} + R_3\alpha^9 + R_4\alpha^{18}; \end{aligned}$$

In our implementation here, since we only focus on the output variable R, evaluations of intermediate input operands A, B are unnecessary. Polynomials about A and B can be removed from J_{ckt}, and R is directly evaluated by initial operands A_{init} and B_{init}, which are associated with present state bit-level inputs $a_0, a_1, \ldots, a_4$ and $b_0, b_1, \ldots, b_4$ by polynomials in $from^i$.

According to line 5 of algorithm 7, we merge J_{ckt}, J_0 and $from^i$, then compute its Gröbner basis with abstraction term order (copy details here). 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 7 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 1:* $\mathbf{from^0} = \{R, A_{init} + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}, B_{init} + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18}\}$,
  $\mathbf{to^1} = \{R' + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init} + (\alpha^4+\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{8} + (\alpha^2)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha^2+\alpha+$

$1)A^2_{init}B^4_{init}+(\alpha^3+\alpha^2+\alpha)A^2_{init}B^2_{init}+(\alpha^4+\alpha)A^2_{init}B_{init}+(\alpha^4+\alpha^3+1)A_{init}B^{16}_{init}+(\alpha^4+\alpha^2)A_{init}B^8_{init}+(\alpha^4+\alpha^3+\alpha+1)A_{init}B^4_{init}+(\alpha^4+\alpha)A_{init}B^2_{init}+(\alpha^3+\alpha+1)A_{init}B_{init}, A_{init}+a'_4\alpha^5+a'_0\alpha^{10}+a'_1\alpha^{20}+a'_2\alpha^9+a'_3\alpha^{18}, B_{init}+b'_4\alpha^5+b'_0\alpha^{10}+b'_1\alpha^{20}+b'_2\alpha^9+b'_3\alpha^{18}\}$

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

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

$\mathbf{to^3} = \{R' + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha)A_{init}^{16}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init}^{4} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{16}B_{init} + (\alpha)A_{init}^{8}B_{init}^{16} + (\alpha+1)A_{init}^{8}B_{init}^{8} + (\alpha^4)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{4}B_{init}^{16} + (\alpha^4)A_{init}^{4}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha^2+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}B_{init}^{4} + (\alpha^4+\alpha)A_{init}B_{init}, A_{init} + a'_2\alpha^5 + a'_3\alpha^{10} + a'_4\alpha^{20} + a'_0\alpha^9 + a'_1\alpha^{18}, B_{init} + b'_2\alpha^5 + b'_3\alpha^{10} + b'_4\alpha^{20} + b'_0\alpha^9 + b'_1\alpha^{18}\}$

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

- *Clock 5:* $\mathbf{from^4} = \{R + (\alpha^3+\alpha+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{16}B_{init}^{8} + (\alpha^4+\alpha)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^3+\alpha+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+\alpha+1)A_{init}^{8}B_{init}^{16} + (\alpha^3+1)A_{init}^{8}B_{init}^{8} + (\alpha^4+\alpha^2+\alpha)A_{init}^{8}B_{init}^{4} + (\alpha^2+\alpha)A_{init}^{8}B_{init}^{2} +$

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

The final result is $from^5(R_{final}) = R_{final} + A_{init} \cdot B_{init}$

5.4.2 Overcome Computational Complexity using RATO

Similar to our improvements in section **??**, RATO [28] is also available to accelerate the GB computation here. More specifically, we obviate GB computation by turning it into a single-step multivariate polynomial division: first, in ideal generated by gates information polynomials and word-level variable definition polynomials, find the unique pair of polynomial generators with leading monomials not relatively prime to each other; then, compute their specification polynomial using definition

$$Spoly(f_w, f_g) = \frac{LCM}{lt(f_w)} \cdot f_w - \frac{LCM}{lt(f_g)} \cdot f_g$$

where LCM is least common multiple of $lm(f_w)$ and $lm(f_g)$, and lt denotes the leading term; last, reduce $Spoly$ with ideal $J_{ckt} + J_0$, it is possible that the remainder will be canonical polynomial function of the circuit. We will illustrate the whole improved procedure by applying RATO on 5-bit RH-SMPO in figure 5.5.

Example 5.8 *Variable order under RATO is:*

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

Search among all generators of J_{ckt} from Ex.5.7 using RATO, we 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 contains both bit-level variables and word-level variables, and the number of remaining 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 contains only word-level variables (such as $R' + \mathcal{F}(A, B)$). One possible solution to this problem is to replace bit-level variable monomials 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-fashion approach is introduced to compute corresponding $\mathcal{G}(A), \mathcal{H}(R)$ efficiently.

Example 5.9 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:*

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

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

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

Use Gaussian elimination on coefficients matrix, for example

$$\begin{aligned} Row\ 2 =& Row\ 1 \times \alpha^5 + Row\ 2 : \\ & a_1 + (\alpha)a_2 + (\alpha^4 + \alpha^2)a_3 + (\alpha^3 + \alpha^2)a_4 \\ =& (\alpha^4 + \alpha^3 + \alpha^2 + 1)A^2 + (\alpha^2 + \alpha)A \end{aligned}$$

Recursively eliminate a_1 from third row, a_2 from fourth row, etc. The final solution to this system of equations is

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

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

Lemma 5.2 *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 $R' + \mathcal{F}(A, B)$.

5.4.3 Solving Linear System for Bit-to-Word Substitution

In example 5.9 we use a Gaussian-elimination-fashion method to solve the system of polynomial equations. There is another formalized method to solve following system of equations

$$\begin{bmatrix} S \\ S^2 \\ S^{2^2} \\ \vdots \\ S^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} \beta & \beta^2 & \beta^{2^2} & \cdots & \beta^{2^{k-1}} \\ \beta^2 & \beta^{2^2} & \beta^{2^3} & \cdots & \beta \\ \beta^{2^2} & \beta^{2^3} & \beta^{2^4} & \cdots & \beta^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \beta^{2^{k-1}} & \beta & \beta^2 & \cdots & \beta^{2^{k-2}} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{k-1} \end{bmatrix} \tag{5.14}$$

Let $\mathbf{s}$ be a vector of k unknowns $s_0, \ldots, s_{k-1}$, then equation 5.14 can be solved by using Cramer's rule:

$$s_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|}, \;\; 0 \leq i \leq k-1, |\mathbf{M}| \neq 0 \tag{5.15}$$

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 belongs to a special sort of determinants:

Definition 5.2 *Let* $\{\alpha_0, \alpha_1, \ldots, \alpha_{k-1}\}$ *be a set of* k *elements of* $\mathbb{F}_{p^k}$*. Then the determinant*

$$\det M(\alpha_0, \ldots, \alpha_{k-1}) = \begin{vmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{k-1} \\ \alpha_0^p & \alpha_1^p & \cdots & \alpha_{k-1}^p \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{p^{k-1}} & \alpha_1^{p^{k-1}} & \cdots & \alpha_{k-1}^{p^{k-1}} \end{vmatrix} \tag{5.16}$$

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

Moore determinant can be written as an explicit expression

$$\det M(\alpha_0, \ldots, \alpha_{k-1}) = \alpha_0 \prod_{i=1}^{k-1} \prod_{c_0,\ldots,c_{i-1} \in \mathbb{F}_p} (\alpha_i - \sum_{j=0}^{i-1} c_j \alpha_j) \tag{5.17}$$

We use an example to help understanding the notations in Eqn.5.17:

Example 5.10 *Let* $\{\alpha_0, \alpha_1, \alpha_2\}$ *be a set of elements of* $\mathbb{F}_{2^3}$*. Then*

$$\begin{aligned} \det M(\alpha_0, \alpha_1, \alpha_2) &= \begin{vmatrix} \alpha_0 & \alpha_1 & \alpha_2 \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 \\ \alpha_0^4 & \alpha_1^4 & \alpha_2^4 \end{vmatrix} \\ &= \alpha_0 \prod_{i=1}^{2} \prod_{c_0,\ldots,c_{i-1} \in \mathbb{F}_2} (\alpha_i - \sum_{j=0}^{i-1} c_j \alpha_j) \end{aligned} \tag{5.18}$$

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

$$\det M(\alpha_0, \alpha_1, \alpha_2) = \alpha_0\alpha_1(\alpha_1 - \alpha_0)\alpha_2(\alpha_2 - \alpha_1)(\alpha_2 - \alpha_0)(\alpha_2 - \alpha_0 - \alpha_1) \tag{5.19}$$

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

$$\det M(\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) = 1 \tag{5.20}$$

Proof A: ccording to the definition Eqn.5.17, the Moore determinant consists of all possible linear combinations of $\{\alpha_0, \alpha_1, \ldots, \alpha_{k-1}\}$ with coefficients over $\mathbb{F}_p$. If $\{\alpha_0, \alpha_1, \ldots, \alpha_{k-1}\}$ 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 5.3 to Eqn.5.15 gives

$$s_i = |\mathbf{M_i}|, \;\; 0 \leq i \leq k-1 \tag{5.21}$$

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

However, we observe a fact that solution, if written as the inversion of matrix M, is a circulant matrix itself. We have following proposition:

...

Therefore, for a certain type of ONB, we can directly write down the inversion of corresponding matrix M as

...

5.5 Software Implementation of Implicit Unrolling Approach

Our experiment on different size of SMPO designs is performed with both SINGULAR [20] symbolic algebra computation system and our customized toolset deployed 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.

5.5.1 Architecture in Singular

The Singular tool can read in scripts written in its own format similar to ANSI-C. For SMPO experiment, the main loop of our script file performs the same function as algorithm 7 describes, while Gröbner basis computation in main loop can be divided into 4 different function parts:

(i) 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 $a_0 \sim a_{k-1}$:

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

The methodology has been discussed in section **??**. For 5-bit SMPO example, we start from word-level expression 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 exponential cannot exceed 2^{63}, so when doing experiments for SMPO larger than 62 bits, we use a little trick (the feasibility of this trick can also be verified in following steps). Since the BLVS method only requires squaring of equations each time, the exponential 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 exponential presenting in Singular tool, we rewrite 2^{i-1} to 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 exponential will not exceed the Singular data size limit.

This step requires limited substitution operations in Singular, 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. For 33 bits experiment, pre-process execution time is 2.7 sec; while for 100 bits experiment time cost is 36 sec.

(ii) Spoly reduction:

First, Spoly is calculated based on RATO, then reduced with the ideal composed by circuit description polynomials (J). 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):

$$redSpoly = \sum r_i + \sum a_i b_i + \sum a_i B + \sum b_i A + R + r$$

Note there is no cross-term for bit-level or word-level variables from same side 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.

(iii) Substitute bit-level variables in reduced Spoly:

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):

$$R + \sum r^i + \sum A^i B^j$$

all coefficients omitted.

(iv) Substitute present state word-level variable r with inputs A and B:

According to section **??**, there is still a polynomial r_{in} in the ideal we want to compute Gröbner basis. This polynomial has form $r + f'(A, B)$, which is last clock cycle's output ($R + f'(A, B)$) with only leading term replaced in step "$from^i \leftarrow to^i$" in algorithm 7. Basically this step has nothing different from last one, however, it must be taken good care of when using our trick. There is power of r, r^m is originally $r^{2^{m-1}}$; so if $r + f'(A, B)$ contains terms $A^i B^j$, the correct result after doing power is

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

So the correct exponential for A and B in $(A^iB^j)^m$ should be $((i+m-2) \bmod k)+1$ and $((j+m-2) \bmod k)+1$, respectively.

Within one main loop, after finishing steps (ii) to (iv), the output should be intermediate multiplication result $R + f(A, B)$. After k loops, the output is $R + A \cdot B$ when SMPO is bug-free.

5.5.2 Architecture in Customized C++ Toolset

5.6 Experimental Results

We have implemented our approach within the SINGULAR symbolic algebra computation system [v. 3-1-6] [20]. 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™ 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 $\mathbb{F}$, we unroll the SMPO over k time-frames, and construct a miter against a combinational implementation of $\mathbb{F}$. 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* [29] solver. We also experiment with the Combinational Equivalence Checking (CEC) engine of the ABC tool [30], which uses AIG-based reductions to identify internal AIG equivalences within the miter to efficiently solve verification. The BDD-based VIS tool [24] is also used for equivalence check. The run-times for verification of (unrolled) RH-SMPO against Mastrovito *spec* are given in Table 5.2 – which shows that the techniques fail beyond 23 bit fields.

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 5.3, where

Table 5.2: 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*

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 5.3: 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 5.4 and Table 5.5, 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 5.4: 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 5.5: 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

5.7 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 inef-

ficient 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" [**?**] can speed up the procedure dividing a low-degree polynomial with term-sparse polynomial ideal.

CHAPTER 6

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

Besides elimination ideal based abstraction, Gröbner basis can also be applied to a vast field about other techniques in formal verification. Satisfiability (SAT) problem is the basis of modern computation theories, as well as the origin of most formal verification problem. In this chapter, we will discuss a sort of problems branching out from SAT. 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 a smaller subset (core) that keeps UNSAT. From the execution of GB algorithm, an auxiliary structure can be obtained to help UNSAT core extraction. This chapter will introduce the details about the motivation, mechanism and implementation.

6.1 Motivation

6.1.1 Exploiting UNSAT cores for abstraction refinement

By exploring previous work, we learn that most state-of-art abstraction refinement techniques require information mining from UNSAT proofs of intermediate abstractions. Here we use an abstraction refinement algorithm from [31] to explain that how an UNSAT proof is utilized in such techniques.

Algorithm 8: 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 P of k-BMC;
6      M' = ABSTRACT(M, 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
```

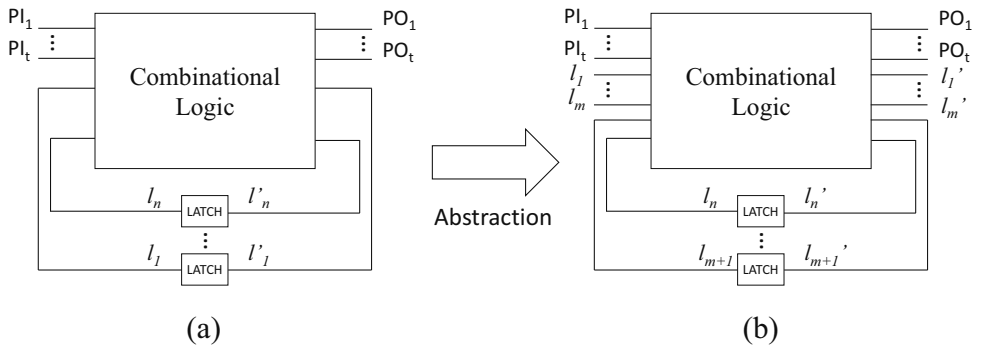


Figure 6.1: Abstraction by reducing latches

Assume that we are given a sequential circuit with n latches as shown in Fig.6.4(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 8 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 8 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 detected. In this algorithm, we analyze the UNSAT core composed by a set of clauses whose conjunction is $UNSAT$. If there are some latch variables not included in this UNSAT core (denoted by L_{abs}), 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 Fig.6.4(b) ($L_{abs} = \{l_1, \ldots, l_m\}$).

The second "if-else" branch means: if we do an ordinary model checking on 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 [31] the UNSAT core is extracted using a conventional CNF-SAT solver, which will encounter the "bit-blasting" problem when the size of datapath (number of latches in Fig.6.4) is very large. **Here we propose a totally 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 based on our experience:**

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

Theorem 6.1 Weak Nullstellensatz: *Given ideal $J \subset \mathbb{F}[x_1, \ldots, x_n]$, its variety over algebraic closure of field $\mathbb{F}$ is empty if and only if its reduced Gröbner basis contains only one generator "1".*

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

It is well known that using Buchberger's algorithm and its variations to compute a GB has a very high time complexity and is usually not practicable. One reason is that the size of 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.

6.1.2 A Demonstration of Motivating Example

One important research topic about UNSAT problems is to identify UNSAT cores efficiently. An UNSAT core in a CNF formula denotes a subset of clauses which is still unsatisfiable. Here we re-define this concept in algebraic geometry:

Definition 6.1 *Assume a set of polynomials F and its subset $F_s \subset F$. If $\mathbf{V}(\langle F \rangle) = \mathbf{V}(\langle F_s \rangle) = \emptyset$, we call F_s an* **UNSAT core** *of F. Additionally, if F_s contains no other UNSAT core, we call it a* **minimal** *UNSAT core.*

We conjecture that based on observation of Buchberger's algorithm's execution, an UNSAT core can be identified.

Proposition 6.1 *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 6.1 *A SAT problem is described with 8 CNF clauses:*

$$c_1 : \bar{a} \vee \bar{b} \qquad c_5 : x \vee y$$
$$c_2 : a \vee \bar{b} \qquad c_6 : y \vee z$$
$$c_3 : \bar{a} \vee b \qquad c_7 : b \vee \neg y$$
$$c_4 : a \vee b \qquad c_8 : a \vee x \vee \neg z$$

Using Boolean to polynomial mappings given in Table **??**, *we can transform them to a set of polynomials F over ring $\mathbb{F}_2[a, b, x, y, z]$:*

$$f_1 : ab \qquad f_5 : xy + y + x + 1$$
$$f_2 : ab + a \qquad f_6 : yz + y + z + 1$$
$$f_3 : ab + b \qquad f_7 : by + y$$
$$f_4 : ab + a + b + 1 \qquad f_8 : axz + az + xz + z$$

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 Fig.6.2(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 Fig.6.2(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 clauses.*

We conclude our approach as a conjecture algorithm (Algorithm 9).

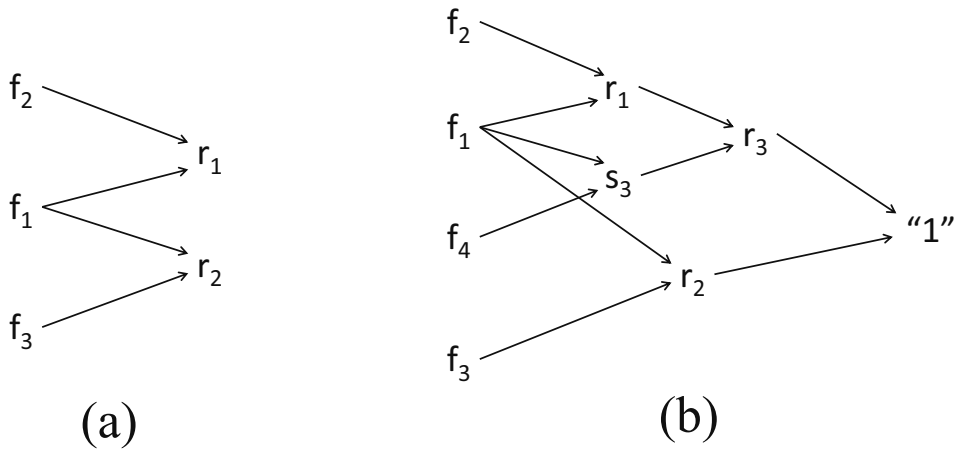


Figure 6.2: DAG representing Spoly computations and multivariate divisions

Algorithm 9: Extract UNSAT core using a variation of Buchberger's algorithm

Input: A set of polynomials $F = \{f_1, f_2, \ldots, f_s\}$
Output: An UNSAT core $\{f_{m_1}, f_{m_2}, \ldots, f_{m_t}\}$
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*

6.2 Formalize the Buchberger's Algorithm based UNSAT Core Identification

Problem: Let $F = \{f_1, \ldots f_s\}$ be a set of multivariate polynomials in the ring $R = \mathbb{F}[x_1, \ldots, x_d]$ that generate ideal $J = \langle f_1, \ldots, f_s \rangle \subset R$. 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.

6.2.1 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 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 6.2 *Consider the following set of polynomials* $F = \{f_1, \ldots, f_9\}$*:*

$f_1 : abc + ab + ac + bc + a + b + c + 1$
$f_2 : b$
$f_3 : ac$
$f_4 : ac + a$
$f_5 : bc + c$
$f_6 : abd + ad + bd + d$
$f_7 : cd$
$f_8 : abd + ab + ad + bd + a + b + d + 1$
$f_9 : abd + ab + bd + b$

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 6.2, 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 Fig. 6.3. 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 refutation tree of Fig. 6.3 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"

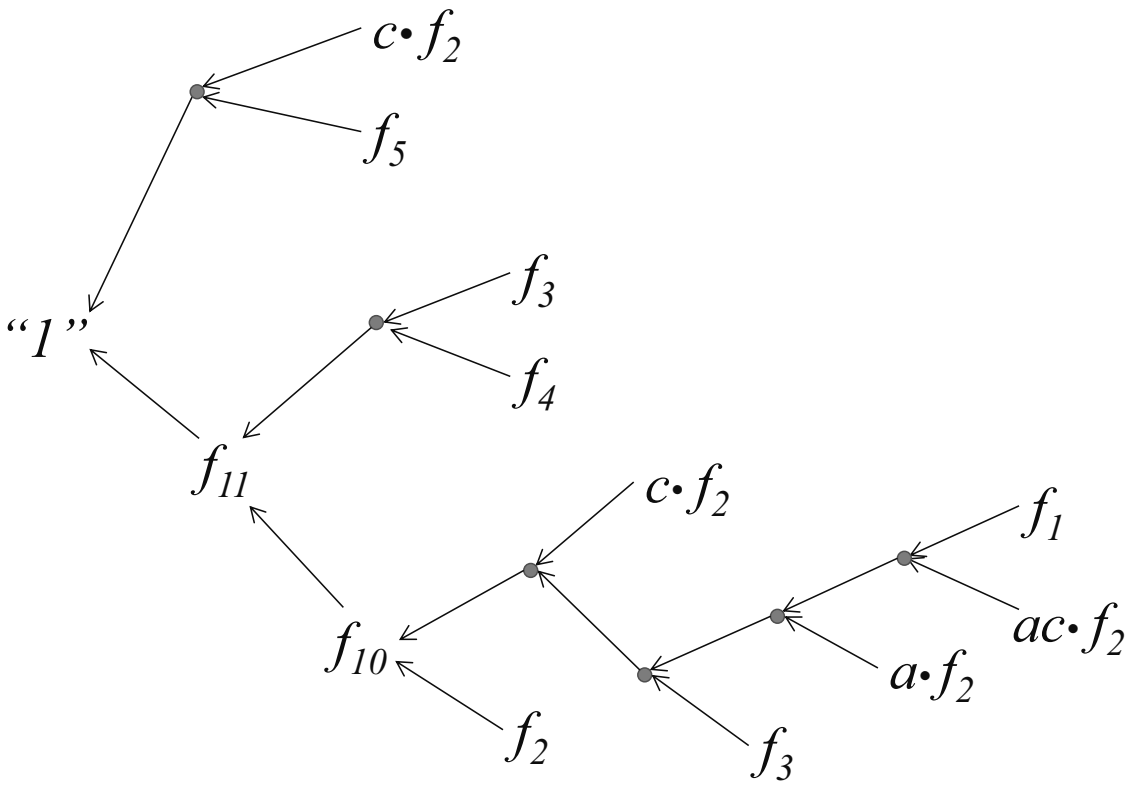


Figure 6.3: Generating refutation trees to record unsat cores

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 Fig. 6.3, $F_c = \{f_1, f_2, f_3, f_4, f_5\}$ is identified as an unsat core of F.

6.3 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 will have to perform a more systematic book-keeping of the data generated during the execution of

Buchberger's algorithm and the refutation tree.

6.3.1 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:

$$g_{ij} = S(f_i, f_j) + \sum_{k=1}^{m} c_k f_k, \tag{6.1}$$

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:

$$((g_{ij})(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \ldots, (c_{kl}, f_{kl})) \tag{6.2}$$

In Eqns. (6.1) and (6.2), 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 Eqn. (6.2), $(c_{k1}, \ldots c_{kl})$ are the respective quotients of division by polynomials $(f_{k1}, \ldots, f_{kl})$, generated during the $Spoly$ reduction.

Example 6.3 *Revisiting Ex. 6.2, and Fig. 6.3, 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 Eqn. (6.2):

$$((f_{10} = g_{12}), (f_1, 1)(f_2, ac)|(a, f_2), (1, f_3), (c, f_2), (1, f_2))$$

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 $G = \{g_1, \ldots, g_t\}$, but also a matrix M of polynomials such that:

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_t \end{bmatrix} = M \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} \tag{6.3}$$

Each element g_i of G is a polynomial combination of $\{f_1, \ldots, f_s\}$. Moreover, this matrix M is constructed precisely using the data that is recorded in the form of Eqn. (6.2). We now give a condition when the matrix M may identify some redundant elements.

Theorem 6.2 *With the notations above, we have that a core for the system of generators $F = \{f_1, \ldots, f_s\}$ 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 $M = [a_1, \ldots, a_s]$ 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 6.4 *Corresponding to Example 6.2 and the refutation tree shown in Fig. 6.3, 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 Fig. 6.3, it follows that:*

$$1 = (f_2 + f_5) + \cdots + \mathbf{1} \cdot \mathbf{f_3} + \cdots + \mathbf{1} \cdot \mathbf{f_3} + \cdots + (f_1 + f_2) \tag{6.4}$$

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.

6.3.2 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:

$$g_{ij} = \underbrace{h_{ij}f_i + h_{ji}f_j}_{Spoly(f_i,f_j)} + \sum_{k=1}^{m} c_k f_k \tag{6.5}$$

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 Eqn. (6.2):

$$((f_{m+1} = g_{ij})(f_i, h_{ij})(f_j, h_{ji})|(c_{k1}, f_{k1}), (c_{k2}, f_{k2}), \ldots, (c_{kl}, f_{kl}))$$

This data forms a part of the refutation tree rooted at node f_{m+1}.

Recording the coefficients: In Eqn. (6.5) 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 Eqn. (6.2). 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 $\{f_1, \ldots, f_s\}$. Algorithm 10 describes this data recording through which the refutation tree T and the matrix M is derived.

Algorithm 10: GB-core algorithm (based on Buchberger's algorithm)

Input: $F = \{f_1, \ldots, f_s\} \in \mathbb{F}[x_1, \ldots, x_d], f_i \neq 0$
Output: Refutation tree T and coefficients matrix M

```
1: Initialize: list G ← F; Dataset D ← ∅; M ← s × s unit matrix
2: for each pair (f_i, f_j) ∈ G do
3:    f_sp, (f_i, h_ij)(f_j, h_ji) ← Spoly(f_i, f_j) {f_sp is the S-polynomial}
4:    g_ij|(c_k1, f_k1), ..., (c_kl, f_kl) ← (f_sp →_{+,G} g_ij)
5:    if g_ij ≠ 0 then
6:       G ← G ∪ g_ij
7:       D ← D ∪ ((g_ij)(f_i, h_ij)(f_j, h_ji)|(c_k1, f_k1), (c_k2, f_k2), ..., (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.

6.4 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 6.2 (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, 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 6.5 *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 Fig.6.4(a).*

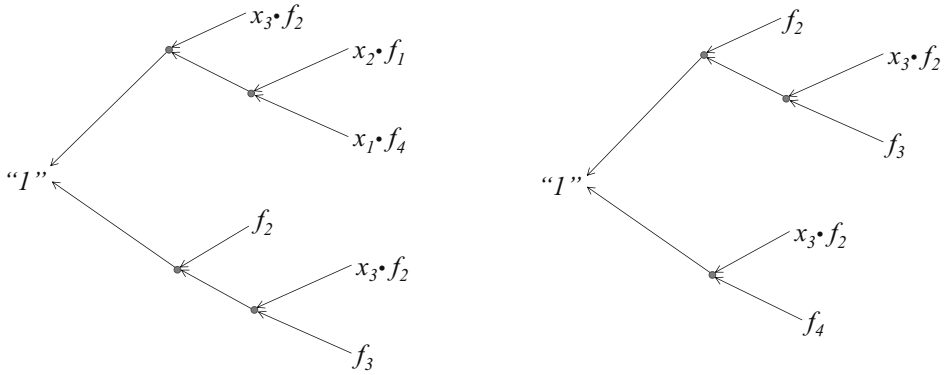


Figure 6.4: Refutation trees of core refinement example

The refutation distance corresponding to polynomial f_2 *is equal to 2 levels. Note that while three leaf nodes in Fig. 6.4 (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 Fig.6.4(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.*

6.5 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 6.3 (Syzygy [19]) *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 Eqn. (6.6), also represented in matrix form in Eqn. (6.7):

$$\begin{cases} c_1^1 f_1 + c_2^1 f_2 + \cdots + c_s^1 f_s = 0 \\ c_1^2 f_1 + c_2^2 f_2 + \cdots + c_s^2 f_s = 0 \\ \quad \vdots \\ c_1^m f_1 + c_2^m f_2 + \cdots + c_s^m f_s = 0 \end{cases} \tag{6.6}$$

$$\begin{bmatrix} c_1^1 & c_2^1 & \cdots & c_s^1 \\ c_1^2 & c_2^2 & \cdots & c_s^2 \\ \vdots & \vdots & \ddots & \vdots \\ c_1^m & c_2^m & \cdots & c_s^m \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} = 0 \tag{6.7}$$

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 Eqn. (6.7) with r_i, and sum up all the rows, we will obtain the following equation:

$$\begin{bmatrix} \sum_{i=1}^m r_i c_1^i & \cdots & 1 & \cdots & \sum_{i=1}^m r_i c_s^i \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \end{bmatrix} = 0 \tag{6.8}$$

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 6.6 *Revisiting Example6.2, 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 $Spoly(f_1, f_3) \xrightarrow{F}_+ 0$.*

$$\begin{array}{c} \\ Spoly(f_1, f_3) \\ Spoly(f_2, f_3) \\ Spoly(f_1, f_4) \\ Spoly(f_2, f_4) \\ Spoly(f_1, f_5) \end{array} \begin{array}{c} \begin{array}{cccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} \end{array} \\ \begin{pmatrix} 1 & a+c+1 & b+1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & c+1 & 1 & b & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & a+c+1 & 0 & 0 & a & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{6.9}$$

Usually, we need to generate extra columns compared to the syzygy matrix of Eqn. (6.7). 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:

$$\begin{array}{c} \\ Spoly(f_1,f_2) \end{array} \begin{array}{c} \begin{array}{cccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} \end{array} \\ \begin{pmatrix} 1 & ac+a+c+1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{6.10}$$

By adding this row vector (Eqn. (6.10)) to the rows in Eqn. (6.9) corresponding to the non-zero entries in the column for f_{10}, we obtain the syzygy matrix only for the polynomials in the core:

$$\begin{array}{c} \\ Spoly(f_1,f_3) \\ Spoly(f_2,f_3) \\ Spoly(f_1,f_4) \\ Spoly(f_2,f_4) \\ Spoly(f_1,f_5) \end{array} \begin{array}{c} \begin{array}{ccccccccc} f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \end{array} \\ \begin{pmatrix} 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ac & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & ac+a & 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & ac & 1 & 0 & a & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

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.

6.6 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] [20]. 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™ i7-4770K Quad-core CPU, 16 GB RAM and 64-bit Ubuntu Linux OS. The experiments are shown in Table 6.1.

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. In order to validate our approach, we use a somewhat customized benchmark library: i) "aim-100" is a modified version of the random 3-SAT benchmark "aim-50/100", modified by adding some redundant clauses; ii) The "subset" series are generated for random subset sum problems; iii) "cocktail" is similarly revised from a combination of factorization and a random 3-SAT benchmark; iv) and "phole4/5" are generated by adding redundant clauses to pigeon hole benchmarks; v) Moreover, SMPO and RH benchmarks correspond to hardware equivalence checking instances of sequential Galois field normal basis modulo multiplier circuits [**?**, 15], compared against a golden model *spec*. Similarly, the "MasVMon" benchmarks are the equivalence checking circuits corresponding to Mastrovito multipliers compared against Montgomery multipliers [32]. 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 [33]), and the GB-Core algorithm and the refinement approach is applied.

Table 6.1: 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

In Table 6.1, #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.

6.7 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 [**?**, **?**, 18, 32].

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Future Work

7.1.1 Multivariate polynomial ideal based FSM Traversal

7.1.2 New Diagram Structure accelerating Polynomial Reduction

7.1.3 Interpolation extraction using GB Algorithm

7.1.4 Verification of Integer Arithmetic Circuits

APPENDIX

A.1 Normal Basis Theory

A.1.1 Characterization of Normal Basis

A.1.2 Construction of General Normal Basis

A.1.3 Bases Conversion

A.2 Optimal Normal Basis

A.2.1 Construction of Optimal Normal Basis

A.2.2 Optimal Normal Basis Multiplier Design

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] 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.

[17] Alex Wee, "Difference between iphone 7 and iphone 6s in a nutshell", *Image*, 2016.

[18] S. Gao, A. Platzer, and E. Clarke, "Quantifier Elimination over Finite Fields with Gröbner Bases", *in Intl. Conf. Algebraic Informatics*, 2011.

[19] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, Springer, 2007.

[20] 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.

[21] UC Berkeley, "Berkeley logic interchange format (blif)", *Oct Tools Distribution*, vol. 2, pp. 197–247, 1992.

[22] Robert Brayton and Alan Mishchenko, "Abc: An academic industrial-strength verification tool", *in Computer Aided Verification*, pp. 24–40. Springer, 2010.

[23] 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.

[24] 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.

[25] Alper Halbutogullari and Çetin K Koç, "Mastrovito multiplier for general irreducible polynomials", *IEEE Transactions on Computers*, vol. 49, pp. 503–518, 2000.

[26] Huapeng Wu, "Montgomery multiplier and squarer for a class of finite fields", *IEEE Transactions on Computers*, vol. 51, pp. 521–529, 2002.

[27] Xiaojun Sun, Priyank Kalla, Tim Pruss, and Florian Enescu, "Formal verification of sequential galois field arithmetic circuits using algebraic geometry", *in Design Automation and Test in Europe, DATE 2015. Proceedings*. IEEE/ACM, 2015.

[28] 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.

[29] Armin Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013", *Proceedings of SAT Competition 2013; Solver and*, p. 51, 2013.

[30] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool", *in Computer Aided Verification*, vol. 6174, pp. 24–40. Springer, 2010.

[31] Liang Zhang, *Design Verification for Sequential Systems at Various Abstraction Levels*, PhD thesis, Citeseer, 2005.

[32] 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.

[33] 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.