FORMAL VERIFICATION AND ABSTRACTION REFINEMENT ON 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

Abstraction plays an important role in digital design, analysis, and verification, as it allows for the refinement of functions through different levels of conceptualization. This dissertation introduces a new method to compute a symbolic, canonical, word-level abstraction of the function implemented by a combinational logic circuit. This abstraction provides a representation of the function as a *polynomial* $Z = F(A)$ over the Galois field $\mathbb{F}_{2^k}$, expressed over the k-bit input to the circuit, A. This representation is easily utilized for formal verification (equivalence checking) of combinational circuits.

The approach to abstraction is based upon concepts from commutative algebra and algebraic geometry, notably the Gröbner basis theory. It is shown that the polynomial $F(A)$ can be derived *by computing a Gröbner basis of the polynomials* corresponding to the circuit, using a specific elimination term order based on the circuits topology. However, computing Gröbner bases using elimination term orders is infeasible for large circuits. To overcome these limitations, this work introduces *an efficient symbolic computation* to derive the word-level polynomial. The presented algorithms exploit i) the structure of the circuit, ii) the properties of Gröbner bases, iii) characteristics of Galois fields $\mathbb{F}_{2^k}$, and iv) modern algorithms from symbolic computation.

While the concept is applicable to any arbitrary combinational logic circuit, it is particularly powerful in verification and equivalence checking of hierarchical, custom-designed and structurally dissimilar Galois field arithmetic circuits. In most applications, the field size and the data-path size k in the circuits is very large, up to 1024 bits.

The proposed abstraction procedure can exploit the hierarchy of the given Galois field arithmetic circuits. A custom abstraction tool is designed to efficiently implement the abstraction procedure. Preliminary experiments show that, using the proposed approach, our tool can abstract and verify Galois field arithmetic circuits *up to* 1024 *bits in size*. Contemporary techniques fail to verify these types of circuits beyond 163 bits and cannot abstract a canonical representation beyond 32 bits.

CHAPTER 1

INTRODUCTION

There is an ever-increasing need for secure communication within information technology. Security of sensitive information relies more and more heavily on encryption methodologies implemented in hardware by cryptographic circuits. One of the most prominent of these methodologies is Elliptical Curve Cryptography (*ECC*), which provides more strength per encryption bit than other encryption methodologies. The main building blocks of ECC hardware implementations are fast, *custom-built Galois field arithmetic circuits*. These circuits are notoriously hard to verify, yet their correctness is vitally important in critical applications. In [1], for example, it is shown that a bug in the hardware could lead to the full leakage of the secret cryptographic key, which could compromise the entire system. Thus, formal verification is imperative in Electronic Design Automation (*EDA*) when dealing with cryptographic circuits.

To facilitate this verification, it is highly desirable to obtain a *word-level representation of the datapath of the ECC arithmetic block* from its bit-level implementation. Ideally, this abstraction should be *canonical*, as this allows the it to be directly applicable to equivalence checking. Such a canonical, word-level abstraction of the Galois field arithmetic block would not only make it easier to verify and reason about the cryptographic system as a whole, but also enable the use of higher level abstraction and synthesis tools. As arithmetic circuits are custom-designed, often modularly, using Galois field arithmetic blocks, the abstraction should also exploit the hierarchical nature of the circuitry. Due to the modular circuit structure, abstraction of each arithmetic block becomes the key in verification of the full circuit. Practical applications of ECC dictate a datapath of a minimum of 163-bits, up to 571-bits, as designated by the National Institute for Standards and Technology (NIST). However abstraction of Galois field arithmetic

circuits has been infeasible for data-paths beyond 16 bits.

This dissertation proposes an algebraic geometry based approach to abstract canonical, word-level representations of bit-level Galois field arithmetic circuits. The approach is able to abstract representations for circuits up to 571 bits in size, which is the largest NIST standard for datapath size in ECC. Verification of circuits for which this abstraction has been computed is shown to be trivial; thus, the focus is on deriving the abstraction quickly and efficiently.

1.1 Hardware Design and Verification Overview

The typical design flow of a hardware system, as shown in Fig. 1.1, starts with a hardware system specification, which describes the necessary functions and parameters that the system must perform and adhere to. The specification is typically modelled using a transaction-level model (*TLM*), which describes communication details between large circuit modules. The TLM is then translated into a register-transfer-level (*RTL*) description, which is composed of abstracted, interconnected circuit blocks that compose the entire system. RTL is typically implemented in hardware description languages (*HDL*) such as Verilog and VHDL, which are the most popular choices in the industry. Next, the RTL is optimized and converted into a *netlist*, i.e. a large collection of small physical blocks (MOSFET, Boolean logic gates, etc.) and the inter-connections (wires) between them. Lastly, the netlist is further optimized and then mapped onto a physical space on a chip, which is then sent off for fabrication. This entire design flow is automated by Computer-Aided Design (*CAD*) tools.

When moving from one abstraction level of the hardware design process to the next, an important issue arises: how can one ensure that the functionality of the optimized design matches original spec? Bugs in hardware design which are not caught early can have costly effects later, such as the need for a redesign. Bugs in arithmetic circuits can be especially catastrophic. One infamous example is the 1994 floating point division (FDIV) bug that affected the Intel Pentium chip [2], and subsequently cost the company $475 million because it was discovered after the chip's release. In another more fatal case, during the Gulf war, an American Patriot Missile battery failed to intercept an

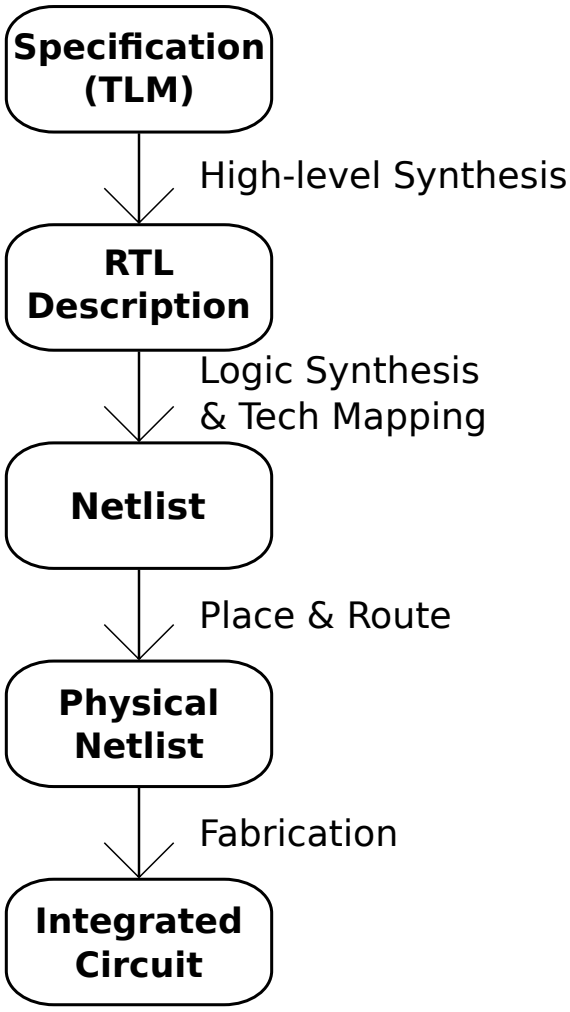


Figure 1.1: Typical hardware design flow.

incoming enemy missile due to an arithmetic error [3]. Since hardware bugs can have significant consequences, there has been extensive work in field of hardware verification to find and eliminate bugs prior to fabrication.

The two main methodologies used in hardware verification are simulation and formal verification. *Simulation* checks correctness by applying exhaustive assignments to the circuit inputs and verifying correctness of the output. This ensures that the circuit performs as designed under all possible inputs. Such exhaustive testing is quite effective for smaller circuits. However, as the size of the circuit increases, it becomes computationally infeasible to simulate all possible test vectors. This is the case with Galois field arithmetic circuits, which are commonly very large in real-world applications. Often for such

large circuits, simulations of a smaller and more manageable subset of test vectors are employed to catch bugs. While these tests can increase confidence in the correctness of the design, *they do not guarantee correctness* since every data-flow of the design hasn't been analyzed.

1.2 Formal Verification

Instead of simulating input vectors, *formal verification* utilizes mathematical theory to reason about the correctness of hardware designs. Formal verification has two main forms: property checking and equivalence checking.

Property checking (or property verification) verifies that a design satisfies certain given properties. Property checking is done mainly in the form of theorem proving, model checking, or approaches which combine the two.

1. *Theorem proving* [4] requires the existence of mathematical descriptors of the specification and implementation of the circuit. Theorem provers apply mathematical rules to these descriptors to derive new properties of the specification. In this way, the tool can reduce a proof goal to simpler sub-goals, which can be automatically verified. However, generating the initial proof-goal requires extensive guidance from the user, so there is an overall lack of automation in theorem proving.

2. *Model checking* [5] is an approach to verifying finite-state systems where specification properties are modeled as a system of logic formulas. The design is then traversed to check if the properties hold. If the design is found to violate a particular property, a counter-example is generated which exercises the incorrect behavior in the design. Such counter-examples allow the designer to trace the behavior and find where the error in the design lies. Modern model checking techniques use the result to automatically refine the system and perform further checking. These tools are typically automated, and thus have found widespread use in CAD tool suites.

Equivalence Checking verifies that two different representations of a circuit design have equivalent functionality. An example of equivalence checking as it applies to the hardware design flow is shown in Fig. 1.2.

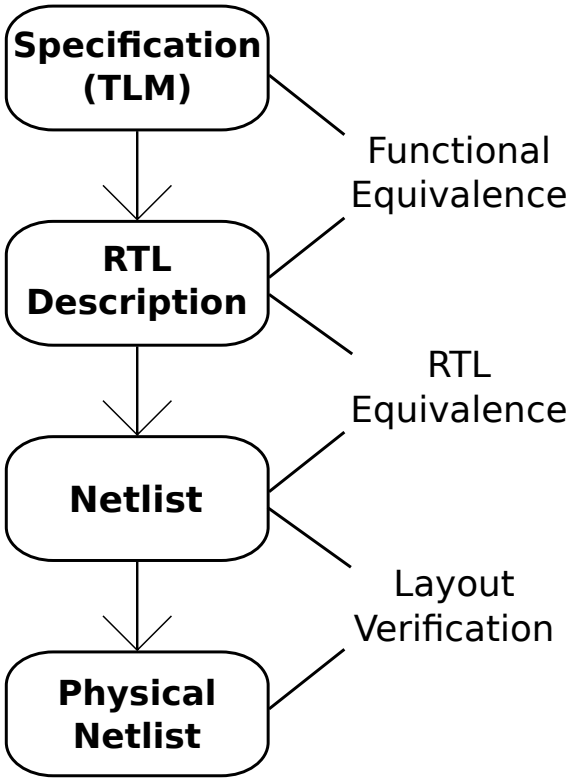


Figure 1.2: Equivalence checking as applied to the hardware design flow.

There are two major equivalence checking techniques: graph-based and satisfiability-based.

1. *Graph-based* techniques construct a canonical graph representation, such as a Binary Decision Diagram (*BDD*) or one of its many variants, of each circuit. A linear comparison is then conducted to determine whether the two graphs are isomorphic. Since the graph representation is canonical, the graphs of the two circuits will be equivalent if and only if the circuits perform the same function.

2. *Satisfiability* techniques construct a miter of the two circuits, typically in a graph such as an And-Inverter graph (*AIG*). A *miter* is a combination of the two circuits with one bit-level output, which is only in a ”1” state when the outputs of the circuits differ given the same given input, as shown in Fig. 1.3. A satisfiability (*SAT*) tool [6] is then employed to simplify the graph and find a solution to the miter, i.e. find an input for which the miter output is ”1”. If a solution is found,

this solution acts as a counter-example of when the circuit outputs differ; otherwise the circuits are functionally equivalent.

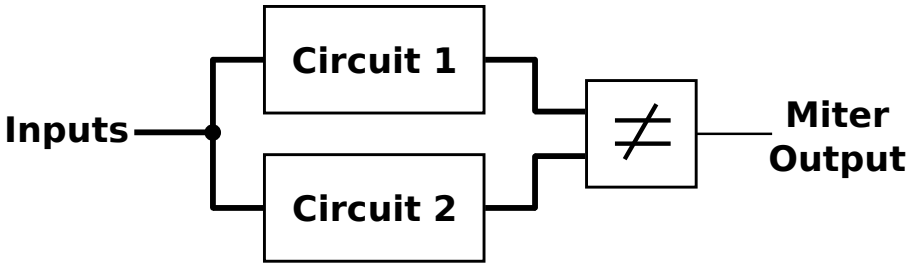


Figure 1.3: A miter of two circuits.

Certain formal verification methods use *computer-algebra* and *algebraic geometry* techniques based on mathematical theories. Unlike SAT-based verification, modern algebraic geometry techniques do not explicitly solve the constraints to find a solution; rather, they reason about the presence or absence of solutions, or explore the geometry of the solutions. These methods [7] [8] [9] transform the circuit design into a polynomial system. Typically, this system of polynomials is then used to compute a Gröbner basis [10]. Computation of Gröbner bases allows for the easy deduction of important properties of a polynomial system, such as the presence or absence of solutions. These properties are then leveraged to perform verification. Unfortunately, such a computation has been shown to be doubly exponential in the worst case, and thus these methods have not been practical for real-world applications. However, recent breakthroughs in computer-algebra hardware verification have shown that it is possible to overcome the complexity of this computation while still utilizing the beneficial properties of a Gröbner bases [11].

1.3 Importance of Word-level Abstraction

Most formal verification techniques can benefit from word-level abstractions of the circuits they verify. Abstraction is defined as state-space reduction, i.e. abstraction reduces state-space by mapping the set of states of a system to a smaller set of states. Because the new representation contains fewer states, it is easier to comprehend and thus

easier to use. Word-level abstraction focuses specifically on abstracting a word-level representation of a circuit out of a bit-level representation. For example, a bit-level representation of an integer multiplier is represented by a collection of Boolean inputs and outputs, whereas a word-level abstraction hides the underlying logic and represents the circuit as two integer inputs and one integer output, e.g. $Z = A \cdot B$. As the bit-size of the multiplier increases, the logical implementation of the multiplier grows (typically exponentially) while the word-level abstraction stays the same.

Word-level abstractions have a wide variety applications in formal verification. Theorem proving techniques can leverage abstraction as an automatic decision procedure or as a canonical reduction engine. For example, since RTL is composed of circuit blocks that represent the underlying circuit, RTL verification methods can exploit abstractions of these blocks. This is seen in the following RTL verification methods:

- Model checking [12], where an approximation abstraction of RTL blocks is generated and then refined.
- Graph-based equivalence checking [13] [14], where abstraction methods are used to generate a canonical word-level graph representation of the circuit.
- Satisfiability-based equivalence checking [15], where abstractions are used identify symmetrics and similarities in order to minimize the amount of logic that is sent to the SAT tool.

Other equivalence checking techniques that employ abstractions include satisfiability modulo theory (*SMT*) techniques [16] [17], which are similar to SAT except they operate on higher-level data structures (integers, reals, bit vectors, etc.), as well as constraint solving techniques [18] [19]. In general, RTL equivalence checking approaches would ideally maintain a high-level of abstraction while still retaining sufficient lower-level functional details (such as bit-vector size, precision, etc) [20].

Word-level hardware abstractions also have applications in RTL and datapath synthesis [21] [22] [23]. Abstractions of circuits allow for design reuse, which allows for tool-automated synthesis of larger circuit blocks. Since hardware design specifications tend to be word-level, synthesis tools can use these larger circuit blocks to generate and

optimize the datapaths and create the RTL of the system. Thus, in order for a circuit to be used by these automated synthesis tools, its word-level abstraction must be known.

Finally, abstractions can also be applied to detect malicious modifications to a circuit, potentially inserted as a hardware trojan horse. Hardware trojans, a relatively new security concern in the hardware industry, use certain techniques to add incorrect behavior to a design. This behavior is only activated under certain rare circumstances that only the mal-intent designer has knowledge of. The behavior is purposely hidden and is very difficult to encounter during simulation of the design. A manufactured chip with a subsystem that contains a hardware trojan could compromise the entire system in which it is used. In some hardware trojan cases, formal verification techniques may be applied to catch a bug in a design and provide a counter-example which exercises it. However, it can be difficult to tell whether the bug in the design was introduced intentionally of not. On the other hand, word-level abstractions of bit-level circuits *effectively reverse-engineer the true function implemented by the circuit*, which could be used to determine the designer's true intention.

1.4 Dissertation Objective, Motivation, and Contributions

This dissertation focuses on abstracting a canonical, word-level representation of hardware (bit-level) implementations of combinational circuits. The proposed technique is a full abstraction solution which can be applied to any arbitrary acyclic combinational circuit. It is particularly efficient when applied to Galois field arithmetic circuits. Using this technique, if the abstraction of the circuit's implementation and its specification are found, they can be easily compared to determine equivalence. Implementation of a custom software tool, developed to compute the abstractions, is also described.

1.4.1 Motivating Application

The motivation for this work comes from applications of Galois field arithmetic circuits in elliptical curve cryptography (*ECC*) hardware systems. The main operations of encryption, decryption, and authentication in ECC rely on operations performed on elliptic curves, which are implemented in hardware as polynomial functions over Galois fields. To be applicable in real-world situations, ECC data-paths should be a minimum

of 163-bits wide, which is the minimum NIST standard, up to a recommended size of 571-bit operand widths. Many non-ECC cryptosystems have datapaths on the order of 1000-bits.

A Galois field arithmetic circuit with a datapath size of k is built as a Boolean function: $\mathbb{B}^k \rightarrow \mathbb{B}^k$. This function is mapped to an operation $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$ over the Galois field $\mathbb{F}_{2^k}$. These circuits are custom-built, modular systems which cannot be synthesized due to their complex nature. Thus, formal verification is needed to ensure they operate correctly.

Recent computer-algebra based formal verification techniques have been able to perform verification of Galois field arithmetic circuits with a datapath size up to 163-bits [11]. Word-level abstractions of Galois field arithmetic circuits could be used to further improve these formal verification techniques to allow for verification of larger circuits, as well as provide the other benefits of word-level abstraction. However, there is currently no technique for computing word-level abstractions of Galois field circuits of any practical size.

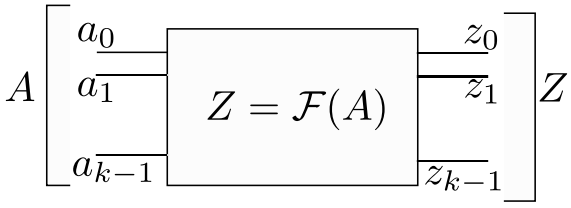


Figure 1.4: Circuit with k-bit input A and k-bit output Z. Abstraction to be derived as $Z = \mathcal{F}(A)$.

While the motivation comes from the need to verify Galois field arithmetic circuits, the presented approach can be generalized to be applicable to any combinational acyclic circuit. Any such circuit with a k-bit input A and a k-bit output Z, such as the one shown in Fig. 1.4, computes $f : \mathbb{B}^k \rightarrow \mathbb{B}^k$ and can thus be analyzed as the function $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$. Over $\mathbb{F}_{2^k}$ this function can be represented as the polynomial $Z = \mathcal{F}(A)$. This is trivially generalized when there are multiple k-bit inputs $A_1, A_2, \ldots, A_i$, i.e. $Z = \mathcal{F}(A_1, \ldots, A_i)$. Now assume the word-size of the input differs from the output, that is the circuit computes $f : \mathbb{B}^m \rightarrow \mathbb{B}^n$ for $m \neq n$. This can be represented as a function

over Galois fields as $f : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^n}$. This function can be analyzed over the field $\mathbb{F}_{2^k}$ such that $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$ and $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$, where $k = LCM(m, n)$.

1.4.2 Dissertation Contributions

To solve the problem of word-level abstraction, this dissertation proposes a full solution consisting of three main contributions.

1. A theory for finding the word-level abstraction from a bit-level circuit over Galois fields is created. The given bit-level circuit implementation is modelled as a system of polynomials over the field. This theory is derived using techniques from computer-algebra, notably the theory of Gröbner basis [24].

2. Using this theory, new algorithms based on symbolic computation are developed to derive the word-level abstraction. The algorithms are designed to be applicable to industry-size arithmetic circuits over Galois fields [25] [26]. A complexity analysis of the algorithmic approach is also presented. Furthermore, the approach is also generalized to make it applicable to arbitrary combinational circuits. Finally, we show how the approach can be used to exploit the hierarchical structure of large Galois field multipliers designed over composite fields.

3. A custom software tool implementation of the algorithmic approach is described, including an analysis of efficient data structures designed for this purpose [27].

Experiments show that the proposed solution can abstract canonical, word-level, polynomial representations of Galois field arithmetic circuits up to 1024-bits in size, while other contemporary approaches are infeasible beyond a 32-bit designs.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews previous applicable work and highlights their drawbacks with respect to the canonical, word-level abstraction problem. Chapter 3 describes the properties of Galois fields, $\mathbb{F}_{2^k}$, and explains the process of constructing them. It also describes how to design arithmetic circuits over such fields, their complexities, and the role of these circuits in Elliptic

Curve Cryptography. Chapter 4 provides a theoretical background of computer-algebra and Gröbner bases and explains their application to Galois fields. Chapter **??** describes an approach to abstract word-level polynomial representations of combinational circuits using a Gröbner basis computation. Chapter **??** improves on this word-level abstraction approach to make it applicable to much larger circuits. Chapter **??** generalizes the abstraction approach to make it applicable to circuits with varying operand word-lengths. It also describes how the approach can take advantage of the hierarchy of arithmetic circuits designed over composite fields. Chapter **??** describes the implementation details of a custom abstraction tool and gives experimental results of abstracting large Galois field multiplier circuits. Chapter 7 concludes the dissertation and outlines potential future research for continuation of this work.

CHAPTER 2

PREVIOUS WORK

This chapter covers previous work in the area of canonical representations of functions, word-level abstractions and their application to design verification. Since the application of our approach is targeted towards formal equivalence verification, modern combinational equivalence checking techniques are also reviewed. Finally, formal verification techniques using computer algebra, algebraic geometry, and polynomial interpolation are also considered.

2.1 Canonical Decision Diagrams

Canonical representations of Boolean functions have been the subject of extensive investigation for logic synthesis and design verification. The Reduced Ordered Binary Decision Diagram (ROBBD) [28] was the first significant contribution in this area. Efficient implementation of ROBDDs as a software package [29] allowed for efficient formal verification of combinational and sequential circuits. ROBDDs represent a Boolean function as an implicit set of points on a canonical directed acyclic graph (DAG). Manipulation of Boolean functions can then be carried out as composition operations on their respective DAGs. The decomposition principle behind BDDs is one of Shannon's expansion, i.e.

$$f(x, y, \dots) = x f_x + x' f_{x'} \tag{2.1}$$

where $f_x = f(x = 1)$ and $f_{x'} = f(x = 0)$ denote the positive and negative co-factors of f w.r.t. x, respectively. Motivated by the success of BDDs, variants of the Shannon's decomposition principle (Davio, Reed-Muller, etc.) were explored to develop other functional decision diagrams. For example, the AND-OR-NOT logic based Shannon's expansion is transformed into an AND-XOR logic based decomposition, termed as the Davio's decomposition:

$$
\begin{aligned}
f(x, y, \dots) &= x f_x + x' f_{x'} && (2.2)\\
&= x f_x \oplus x' f_{x'} && (2.3)\\
&= x f_x \oplus (1 \oplus x) f_{x'} && (2.4)\\
&= f_{x'} \oplus x(f_x \oplus f_{x'}) && (2.5)
\end{aligned}
$$

Decision diagrams based on such decompositions include FDDs [30], ADDs [31], MTBDDs [32], and their hybrid edge-valued counterparts, HDDs [33] and EVBDDs [34]. While these are referred to as *Word-Level Decision Diagrams* [13], the decomposition is still point-wise, binary, w.r.t. each Boolean variable. These representations do not serve the purpose of word-level abstraction from bit-level representations.

Binary Moment Diagrams (BMDs) [35], and its derivatives K*BMDs [36] and *PHDDs [37], depart from the Boolean decomposition and perform the decomposition of a *linear* function based on its two moments. BMDs provide a compact representation for integer arithmetic circuits such as multipliers and squarers. However, these are inapplicable to word-level abstraction of modulo-arithmetic circuits over Galois fields.

Taylor Expansion Diagrams (TEDs) [38] are a word-level canonical representation of a *polynomial expression*, based on the Taylor's series expansion of a polynomial. However, they do not represent a *polynomial function* canonically. For example, $f_1 = 0$ and $f_2 = 2x^2 - 2x \pmod 4$ are two different polynomial representations of the zero function over $\mathbb{Z}_4$; but they are symbolically different polynomials and they have non-isomorphic TED DAGs. While [39] and [40] provide canonical representations of polynomial functions, they do so over finite integer rings $\mathbb{Z}_{2^k}$ and not over Galois fields $\mathbb{F}_{2^k}$.

MODDs [41] [42] are a DAG representation of the characteristic function of a circuit over Galois fields $\mathbb{F}_{2^k}$. MODDs come very close to satisfying our requirements as a canonical word-level representation that can be employed over Galois fields, as it essentially interpolates a polynomial from the characteristic function. However, MODDs do not scale very well for large circuits — this is because every node in the DAG can have up to k children and the normalization operations are very complicated for MODDs.

They also suffer from the size explosion problem during intermediate computations. They are known to be infeasible in representing functions over 32-bit operand words.

2.2 Word-Level Techniques in RTL Synthesis and Verification

Other attempts to derive high-level representations of functions, along with associated decision procedures, can be found in the rich domain of formal model checking [43] [44], theorem proving [14], bit-vector SMT-solvers [16] [45] [46] [47], automated decision procedures for Presburger arithmetic [48] [49], algebraic manipulation techniques [50], or the ones based on term re-writing [51], etc. Polynomial, integer and other non-linear representations have also been researched: Difference Decision Diagrams (DDDs) [52] [53], interval diagrams [54], interval analysis using polynomials [55], etc. Most of these have found application in constraint satisfaction for simulation-based validation: [56] [57] [15] [58] [59] [47]. Among these, [58] [59] [47] have been used to *solve* integer modular arithmetic on linear expressions - a different application from *representing* finite field modulo-arithmetic on polynomials in a canonical form.

2.3 Combinational Equivalence Checking

The verification problem addressed in this dissertation is a manifestation of the combinational equivalence checking (CEC) problem, where the specification (polynomial) and the implementation (circuit) are custom-designed, structurally very dissimilar circuits. To make use of contemporary gate-level CEC tools, we can take the specification circuit ("golden model") and check its equivalence against the implementation circuit. Canonical decision diagrams (BDDs [28] and their word-level variants [13]), And-Invert-Graph (AIG) based reductions [60] [61], circuit-SAT solvers [6], etc., are among the many techniques that can be employed for this CEC. When one circuit is synthesized from the other, this problem can be efficiently solved using AIG-based reductions (e.g. the ABC tool [62]) and circuit-SAT solvers (e.g., CSAT [6]). Synthesized circuits generally contain many sub-circuit equivalences which AIG and CSAT based tools can identify and exploit for verification. However, when the circuits are functionally equivalent but structurally very dissimilar (e.g., Mastrovito [63] versus Montgomery implementations [64] of Galois field circuits), none of the contemporary techniques,

including ABC and CSAT, offer a practical solution. Automatic formal verification of large *custom-designed modulo-arithmetic circuits* largely remains unsolved today.

This verification problem is very hard for SAT solvers and also for quantifier-free bit-vector (QF-BV) theory based SMT-solvers, due to the large circuit size, and the presence of AND-XOR-SHIFT structures. Similarly, the Cryptol tool-set [65] also employs AIG-based reductions (SAT-sweeping) and SAT/SMT-solvers for verification of crypto-protocols. For applications where AIGs/SAT/SMT-techniques fail, the *Cryptol* tool-set also does not deliver. As shown in [11], *none of BDDs, SAT, SMT solvers, nor the ABC tool can prove design equivalence beyond* 16*-bit circuits.*

In [66], the authors present a method for verification of integer multipliers using a data-flow approach. This work abstracts a polynomial function of the given multiplier and then solves the network flow problem using algebraic techniques. However, the abstraction is solely bit-level, and is thus not applicable to deriving a word-level representation of a given design.

2.4 Verification of Galois Field Circuits

Symbolic computer algebra techniques have been employed for formal verification of circuits over $\mathbb{Z}_{2^k}$ and also over Galois fields $\mathbb{F}_{2^k}$. The work of [67] shows how to use Gröbner basis techniques to count the zeros of an ideal J over $\mathbb{F}_q$ (i.e. count $V_{\mathbb{F}_q}(J)$). The authors then follow-up with an approach for *quantifier elimination* over Galois fields $\mathbb{F}_q$ [68]. However, computing a Gröbner basis is computationally expensive. While these works present the proper theory and algorithms, efficiency/improvements to the Gröbner basis computation is not addressed. This is also the case with other general verification techniques using Gröbner bases [7] [9] [69], etc.

In [70] [71] [72], the authors present the BLUEVERI tool from IBM for verification of Galois field circuits for error correcting codes against an algorithmic spec. The implementation consists of a set of (pre-designed and verified) circuit blocks that are interconnected to form the error correcting system. The spec is given as a set of design constraints on a "check file". Their objective is to prove the equivalence of the implementation against this check file. They model the verification instance as a data-flow graph,

represent each sub-circuit block with its known (word-level) polynomial over $\mathbb{F}_q$, and formulate the verification problem using the *Weak Nullstellensatz* — i.e. to check if the *variety* of the algebraic system "*spec* $\neq$ *implementation*" is empty for which they employ a Nullstellensatz formulation. Their main contributions are: i) a "term re-writing" to specify the algorithmic description using polynomials (ideal); and ii) integrating an AIG-style [60] Boolean solver with their word-level decision procedure, with lazy signal computations and Boolean reasoning. For final verification, the polynomial system is given to a computer algebra tool (SINGULAR [73]) to *compute* a reduced Gröbner basis. However, improvements to the core Gröbner basis computational engine are not the subject of their work.

In [11] [74] [75] [76] [77], *Lv et al.* present computer algebra techniques for formal verification of Galois field arithmetic circuits. Given a specification polynomial f, and a circuit C, they formulate the verification problems as an ideal membership test using the Strong Nullstellensatz and Gröbner bases. In [75], the authors show that for any combinational circuit, *there exists a term order* $>_1$ *that renders the set of polynomials of the circuit itself a Gröbner basis — and this term order can be easily derived by performing a topological traversal of the circuit*. By exploiting this term order, verification can be significantly scaled to 163-bit (NIST-specified) cryptography circuits. In contrast to the work of [11], we are not given a specification polynomial. Instead, given the circuit C, we want to derive (extract) the word-level specification f. In our work, we borrow and further build upon the results of [67] [68] [75] [11].

2.5 Verification of Integer Arithmetic Circuits using Gröbner Bases

Symbolic computer algebra techniques have been used for verification of integer arithmetic circuits [78] and also for decision procedures over Galois fields [67]. The paper [78] addresses verification of finite precision integer datapath circuits using the concepts of Gröbner bases over the ring $\mathbb{Z}_{2^k}$. This work models the circuit constraints by way of arithmetic-bit-level (ABL) polynomials ($\{G\}$), and formulates the verification test as an equivalent variety subset problem. This problem is solved by deriving a term order that already makes $\{G\}$ a Gröbner basis, then computing a normal form f of

the specification g w.r.t. $\{G\}$. Circuit correctness is established by testing whether or not f is a vanishing polynomial over $\mathbb{Z}_{2^k}$ [79]. In [80], the authors further show that the vanishing polynomial test can be omitted by formulating the problem directly over $Q := \mathbb{Z}_{2^k}[X]/\langle x^2 - x : x \in X\rangle$. However, in these works, the problem requires that the word-level abstraction of the circuit be known, whereas our approach derives this abstraction polynomial.

2.6 Polynomial Interpolation in Symbolic Computation

The problem of polynomial interpolation is a fundamental problem in symbolic and algebraic computing which finds application in modular algorithms, such as the GCD computation and polynomial factorization. The problem is stated as follows: Given n distinct data points $x_1, \ldots, x_n$, and their evaluations at these points $y_1, \ldots, y_n$, *interpolate* a polynomial $\mathcal{F}(X)$ of degree $n-1$ (or less) such that $\mathcal{F}(x_i) = y_i$ for $1 \leq i \leq n$. Let t be the number of non-zero terms in $\mathcal{F}$ and let T be the total number of possible terms. When $\frac{t}{T} << 1$, the polynomial $\mathcal{F}$ is *sparse*, otherwise it is *dense*. Much of the work in polynomial interpolation addresses sparse interpolation using the "black-box" model (also called the algebraic circuit model) as shown in Fig. 2.1.

Figure 2.1: The black-box or the algebraic circuit representation.

Let $\mathcal{F}$ be a multivariate polynomial in n variables $\{x_1, \ldots, x_n\}$, with t non-zero terms ($0 < t < T$), represented with a black-box B. On input $(x_1, \ldots, x_n)$, the black-box evaluates $y_i = \mathcal{F}(x_1, \ldots, x_n)$. Given also a degree bound d on $\mathcal{F}$, the goal is to interpolate the polynomial $\mathcal{F}$ with a minimum number of *probes* to the black-box. The early work of Zippel [81] and Ben-Or/Tiwari [82] require $O(ndt)$ and $O(T \log n)$ probes, respectively, to the black-box. These bounds have since been improved significantly; the recent algorithm of [83] interpolates with $O(nt)$ probes.

Our problem of polynomial abstractions of Galois field circuits falls into the category of dense interpolation, as we require a polynomial that describes the function at each of the q points of the field $\mathbb{F}_q$. Newton's interpolation technique, with the black-box model, bounds the number of probes by $(d+1)^n$ — which exhibits very high complexity. In the logic synthesis area, the work of [84] investigates dense interpolation. Due to this high-complexity, their approach is feasible only for applications over small fields, *e.g.* computing Reed-Muller forms for multi-valued logic over $\mathbb{F}_2$.

For our problem, we can also employ the black-box model by replacing the black-box (algebraic circuit) by the given circuit C; then every *probe* of the black-box would correspond to a *simulation of the circuit*. However, as we desire a polynomial representation of the entire function over the Galois field, exhaustive simulation would be required, which is infeasible.

2.7 Concluding Remarks

For the problem of word-level, canonical, polynomial abstractions of Galois field arithmetic circuits over $\mathbb{F}_{2^k}$, previous related work is either inapplicable or only applicable to circuits no larger than 32-bits in size. Therefore, we propose a *symbolic approach* to polynomial interpolation from a circuit using the Gröbner basis computation. However, the complexity of a Gröbner basis computation is prohibitively expensive; thus, we propose further improvements to this approach by deriving a smaller subset of computations based on a Gröbner basis analysis. These improvements allow for abstractions of flattened Galois field circuits up to 571-bits, which is the largest NIST standard for ECC, or up to 1024-bits when a hierarchy is given. Furthermore, we propose applications of this approach to allow for formal verification of flattened Galois field circuits up to 1024-bits, where current techniques are only applicable for circuits up to 163-bits.

CHAPTER 3

GALOIS FIELDS PRELIMINARIES AND APPLICATION IN HARDWARE DESIGN

This chapter provides a mathematical background for understanding 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. The material is referred from [85] [86] [87] for Galois field concepts and [63] [88] [64] [89] [90] for hardware design over Galois fields and previous work by *Lv* [11].

3.1 Rings, Fields and Polynomials

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* $a \in \mathbb{R}$, $a \cdot 1 = a = 1 \cdot a$

For the purpose of this dissertation, any time we refer to a **ring**, we are specifically referring to a **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 *The* **modular number system** *with base* n *is a set of positive integers* $Z_n = \{0, 1, \ldots, n-1\}$*, with the two operations* $+$ *and* $\cdot$ *satisfying the properties below:*

$$\begin{aligned}
(a+b) \pmod n &\equiv ((a \pmod n) + (b \pmod n)) \pmod n \\
(a \cdot b) \pmod n &\equiv ((a \pmod n) \cdot (b \pmod n)) \pmod n \\
(-a) \pmod n &\equiv (n-a) \pmod n
\end{aligned}$$

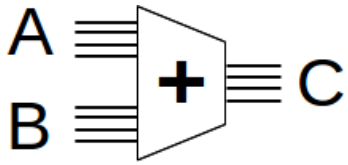
Example 3.1 *The set* $Z_8 = \{0, 1, \ldots, 7\}$ *denotes the modular number system with base* 8*. Examples of some operations performed* $\pmod 8$ *are:*

$$\begin{aligned}
3+6 &= 9 \pmod 8 = 1 \\
3 \cdot 6 &= 18 \pmod 8 = 2 \\
(-3) &= 8-3 \pmod 8 = 5
\end{aligned}$$

The modular number system $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$, where n is a positive integer, forms a ring. Since this type of ring contains a finite number of elements n, it is termed a

finite integer ring, where addition and multiplication are computed *modulo n* $(\mathrm{mod}\ n)$. In hardware applications, arithmetic over k-bit vectors manifests itself as algebra over the finite integer ring $\mathbb{Z}_{2^k}$, where the k- bit vector represents integer values from $\{0, ..., 2^k - 1\}$.

Example 3.2 *Consider the following arithmetic circuit:*



This circuit takes two 4-bit inputs, A and B, and computes a 4-bit sum C. Since A, B, and C are all bit-vectors of size 4, the addition computation this circuit performs is modulo 2^4. Hence, this circuit exemplifies arithmetic computations over the ring Z_{2^4}.

Some examples of possible inputs and outputs of the circuit:

Addition over $\mathbb{Z}_{2^4}$					Boolean Circuit Implementation		
$5+8$	$=$	13	$(\mathrm{mod}\ 16)$	$=13$	$A=0101, B=1000$	$\rightarrow$	$C=1101$
$10+9$	$=$	19	$(\mathrm{mod}\ 16)$	$=3$	$A=1010, B=1001$	$\rightarrow$	$C=0011$
$12+4$	$=$	16	$(\mathrm{mod}\ 16)$	$=0$	$A=1100, B=0100$	$\rightarrow$	$C=0000$

Definition 3.4 *Let $\mathbb{R}$ be a ring. A* **polynomial** *over $\mathbb{R}$ in the indeterminate x is an expression of the form:*

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k = \sum_{i=0}^{k} a_i x^i, \forall a_i \in \mathbb{R}. \tag{3.1}$$

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

Definition 3.5 *The set of all polynomials in the indeterminate x with coefficients in the ring $\mathbb{R}$ forms a* **ring of polynomials** $\mathbb{R}[x]$. *Similarly, $\mathbb{R}[x_1, x_2, \cdots, x_n]$ represents the ring of multivariate polynomials with coefficients in $\mathbb{R}$.*

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

Definition 3.6 *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.

3.2 Galois Fields

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 [85] [86] [87] and hardware arithmetic designs over such fields [63] [88] [64] [89] [90].

Definition 3.7 *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 3.3 *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 3.8 *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:

$$A = \sum_{j=0}^{1} (a_j \cdot i^j) = a_0 + a_1 \cdot i \tag{3.2}$$

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:

$$A = (a_{k-1} a_{k-2} \cdots a_0) \tag{3.3}$$

Example 3.4 *A 4-bit Boolean vector,* $(a_3 a_2 a_1 a_0)$ *can be presented over* $\mathbb{F}_{2^4}$ *as:*

$$a_3 \cdot \alpha^3 + a_2 \cdot \alpha^2 + a_1 \cdot \alpha + a_0 \tag{3.4}$$

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

Example 3.5 *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:* $A = a_3 x^3 + a_2 x^2 + a_1 x + a_0$ *(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*

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

element in $\mathbb{F}_{2^4}$ can then be viewed as a 4-bit vector over $\mathbb{F}_2$. Each element also has an exponential α representation. All three representations are shown in Table 3.1.

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

$$\alpha^4 \pmod{\alpha^4 + \alpha^3 + 1} = -\alpha^3 - 1 = \alpha^3 + 1 \tag{3.5}$$

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

$$\alpha^5 = \alpha^4 \cdot \alpha = (\alpha^3 + 1) \cdot \alpha = \alpha^4 + \alpha = \alpha^3 + \alpha + 1 \tag{3.6}$$

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

An irreducible polynomial can also be a primitive polynomial.

Definition 3.9 *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:*

$$\begin{aligned} A^q &\equiv A \\ A^q - A &\equiv 0 \end{aligned} \tag{3.7}$$

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

Definition 3.10 *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 3.6 *Given $\mathbb{F}_{2^2} = \{0, 1, \alpha, \alpha+1\}$ with $P(x) = x^2 + x + 1$, where $P(\alpha) = 0$.*

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

3.2.1 Containment of Galois Fields

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 3.1 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 $\mathbb{F}_2$.

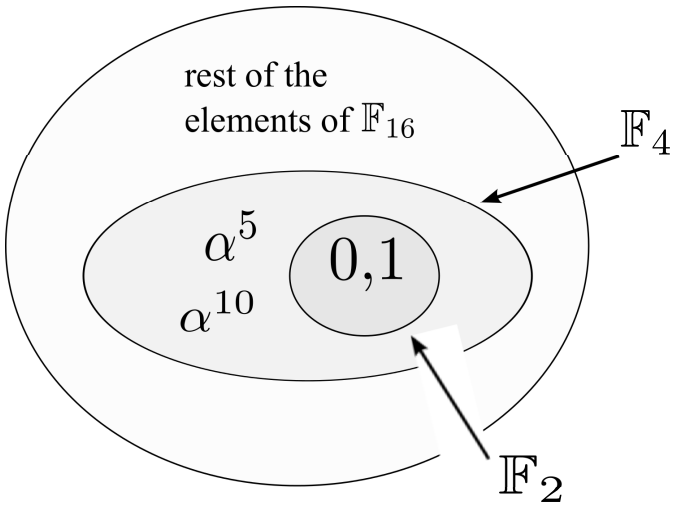


Figure 3.1: Containment of Fields: $\mathbb{F}_2 \subset \mathbb{F}_4 \subset \mathbb{F}_{16}$

Consider the element α^5 of $\mathbb{F}_{16}$. Deriving all $(\alpha^5)^i$ for $i \geq 0$ over $\mathbb{F}_{16}$ gives the following recurrence:

$$\begin{aligned} (\alpha^5)^0 &= 1 \\ (\alpha^5)^1 &= \alpha^5 \\ (\alpha^5)^2 &= \alpha^{10} \\ (\alpha^5)^3 &= \alpha^{15} = 1 \end{aligned} \tag{3.8}$$

The only elements that are generated in this recurrence are $\{1, \alpha^5, \alpha^{10}\}$. Every field contains $\{0, 1\}$, so the elements $\{0, 1, \alpha^5, \alpha^{10}\}$ form $\mathbb{F}_4$. Let $P(x) = x^4 + x^3 + 1$ be the primitive polynomial used to generate $\mathbb{F}_{2^4} = \mathbb{F}_{16}$. A primitive polynomial of degree 2 used to generate $\mathbb{F}_{2^2} = \mathbb{F}_4$ can be found as follows:

$$\begin{aligned} & (x + \alpha^5) \cdot (x + \alpha^{10}) \mod P(x) \\ = \; & x^2 + (\alpha^{10} + \alpha^5)x + \alpha^{15} \mod P(x) \\ = \; & x^2 + x + 1 \end{aligned} \tag{3.9}$$

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

3.2.2 Polynomial Interpolation over Galois Fields

In the construction of digital circuits, arbitrary mappings between two bit-vectors of size k can be constructed. Each such mapping generates a function $f : \mathbb{B}^k \to \mathbb{B}^k$. As every k-bit vector can be construed as an element in $\mathbb{F}_{2^k}$ (as shown in the previous section), every such function also corresponds to a function over a Galois field: $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$.

Definition 3.12 *A function* $f : \mathbb{R} \to \mathbb{R}$ *over a ring* R *is considered a* **polynomial function** *if there exists a polynomial* $\mathcal{F} \in \mathbb{R}[x_1, \ldots, x_d]$ *such that* $\mathcal{F}(x_1, \ldots, x_d) = f(x_1, \ldots, x_d)$*.*

Theorem 3.4 *From [87]: Let* $\mathbb{F}_q$ *be a Galois field of* q *elements where* q *is a power of a prime integer. Given any function* $f : \mathbb{F}_q \to \mathbb{F}_q$*, there exists a polynomial* $\mathcal{F} \in \mathbb{F}_q[x]$ *such that* $f(a) = \mathcal{F}(a)$*, for all* $a \in \mathbb{F}_q$*. Thus, every function* $f : \mathbb{F}_q \to \mathbb{F}_q$ *is a polynomial function.*

Thus, since every function over a Galois field, $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$, is a polynomial function, every mapping between two bit-vectors of size k is a polynomial function over $\mathbb{F}_{2^k}$. Furthermore, every polynomial can be derived using Lagrange interpolation.

Theorem 3.5 *(***Lagrange Interpolation***):*

Given a set of k data points over a function f,

$$(x_0, f(x_0)), \ldots, (x_{k-1}, f(x_{k-1}))$$

where no two $x_i \in \{x_0, \ldots, x_{k-1}\}$ are the same elements, the polynomial representation of f, $\mathcal{F}(x)$, can be interpolated as follows:

$$\mathcal{F}(x) = \sum_{i=0}^{k-1} f(x_i) \cdot L_i(x)$$

$$L_i(x) = \prod_{(0 \le j \le k-1),(j \ne i)} \frac{x - x_j}{x_i - x_j}$$

By applying Lagrange interpolation over every element in the Galois field $\mathbb{F}_{2^k}$, we can derive the polynomial representation $\mathcal{F}$ of any function $f : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$. Furthermore, $\mathcal{F}$ is a polynomial of degree at most $2^k - 1$ in x and $\mathcal{F}(a) = f(a)$ for all $a \in \mathbb{F}_{2^k}$.

While every function over a Galois field is a polynomial function, not every function over the integer ring $\mathbb{Z}$ is a polynomial function.

Example 3.7 *Let $A = \{a_2, a_1, a_0\}$ and $Z = \{z_2, z_1, z_0\}$ be 3-bit vectors. Thus, A and $Z \in \mathbb{B}^3$. Consider the following function:*

$$f : Z[2:0] = A[2:0] >> 1$$

f is a **bit-vector right shift** *operation on A. This function can be analyzed as a mapping over different forms: $\mathbb{B}^3 \to \mathbb{B}^3$, $\mathbb{Z}_8 \to \mathbb{Z}_8$, and $\mathbb{F}_{2^3} \to \mathbb{F}_{2^3}$. These mappings from A to Z are:*

$\{a_2a_1a_0\} \in \mathbb{B}^3$	$A \in \mathbb{Z}_8$	$A \in \mathbb{F}_{2^3}$	$\to$	$\{z_2z_1z_0\} \in \mathbb{B}^3$	$Z \in \mathbb{Z}_8$	$Z \in \mathbb{F}_{2^3}$
000	*0*	*0*	$\to$	*000*	*0*	*0*
001	*1*	*1*	$\to$	*000*	*0*	*0*
010	*2*	α	$\to$	*001*	*1*	*1*
011	*3*	$\alpha + 1$	$\to$	*001*	*1*	*1*
100	*4*	α^2	$\to$	*010*	*2*	α
101	*5*	$\alpha^2 + 1$	$\to$	*010*	*2*	α
110	*6*	$\alpha^2 + \alpha$	$\to$	*011*	*3*	$\alpha + 1$
111	*7*	$\alpha^2 + \alpha + 1$	$\to$	*011*	*3*	$\alpha + 1$

$f : \mathbb{Z}_8 \to \mathbb{Z}_8$ is not a polynomial function (this can be verified using the results of [91] [92] [93]). However, $f : \mathbb{F}_{2^3} \to \mathbb{F}_{2^3}$ is a polynomial function. By applying Lagrange's

interpolation formula to f over $\mathbb{F}_{2^3}$ for every element in $\mathbb{F}_{2^3}$, we obtain the following polynomial function: $Z = (\alpha^2+1)A^4 + (\alpha^2+1)A^2$, where $P(\alpha) = \alpha^3 + \alpha + 1 = 0$.

Since every function over $\mathbb{F}_{2^k}$ is a polynomial function, the functional mapping of a Galois field arithmetic circuit over $\mathbb{F}_{2^k}$ must exist in polynomial form. Construction of these arithmetic circuits is described next.

3.3 Hardware Implementations of Arithmetic Operations Over Galois Fields

There are two main applications of hardware implementations of Galois field arithmetic. In the first case, Galois field arithmetic computations, such as ADD OR MUL, are implemented in hardware, and algorithms are then implemented in software (e.g. cryptoprocessors [94] [95]). In other cases, the entire design can be implemented in hardware, such as a one-shot Reed-Solomon encoder-decoder chip [96] [97], or point multiplication circuitry [98] used in elliptic curve cryptosystems. Therefore, there has been extensive research in efficient hardware design of primitive arithmetic computations over Galois fields. In this section, we describe the design principles of such circuits with focus on their architecture and verification complexity.

Addition in $\mathbb{F}_{2^k}$ is performed by correspondingly adding the polynomials together and reducing the coefficients of the result modulo 2.

Example 3.8 *Given $A = \alpha^3 + \alpha^2 + 1 = (1101)$ and $B = \alpha^2 + 1 = (0101)$ in $\mathbb{F}_{2^4}$,*

$$A + B = (\alpha^3 + \alpha^2 + 1) + (\alpha^2 + 1) = (\alpha^3) + (\alpha^2 + \alpha^2) + (1 + 1) = \alpha^3 = (1000).$$

Effectively, the addition operation is only performed on the coefficients, which are in $\mathbb{F}_2$. As addition over $\mathbb{F}_2$ performs an *XOR* operation, constructing an addition circuit over $\mathbb{F}_{2^k}$ is trivial as it only consists of k number of *XOR* gates. A 4-bit adder over $\mathbb{F}_{2^4}$ is shown in Fig. 3.2.

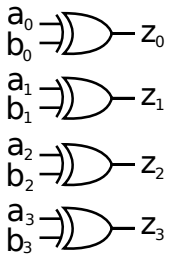


Figure 3.2: 4-bit adder over $\mathbb{F}_{2^4}$.

Multiplication $Z = A \times B \pmod{P(x)}$ in $\mathbb{F}_{2^k}$ conceptually consists of two steps. In the first step, the multiplication $A \times B$ is performed. In the second step, the result is reduced modulo the irreducible polynomial $P(x)$. This multiplication procedure is shown in Example 3.9.

Example 3.9 *Consider the field* $\mathbb{F}_{2^4}$ *with the irreducible polynomial* $P(x) = x^4 + x^3 + 1$ *and* $P(\alpha) = 0$. *We take as inputs:* $A = a_0 + a_1 \cdot \alpha + a_2 \cdot \alpha^2 + a_3 \cdot \alpha^3$ *and* $B = b_0 + b_1 \cdot \alpha + b_2 \cdot \alpha^2 + b_3 \cdot \alpha^3$. *We have to perform the multiplication* $Z = A \times B \pmod{P(x)}$. *The coefficients of* $A = \{a_0, \ldots, a_3\}, B = \{b_0, \ldots, b_3\}$ *are in* $\mathbb{F}_2 = \{0, 1\}$. *This multiplication can be performed as shown:*

			a_3	a_2	a_1	a_0
$\times$			b_3	b_2	b_1	b_0
			$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$
		$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	
	$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$		
$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$			
s_6	s_5	s_4	s_3	s_2	s_1	s_0

The result $Sum = s_0 + s_1 \cdot \alpha + s_2 \cdot \alpha^2 + s_3 \cdot \alpha^3 + s_4 \cdot \alpha^4 + s_5 \cdot \alpha^5 + s_6 \cdot \alpha^6$, *where*

$$
\begin{aligned}
s_0 &= a_0 \cdot b_0 \\
s_1 &= a_0 \cdot b_1 + a_1 \cdot b_0 \\
s_2 &= a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 \\
s_3 &= a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_2 + a_3 \cdot b_1 \\
s_4 &= a_1 \cdot b_3 + a_2 \cdot b_1 + a_3 \cdot b_1 \\
s_5 &= a_2 \cdot b_3 + a_3 \cdot b_2 \\
s_6 &= a_3 \cdot b_3
\end{aligned}
$$

Here the multiply "$\cdot$" and add "$+$" operations are performed modulo 2, so they can be implemented in a circuit using AND and XOR gates respectively. Note that unlike integer multipliers, there are no carry-chains in the design, as the coefficients are always reduced modulo 2*. However, the result is yet to be reduced modulo the primitive polynomial* $P(x) = x^4 + x^3 + 1$*. This transforms every exponent representation,* α^d*, to a polynomial representation where* $d \geq k = 4$*.*

α^3	α^2	α	1	
s_3	s_2	s_1	s_0	
s_4	0	0	s_4	$s_4 \cdot \alpha^4 \pmod{P(\alpha)} = s_4 \cdot (\alpha^3 + 1)$
s_5	0	s_5	s_5	$s_5 \cdot \alpha^5 \pmod{P(\alpha)} = s_5 \cdot (\alpha^3 + \alpha + 1)$
s_6	s_6	s_6	s_6	$s_6 \cdot \alpha^6 \pmod{P(\alpha)} = s_6 \cdot (\alpha^3 + \alpha^2 + \alpha + 1)$
z_3	z_2	z_1	z_0	

The final result (output) of the circuit is: $Z = z_0 + z_1\alpha + z_2\alpha^2 + z_3\alpha^3$*; where* $z_0 = s_0 + s_4 + s_5 + s_6$; $z_1 = s_1 + s_5 + s_6$; $z_2 = s_2 + s_6$; $z_3 = s_3 + s_4 + s_5 + s_6$.

The above multiplier design is called the *Mastrovito multiplier* [63] which is the most straightforward way to design a multiplier over $\mathbb{F}_{2^k}$. A logic circuit for a 4-bit *Mastrovito* multiplier over *Galois field* $\mathbb{F}_{2^4}$ is illustrated in Fig. 3.3.

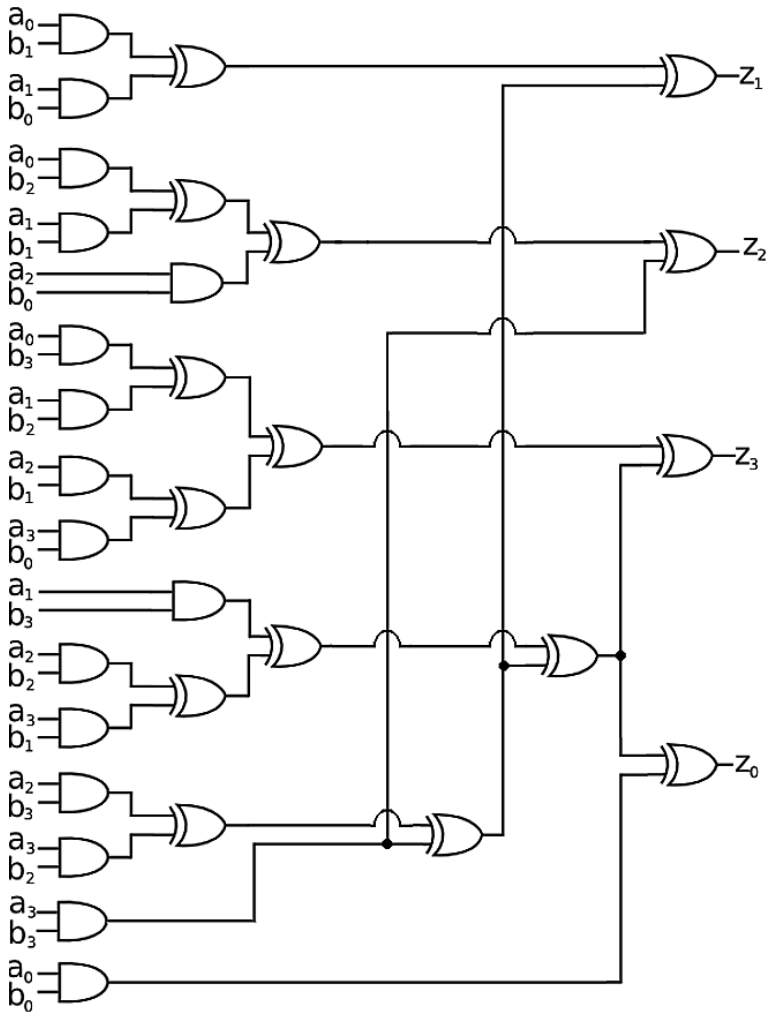


Figure 3.3: Mastrovito multiplier over $\mathbb{F}_{2^4}$.

Modular multiplication is at the heart of many public-key cryptosystems, such as Elliptic Curve Cryptography (ECC) [99]. Due to the very large field size (and hence the data-path width) used in these cryptosystems, the above *Mastrovito* multiplier architecture is inefficient, especially when exponentiation and repeat multiplications are performed. Therefore, efficient hardware and software implementations of modular multiplication algorithms are used to overcome the complexity of such operations. One such algorithm which we will focus on is the Montgomery reduction [88] [64].

3.3.1 Montgomery Multipliers

Montgomery Reduction (MR) computes:

$$G = MR(A, B) = A \cdot B \cdot R^{-1} \pmod{P(x)} \tag{3.10}$$

where A, B are k-bit inputs, $R = \alpha^k$, R^{-1} is multiplicative inverse of R in $\mathbb{F}_{2^k}$, and $P(x)$ is the irreducible polynomial for $\mathbb{F}_{2^k}$. Since Montgomery reduction cannot directly compute $A \cdot B$, we need to pre-compute $A \cdot R$ and $B \cdot R$, as shown in Fig. 3.4.

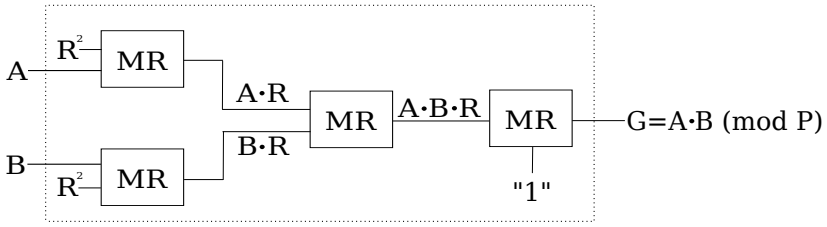


Figure 3.4: Montgomery multiplier over $\mathbb{F}_{2^k}$

Each MR block in Fig. 3.4 represents a Montgomery reduction step which is a hardware implementation of the algorithm shown in Algorithm 1.

Algorithm 1: Montgomery Reduction Algorithm [64]

Input: $A(x), B(x) \in \mathbb{F}_{2^k}$; irreducible polynomial $P(x)$.
Output: $G(x) = A(x) \cdot B(x) \cdot x^{-k} \pmod{P(x)}$.
$G(x)$:=0
for *(*$i = 0$*;* $i \leq k-1$*;* ++*i)* **do**
    $G(x) := G(x) + A_i \cdot B(x)$ /*A_i is the i^{th} bit of A*/;
    $G(x) := G(x) + G_0 \cdot P(x)$ /*G_0 is the lowest bit of G*/;
    $G(x) := G(x)/x$ /*Right shift 1 bit*/;
end

The design of Fig. 3.4 is not efficient to computing $A \cdot B \pmod{P(x)}$ when compared to the Mastrovito implementation. However, when these multiplications are performed repeatedly, such as in iterative squaring, then the Montgomery approach speeds-up the computation. As shown in [89], the critical path delay and gate counts of a

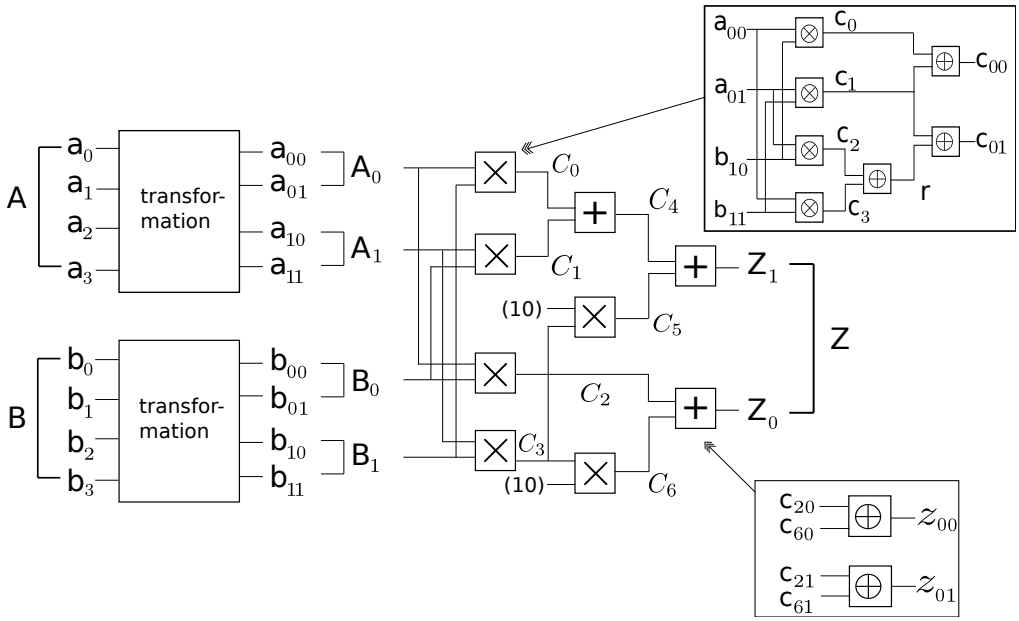


Figure 3.5: 4-bit composite multiplier designed over $\mathbb{F}_{(2^2)^2}$

squarer designed using the Montgomery approach are much smaller than the traditional approaches.

3.3.2 Circuit Designs over Composite Fields

The Galois field $\mathbb{F}_{2^k}$ is a k-dimensional vector space over the sub-field $\mathbb{F}_2$. If $k = m \cdot n$, the field $\mathbb{F}_{2^k}$ can be decomposed as $\mathbb{F}_{(2^m)^n}$. Such a field representation is called a **composite field**, and it is constructed as a n-dimensional extension of the sub-field $\mathbb{F}_{2^m}$. The sub-field $\mathbb{F}_{2^m}$ is called the ground field. Note that we have $\mathbb{F}_2 \subset \mathbb{F}_{2^m} \subset \mathbb{F}_{(2^m)^n}$.

A Galois field arithmetic circuit over $\mathbb{F}_{2^k}$ can thus be composed as circuit over $\mathbb{F}_{(2^m)^n}$ if $k = m \cdot n$. Since the base field is $\mathbb{F}_{2^m}$, this composite field circuit is composed of blocks of m-bit multipliers and adders, along with m-bit buses that act as the inputs and outputs of these blocks. A $\mathbb{F}_{2^4}$ Galois field multiplier designed over the composite field $\mathbb{F}_{(2^2)^2}$ is shown in Fig. 3.5. Design methodologies of these circuits are examined more closely in Chapter **??**.

3.3.3 Applications to Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is one of the most influential applications of Galois fields. ECC is an approach to public-key (or asymmetric-key) cryptography based

on the algebraic structure of elliptic curves over Galois fields. Due to the complex nature of these curves, key sizes in ECC can be smaller than other public-key cryptography techniques while providing the same level of security [100]. The main operations of encryption, decryption and authentication in ECC rely on *point multiplications*.

Point multiplication involves a series of addition and doubling of points on the elliptic curve. A drawback of traditional point multiplication is that each point addition and doubling require a multiplicative inverse operation over Galois fields, the computation of which is costly. Modern methods, however, represent the points in projective coordinate systems [98], which has eliminated the need for a multiplicative inverse operation by replacing it with addition and multiplication operations over Galois fields. This has increased the efficiency of point multiplication operations, but it has also increased the need for fast, custom hardware designs of Galois field arithmetic.

In-depth analysis of elliptic curve theory is beyond the scope of this dissertation. Instead, we will look at some examples of point addition and point doubling to give a general idea of the operations involved in ECC and how they apply to Galois field arithmetic. Our experiments use custom Galois field arithmetic designs based on L*ó*pez-Dahab (LD) coordinate system [101], so these examples will use the same coordinate system.

Example 3.10 *Consider point addition in a LD projective coordinate system, as seen in Fig. 3.6.*

Given an elliptic curve: $Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$ *over* $\mathbb{F}_{2^k}$*, where* X, Y, Z *are* k*-bit vectors that are elements in* $\mathbb{F}_{2^k}$ *and similarly,* a, b *are constants from the field. Let* $P + Q = R$ *represent point addition over the elliptic curve.* $P = (X_1, Y_1, Z_1)$ *and* $Q = (X_2, Y_2, 1)$ *are given. Then* $R = (X_3, Y_3, Z_3)$ *can be computed as follows:*

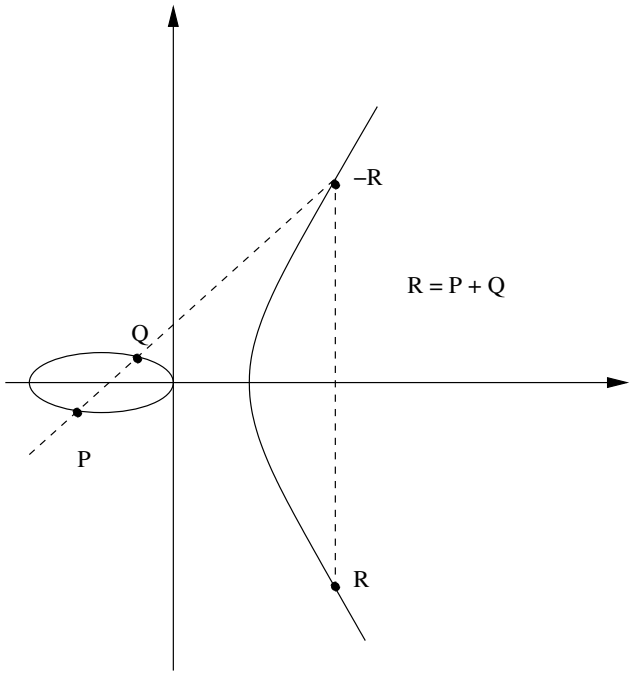


Figure 3.6: Point addition over an Elliptic Curve (R=P+Q)

$$A = Y_2 \cdot Z_1^2 + Y_1$$
$$B = X_2 \cdot Z_1 + X_1$$
$$C = Z_1 \cdot B$$
$$D = B^2 \cdot (C + aZ_1^2)$$
$$Z_3 = C^2$$
$$E = A \cdot C$$
$$X_3 = A^2 + D + E$$
$$F = X_3 + X_2 \cdot Z_3$$
$$G = X_3 + Y_2 \cdot Z_3$$
$$Y_3 = E \cdot F + Z_3 \cdot G$$

Example 3.11 *Consider point doubling in a LD projective coordinate system. Given an elliptic curve:* $Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$. *Let 2(*$X_1$, Y_1, Z_1*) = (*X_3, Y_3, Z_3*),*

then

$$X_3 = X_1^4 + b \cdot Z_1^4$$
$$Z_3 = X_1^2 \cdot Z_1^2$$
$$Y_3 = bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4)$$

In the above examples, polynomial multiplication and squaring operations can be implemented in hardware using Montgomery reductions over Galois fields $\mathbb{F}_{2^k}$. In practical applications, the field size k of $\mathbb{F}_{2^k}$ is 163, or larger. However, there are no word-level abstraction techniques applicable to circuits of such size, so hardware implementations of Galois field arithmetic circuits cannot benefit from the many advantages of abstraction. Thus, we propose a computer-algebra approach to word-level polynomial abstractions of Galois field arithmetic circuits. Recent computer-algebra formal verification techniques [11] have been able to verify these circuits up to 163 bits. We propose an application of our abstraction approach to improve these techniques. These improvements allow us to perform formal verification of these circuits up to 571 bits. These proposals are described in detail in subsequent chapters.

CHAPTER 4

COMPUTER ALGEBRA FUNDAMENTALS

This chapter reviews fundamental concepts of commutative and computer algebra which are used in this work. Specifically, this chapter covers monomial ordering, polynomial ideals and varieties, and the computation of Gröbner bases. It also overviews elimination theory as well as Hilbert's Nullstellensatz theorems and how they apply to Galois fields. The results of these theorems are used in polynomial abstraction and formal verification of Galois field circuits and are discussed in subsequent chapters. The material of this chapter is mostly referred from the textbooks [102] [10] and previous work by *Lv* [11].

4.1 Monomials, Polynomials, and Term Orderings

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

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

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

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

Definition 4.2 *A* **multivariate polynomial** *f in variables $x_1, x_2, \ldots, x_d$ with coefficients in any given field $\mathbb{K}$ is a finite linear combination of monomials with coefficients in $\mathbb{K}$:*

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

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

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

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

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

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

Definition 4.3 *Let $\mathbb{T}^d = \{x^\alpha : \alpha \in \mathbb{Z}^d_{\geq 0}\}$ be the set of all monomials in $x_1, \ldots, x_d$. A* **monomial order** *$>$ on $\mathbb{T}^d$ is a total well-ordering satisfying:*

- *For all $x^\alpha, x^\beta \in \mathbb{T}^d$, x^α and x^β are comparable*
- *For any $x^\alpha \in \mathbb{T}^d$, $x^\alpha > 1$*
- *For all $x^\alpha, x^\beta, x^\gamma \in \mathbb{T}^d$, $x^\alpha > x^\beta \Rightarrow x^\alpha \cdot x^\gamma > x^\beta \cdot x^\gamma$*

Term-orderings are totally ordered, i.e. anti-symmetric with constant terms last in the ordering. A total-order ensures that there is no ambiguity with respect to where a term is found in the term-ordering. Total orderings for monomials come in different forms, notably lexicographic orderings (lex), and its variants: degree-lexicographic ordering (deglex) and reverse degree-lexicographic ordering (degrevlex).

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

Definition 4.4 Lexicographic order: *Let $x_1 > x_2 > \cdots > x_d$ lexicographically. Also let $\alpha = (\alpha_1, \ldots, \alpha_d)$; $\beta = (\beta_1, \ldots, \beta_d) \in \mathbb{Z}^d_{\geq 0}$. Then we have:*

$$x^\alpha > x^\beta \iff \begin{cases} \textit{Starting from the left, the first co-ordinates of } \alpha_i, \beta_i \\ \textit{that are different satisfy } \alpha_i > \beta_i \end{cases} \tag{4.2}$$

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

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

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i & \textit{or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and } x^\alpha > x^\beta & \textit{w.r.t. lex order} \end{cases} \tag{4.3}$$

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

$$x^\alpha > x^\beta \iff \begin{cases} \sum_{i=1}^d \alpha_i > \sum_{i=1}^d \beta_i \textit{ or} \\ \sum_{i=1}^d \alpha_i = \sum_{i=1}^d \beta_i \textit{ and the first co-ordinates} \\ \alpha_i, \beta_i \textit{ from the right, which are different, satisfy } \alpha_i < \beta_i \end{cases} \tag{4.4}$$

Applying these term orderings, we have the following relations, where $a > b > c$.

$$\text{lex:} a^2b > a^2 > abc > ab > ac^2 > ac > b^2c > b^2 > bc^3 > 1 \tag{4.5}$$

$$\text{deglex:} bc^3 > a^2b > abc > ac^2 > b^2c > a^2 > ab > ac > b^2 > 1 \tag{4.6}$$

$$\text{degrevlex:} bc^3 > a^2b > abc > b^2c > ac^2 > a^2 > ab > b^2 > ac > 1 \tag{4.7}$$

The difference between the *lex* and two *deg*- orderings is obvious, while the difference between the two degree-based orderings can be seen by considering from which direction the term is lexed, e.g. $a \cdot c \cdot c > b \cdot b \cdot c$ (deglex, left-to-right) versus $b \cdot b \cdot c > a \cdot c \cdot c$ (degrevlex, right-to-left).

Example 4.1 *Let $f = 2x^2yz + 3xy^3 - 2x^3$. The effects of different term orderings on f are:*

- *lex $x > y > z$: $f = -2x^3 + 2x^2yz + 3xy^3$*
- *deglex $x > y > z$: $f = 2x^2yz + 3xy^3 - 2x^3$*

- *degrevlex* $x > y > z$: $f = 3xy^3 + 2x^2yz - 2x^3$

Definition 4.7 *The* **leading term** *is the first term in a term-ordered polynomial. Likewise, the* **leading coefficient** *is the coefficient of the leading term. Finally, a* **leading monomial** *is the leading term lacking the coefficient. We use the following notation:*

$$lt(f) \quad \text{— } \textit{Leading Term} \tag{4.8}$$

$$lc(f) \quad \text{— } \textit{Leading Coefficient} \tag{4.9}$$

$$lm(f) \quad \text{— } \textit{Leading Monomial} \tag{4.10}$$

$$tail(f) \quad f - lt(f) \tag{4.11}$$

Example 4.2

$$f = 3a^2b + 2ab + 4bc \tag{4.12}$$

$$lt(f) = 3a^2b \tag{4.13}$$

$$lc(f) = 3 \tag{4.14}$$

$$lm(f) = a^2b \tag{4.15}$$

$$tail(f) = 2ab + 4bc \tag{4.16}$$

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

Definition 4.8 *Let* $\mathbb{K}$ *be a field and let* $f, g \in \mathbb{K}[x_1, x_2, \ldots, x_d]$ *be polynomials over the field.* **Polynomial division** *of* f *by* g *is computes following:*

$$f - \frac{lt(f)}{lt(g)} \cdot g \tag{4.17}$$

This polynomial division is denoted

$$f \xrightarrow{g} r \tag{4.18}$$

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

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

Example 4.3 *Over* $\mathbb{R}[x, y, z]$*, set the lex term order* $x > y > z$*. Let* $f = -2x^3 + 2x^2yz + 3xy^3$ *and* $g = x^2 + yz$*.*

$$\frac{lt(f)}{lt(g)} = \frac{-2x^3}{x^2} = -2x \tag{4.19}$$

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

$$\begin{aligned} r &= f - \frac{lt(f)}{lt(g)} \cdot g = -2x^3 + 2x^2yz + 3xy^3 - (-2x \cdot (x^2 + yz)) \\ &= -2x^3 + 2x^2yz + 3xy^3 - (-2x^3 - 2xyz) = 2x^2yz + 3xy^3 + 2xyz \end{aligned} \tag{4.20}$$

Notice that the division cancels the leading term of f*.*

4.2 Varieties and Ideals

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

Definition 4.9 *Let* $\mathbb{K}$ *be a field, and let* $f_1, \ldots, f_s \in \mathbb{K}[x_1, x_2, \ldots, x_d]$*. We call* $V(f_1, \ldots, f_s)$ *the* **affine variety** *defined by* $f_1, \ldots, f_s$ *as:*

$$V(f_1, \ldots, f_s) = \{(a_1, \ldots, a_d) \in \mathbb{K}^d : f_i(a_1, \ldots, a_d) = 0, \forall i, 1 \leq i \leq s\} \tag{4.21}$$

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

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

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

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

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

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

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

Definition 4.10 *A subset $I \subset \mathbb{K}[x_1, x_2, \ldots, x_d]$ is an* **ideal** *if it satisfies:*

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

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

Definition 4.11 *Let $f_1, f_2, \ldots, f_s$ be polynomials of the ring $\mathbb{K}[x_1, x_2, \ldots, x_d]$. Let I be an ideal generated by $f_1, f_2, \ldots, f_s$. Then:*

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

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

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

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

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

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

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

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

$$V(F_1) = V(F_2) = \{\pm 2, \pm 1\}. \tag{4.22}$$

Ideals and their varieties are a key part of computer-algebra based formal verification. A given hardware design can be transformed into a set of polynomials over a field, $f_1, \ldots, f_s \in F$ (we showed how this is done for Galois field arithmetic circuits in the previous chapter). This set of polynomials gives the system of equations:

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

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

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

$$x = z + 1$$
$$y = x^2 + 1$$

we can eliminate x *to obtain a new equation:*

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

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

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

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

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

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

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

Definition 4.12 *For any variety* V *of* $\mathbb{K}^d$*, the ideal of polynomials that vanish on* V*, called the vanishing ideal of* V*, is defined as* $I(V) = \{f \in \mathbb{F}[x_1, \ldots, x_d] : \forall \mathbf{a} \in V, f(\mathbf{a}) = 0\}$.

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

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

Definition 4.13 *Let $J \subset \mathbb{K}[x_1, \ldots, x_d]$ be an ideal. The radical of J is defined as $\sqrt{J} = \{f \in \mathbb{K}[x_1, \ldots, x_d] : \exists m \in \mathbb{N}, f^m \in J\}$.*

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

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

4.3 Gröbner Bases

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

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

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

Definition 4.14 *A set of non-zero polynomials $G = \{g_1, \ldots, g_t\}$ which generate the ideal $I = \langle g_1, \ldots, g_t \rangle$, is called a* **Gröbner basis** *for I if and only if for all $f \in I$ where $f \neq 0$, there exists a $g_i \in G$ such that $lm(g_i)$ divides $lm(f)$.*

$$G = \textit{GröbnerBasis}(I) \iff \forall f \in I : f \neq 0, \exists g_i \in G : lm(g_i) \mid lm(f) \tag{4.23}$$

The foundation for computing the Gröbner basis of an ideal was laid out by Buchberger [103]. Given a set of polynomials $F = \{f_1, \ldots, f_s\}$ that generate ideal $I = \langle f_1, \ldots, f_s \rangle$, Buchberger gives an algorithm to compute a Gröbner basis $G = \langle g_1, \ldots, g_t \rangle$. This algorithm relies on the notions of S-polynomials and polynomial reduction.

Definition 4.15 *For $f, g \in \mathbb{K}[x_1, \ldots, x_d]$, an* **S-polynomial** *$Spoly(f, g)$ is defined as:*

$$Spoly(f, g) = \frac{L}{lt(f)} \cdot f - \frac{L}{lt(g)} \cdot g \tag{4.24}$$

$$\textit{where } L = lcm\left(lt(f), lt(g)\right)$$

Note lcm denotes least common multiple.

Definition 4.16 *The* **reduction** *of a polynomial f, by another polynomial g, to a reduced polynomial r is denoted:*

$$f \xrightarrow{g} r$$

Reduction is carried out using multivariate, polynomial long division.

For sets of polynomials, the notation

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

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

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

Algorithm 2: Polynomial Reduction

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

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

Example 4.12 *Given* $f = y^2 - x$ *and* $f_1 = y - x$ *in* $\mathbb{Q}[x, y]$ *with* $deglex$*:* $y > x$*, perform* $f \xrightarrow{f_1}_+ r$*:*

1. $f = y^2 - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = y^2 - x - (y^2/y) \cdot (y - x) = y \cdot x - x$
2. $f = y \cdot x - x$, $f/f_1 = f - lt(f)/lt(f_1) \cdot f_1 = (y \cdot x - x)/f_1 = x^2 - x$
3. $f = x^2 - x$, *no more operations possible, so* $r = x^2 - x$

With the notions of S-polynomials and polynomial reduction in place, we can now present Buchberger's Algorithm for computing Gröbner bases [103]. Note that a fixed monomial (term) ordering is required for a Gröbner basis computation to ensure that polynomials are manipulated in a consistent manner.

Algorithm 3: Buchberger's Algorithm

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

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

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

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

When computing a Gröbner basis, it's important to note that if $lt(f_i)$ and $lt(f_j)$ have no common variables, the S-poly reduction step in Buchberger's algorithm, $Spoly(f_i, f_j) \xrightarrow{G'}_+$

r, will produce $r = 0$.

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

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

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

As mentioned previously, a Gröbner basis gives a decision procedure to test for polynomial membership in an ideal. This is explained in the following Theorem.

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

In other words,

$$f \in I \iff f \xrightarrow{G}_{+} 0 \tag{4.25}$$

Example 4.14 *Consider Example 4.13. Let $f = y^2x - x$ be another polynomial. Note that $f = yf_1 + f_2$, so $f \in I$. If we divide f by f_1 first and then by f_2, we will obtain a zero remainder. However, since the set $\{f_1, f_2\}$ is not a Gröbner basis, we find that the reduction $f \xrightarrow{f_2} x^2 - x \xrightarrow{f_1} x^2 - x \neq 0$; i.e. dividing f by f_2 first and then by f_1 does not lead to a zero remainder. However, if we compute the Gröbner basis G of I, $G = \{x^2 - x, yx - y, y^2 - x\}$, dividing f by polynomials in G in any order will always lead to the zero remainder. Therefore, one can decide ideal membership unequivocally using the Gröbner basis.*

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

Definition 4.17 *A* **minimal Gröbner basis** *for a polynomial ideal I is a Gröbner basis G for I such that*

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

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

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

A minimal Gröbner basis can then be further reduced.

Definition 4.18 *A* **reduced Gröbner basis** *for a polynomial ideal I is a Gröbner basis $G = \{g_1, \ldots, g_t\}$ such that:*

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

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

Definition 4.19 *Let $H = \{h_1, \ldots, h_t\}$ be a minimal Gröbner basis. Apply the following reduction process:*

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

  $\vdots$

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

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

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

Proposition 4.3 *[10] Let* $I \neq \{0\}$ *be a polynomial ideal. Then, for a given monomial ordering,* I *has a unique reduced Gröbner basis.*

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

4.4 Elimination Theory

Elimination theory uses **elimination ordering** to systematically eliminate variables from a system of polynomial equations.

Definition 4.20 *Let* I *be an ideal in* $\mathbb{K}[x_1, \ldots, x_k]$. *The* i*-th* **elimination ideal** I_i *is the ideal of* $\mathbb{K}[x_{i+1}, \ldots, x_k]$ *defined by*

$$I_k = I \cap \mathbb{K}[x_{i+1}, \ldots, x_k] \tag{4.26}$$

The elimination ideal I_i has eliminated all the variables $x_1, \ldots, x_i$, i.e. it only contains polynomials with variables in $x_{i+1}, \ldots, x_k$. We can generate elimination ideals by computing Gröbner bases using elimination orderings.

Theorem 4.2 [**Elimination Theorem**] *Let* I *be an ideal in* $\mathbb{K}[x_1, \ldots, x_k]$ *and let* G *be the Gröbner basis of* I *with respect to the lex order (elimination order)* $x_1 > x_2 > \cdots > x_k$. *Then, for every* $0 \leq i \leq k$,

$$G_k = G \cap \mathbb{K}[x_{i+1}, \ldots, x_k] \tag{4.27}$$

is a Gröbner basis of the i-th elimination ideal I_i.

This can be better visualized using the following example.

Example 4.15 *Given the following equations in $\mathbb{R}[x, y, z]$*

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

let I be the ideal generated by these equations:

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

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

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

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

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

Elimination theory provides the basis for our abstraction approach.

4.5 Hilbert's Nullstellensatz

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

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

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

An interesting result is one of **Strong Nullstellensatz**. The strong Nullstellensatz establishes the correspondence between radical ideals and varieties.

Theorem 4.3 *(The Strong Nullstellensatz [10]) Let $\overline{\mathbb{K}}$ be an algebraically closed field, and let J be an ideal in $\overline{\mathbb{K}}[x_1, \ldots, x_d]$. Then we have $I(V_{\overline{\mathbb{K}}}(J)) = \sqrt{J}$.*

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

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

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

The proof is given in [67]. Here, we provide a proof outline.

Proof.

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

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

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

■

4.6 Concluding Remarks

Our approach to word-level abstraction of Galois field arithmetic circuits applies concepts of polynomial ideals, varieties, Gröbner basis, and elimination theory to abstract a word-level representation of the circuit. This approach is described in the next chapter. However, a Gröbner basis computation is prohibitively expensive; thus we propose improvements to our original approach in a subsequent chapter.

CHAPTER 5

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.

5.1 Motivation

The inspiration of this research mainly comes from a journal paper [104]. In that paper, the author proposed an traversal algorithm using concept “implicit state enumeration”. Concretely, the algorithm is written as follows:

Algorithm 4: SIMPSON: Scan Register Selection using Implicit State Enumeration [104]

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

$from^0 = reached = S^0$;
$i = 0$;
while *TRUE* **do**
    $i++$;
    to^i =IMAGE($\Delta, from^{i-1}$);
    $new^i = to^i \cdot \overline{reached}$;
    for *each state variable* r_j **do**
        record_if_transitions_present_or_missing(r_j, new^i);
        compute_degree_of_unsettability(r_j, new^i);
    end
    if $new^i == 0$ **then**
        break;
    end
    $reached = reached + new^i$;
    $from^i = new^i$;
end
/* FSM traversal completed */
for *each state variable* r_j **do**
    if *missing transition for* r_j **then**
        scan state variable r_j;
    end
end
$illegal\_states$ =bdd_complement($reached$);
for *each state variable* r_j **do**
    compute_degree_of_unateness($r_j, illegal\_states$);
    non-controllability(r_j) =degree_of_unsettability(r_j)+degree_of_unateness(r_j);
end
order state variables in terms of their non-controllabilities;/* Sorting */
output the required scan registers;

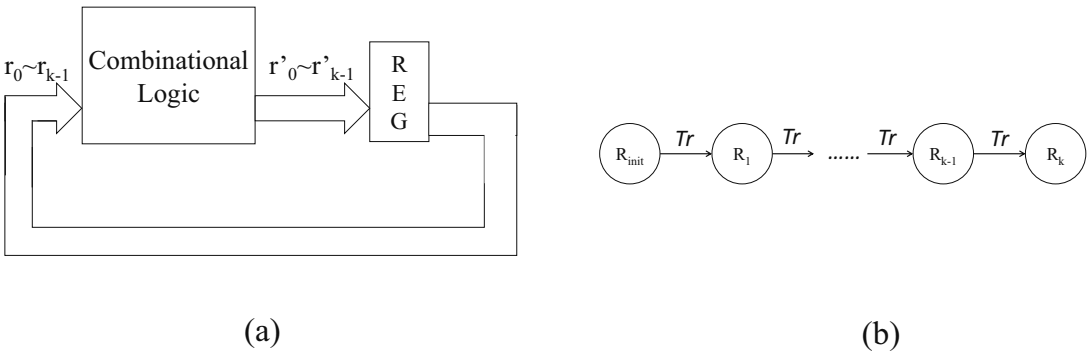
CHAPTER 6

FUNCTIONAL VERIFICATION OF SEQUENTIAL NORMAL BASIS MULTIPLIER

In order to utilize our traversal algorithm, it is necessary to find out a sort of suitable circuit benchmarks which is easy to compute its Gröbner basis (GB). From the work of Lv 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.

6.1 Motivation

From the preliminaries (Chapter 3) about FSMs, we learn that the Moore machine does not rely on inputs for state transitions. As depicted in Figure 6.1(a), a typical Moore machine implementation consists of combinational logic component and register files, where $r_0, \ldots, r_k$ 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 6.1(b) shows the state transition graph (STG) of a Moore machine with $k+1$ distinct states. We notice that it forms a simple chain, with k consecutive transitions the machine reaches final state R_k.



(a) (b)

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

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

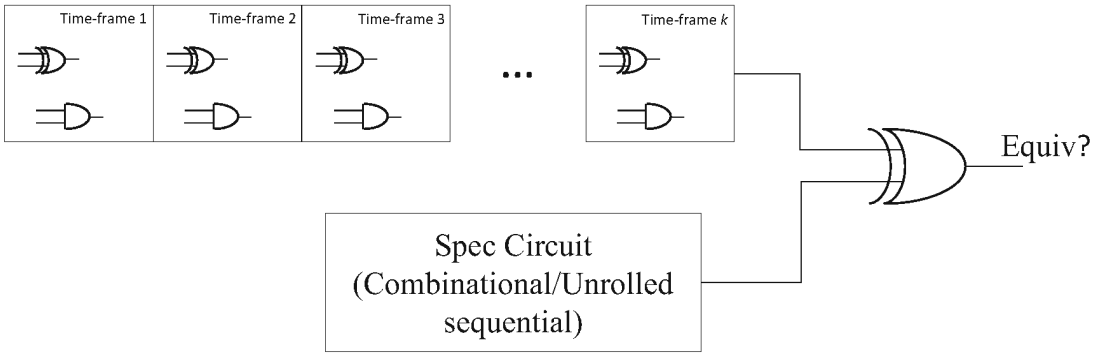


Figure 6.2: Conventional verification techniques based on bit-level unrolling and equivalence checking

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

6.2 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.

6.2.1 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 6.1: 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

$$\begin{aligned} A + B = A - B &= (a_0, a_1, \ldots, a_{k-1})_{StdB} \bigoplus (b_0, b_1, \ldots, b_{k-1})_{StdB} \\ &= (a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{k-1} \oplus b_{k-1})_{StdB} \end{aligned} \tag{6.1}$$

6.2.2 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 [105] 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 6.1. 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 6.1 (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 6.1 (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 6.1 can be well utilized.

Example 6.2 (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}:

$$c_{n-1} = f_{mult}(a_0, a_1, \ldots, a_{n-1}; b_0, b_1, \ldots, b_{n-1}) \tag{6.2}$$

Before discussing the details of the function f_{mult}, we can take a square on both side of equation 6.2, i.e. $C^2 = A^2 \times B^2$. Obviously, using the property in lemma 6.1,

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

$$c_{k-2} = f_{mult}(a_{k-1}, a_0, a_1, \ldots, a_{k-2}; b_{k-1}, b_0, b_1, \ldots, b_{k-2}) \tag{6.3}$$

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

$$c_l = f_{mult}(A, B) = A \times M \times B^T \tag{6.4}$$

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 6.4, 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 6.3 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 6.3 (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

$$Product\ vector\ C = (\sum_{i=0}^{k-1} a_i \beta^{2^i})(\sum_{j=0}^{k-1} b_j \beta^{2^j}) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j \beta^{2^i} \beta^{2^j} \tag{6.5}$$

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

$$\beta^{2^i}\beta^{2^j} = \sum_{l=0}^{k-1} \lambda_{ij}^{(l)} \beta^{2^l}, \quad \lambda_{ij}^{(l)} \in \mathbb{F}_2. \tag{6.6}$$

Substitution yields, result is an expression for l-th digit of product as showed in equation 6.2:

$$c_l = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \lambda_{ij}^{(l)} a_i b_j \tag{6.7}$$

$\lambda_{ij}^{(l)}$ 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. [106] 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.

6.2.3 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 6.4 (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 [107]. 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 [108]. 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 [105]) contains 53 gates in total, which is a great saving in area cost comparing to StdB multipliers.

6.3 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 [105]. 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.

6.3.1 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 **??** 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:

$$c_l = row_1(A \times M^{(l)}) \times B + row_2(A \times M^{(l)}) \times B + \cdots + row_k(A \times M^{(l)}) \times B \quad (6.8)$$

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. [109].

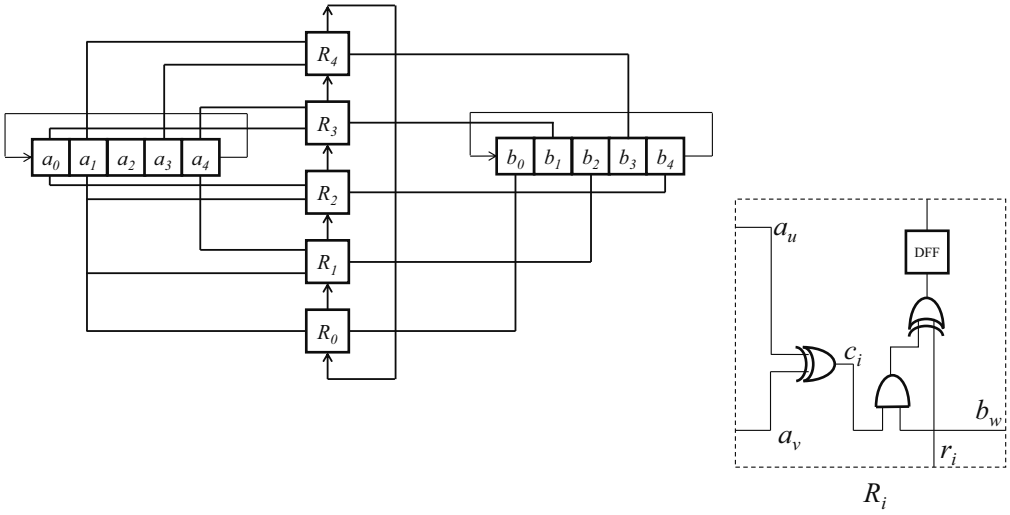


Figure 6.3: 5-bit Agnew's SMPO. Index i satisfies $0 < i < 4$, indices u, v are determined by column # of nonzero entries in i-th row of λ-Matrix $M^{(0)}$, i.e. if entry $M_{ij}^{(0)}$ is a nonzero entry, u or v equals to $i + j \pmod 5$. Index $w = 2i \pmod 5$

Example 6.5 (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 6.3.

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

$$c_0 = a_1b_0 + (a_0 + a_3)b_1 + (a_3 + a_4)b_2 + (a_1 + a_2)b_3 + (a_2 + a_4)b_4 \tag{6.9}$$

Note it is written in the form of equation 6.8. In next clock cycles we can obtain remaining bits of the product, which can be written in following general form polynomial:

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

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 6.9. 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 6.9. 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 6.9. 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.

6.3.2 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 [110], which is therefore called RH-SMPO.

Figure **??** is a 5-bit RH-SMPO which is functionally equivalent to 5-bit Agnew's SMPO in figure 6.3. A brief proof is as follows:

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

$$F_i(A,B) = a_i b_i \beta + \sum_{j=1}^{v} d_{i,j} \beta^{1+2^j} \tag{6.10}$$

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

$$d_{i,j} = c_{a,i} c_{b,i} = (a_i + a_{i+j})(b_i + b_{i+j}), \;\; 1 \leq j \leq v \tag{6.11}$$

$i+j$ 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 [110]):

Theorem 6.1 *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 5: NB Multiplication Algorithm in RH-SMPO [110]

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.6.10*/;
    $A \leftarrow A^2,\ B \leftarrow B^2$ /*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 6.6 (Designing a 5-bit RH-SMPO) *From equation 6.10 we can deploy AND gates in d-layer according to $d_{i,j}$, and XOR gates in c-layer according to equation 6.11. Concretely, as algorithm 5 describes, we implement auxiliary function F_{k-1} in the logic:*

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

$$F_4(A, B) = a_4 b_4 \beta + \sum_{j=1}^{2} d_{4,j} \beta^{1+2^j} = d_0 \beta + \sum_{j=1}^{2} d_{4,j} \beta^{1+2^j} \tag{6.12}$$

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

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

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

The difficult part of the whole design is to deploy XOR gates in e-layer. As the logic layer closest to the outputs R_i, e-layer actually finishes the implementation of $F_{k-1}(A, B)$. But it is not a simple addition; the reason is before bit-wise adding to X^2, it is necessary to

turn the sum to NB form. In other words, theoretically we need k XOR gates in e-layer, the output of i-th gate corresponds to the bit multiplying β^{2^i}.

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

$$\begin{bmatrix} \beta^{1+2^0} \\ \beta^{1+2^1} \\ \beta^{1+2^2} \\ \vdots \\ \beta^{1+2^{k-1}} \end{bmatrix} = \beta \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} T_{0,0} & T_{0,1} & \dots & T_{0,k-1} \\ T_{1,0} & T_{1,1} & \dots & T_{1,k-1} \\ T_{2,0} & T_{2,1} & \dots & T_{2,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ T_{k-1,0} & T_{k-1,1} & \dots & T_{k-1,k-1} \end{bmatrix} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} = \mathbf{T} \begin{bmatrix} \beta \\ \beta^2 \\ \beta^4 \\ \vdots \\ \beta^{2^{k-1}} \end{bmatrix} \tag{6.13}$$

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:

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 6.13:

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

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 6.12, 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 **??***, 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 $2k + \lfloor \frac{k}{2} \rfloor$.

6.4 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 [111].

6.4.1 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 6.10. 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 6: 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 $from_k(R_{final})$

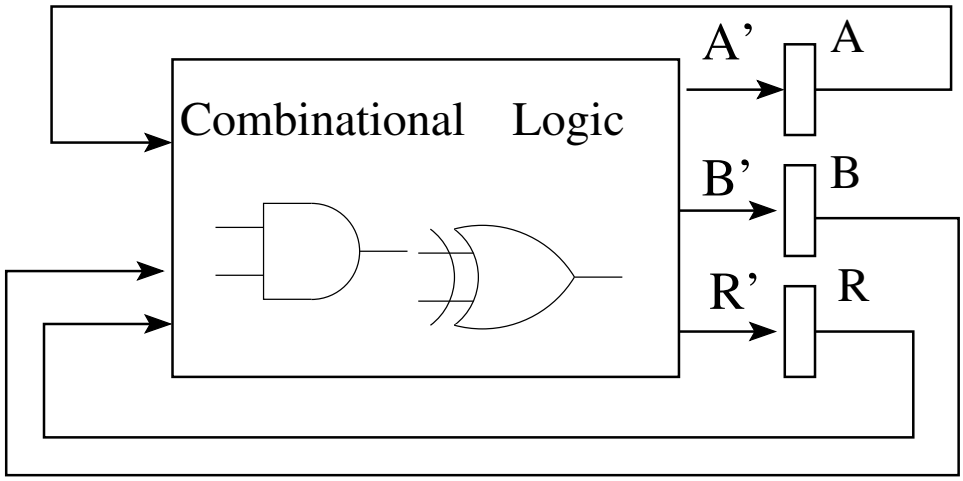


Figure 6.4: A typical normal basis GF sequential circuit model. $A = (a_0, \ldots, a_{k-1})$ and similarly B, R are k-bit registers; A', B', R' denote next-state inputs.

We follow the sequential GF circuit model of figure 6.4, with word-level variables A, B, R denoting *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 vari-

ables (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 6 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 6.7 (Functional verification of 5-bit RH-SMPO) *Figure* **??** *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 6 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 6, 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 6 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:* $\mathbf{from^0} = \{R, A_{init} + a_0\alpha^5 + a_1\alpha^{10} + a_2\alpha^{20} + a_3\alpha^9 + a_4\alpha^{18}, B_{init} + b_0\alpha^5 + b_1\alpha^{10} + b_2\alpha^{20} + b_3\alpha^9 + b_4\alpha^{18}\}$,
  $\mathbf{to^1} = \{R' + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}^{16}B_{init}^{4} + (\alpha^3+1)A_{init}^{16}B_{init}^{2} + (\alpha^4+\alpha^3+1)A_{init}^{16}B_{init} + (\alpha^4+\alpha^3+\alpha^2+1)A_{init}^{8}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{4} + (\alpha^3+\alpha+1)A_{init}^{8}B_{init}^{2} + (\alpha^4+\alpha^2)A_{init}^{8}B_{init} + (\alpha^4+\alpha^2)A_{init}^{4}B_{init}^{16} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init}^{8} + (\alpha^2)A_{init}^{4}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{4}B_{init}^{2} + (\alpha^4+\alpha^3+\alpha+1)A_{init}^{4}B_{init} + (\alpha^3+1)A_{init}^{2}B_{init}^{16} + (\alpha^3+\alpha+1)A_{init}^{2}B_{init}^{8} + (\alpha^3+\alpha^2+\alpha+1)A_{init}^{2}B_{init}^{4} + (\alpha^3+\alpha^2+\alpha)A_{init}^{2}B_{init}^{2} + (\alpha^4+\alpha)A_{init}^{2}B_{init} + (\alpha^4+\alpha^3+1)A_{init}B_{init}^{16} + (\alpha^4+\alpha^2)A_{init}B_{init}^{8} + (\alpha^4+\alpha^3+\alpha+1)A_{init}B_{init}^{4} + (\alpha^4+\alpha)A_{init}B_{init}^{2} + (\alpha^3+\alpha+1)A_{init}B_{init}, A_{init} + a'_4\alpha^5 + a'_0\alpha^{10} + a'_1\alpha^{20} + a'_2\alpha^9 + a'_3\alpha^{18}, B_{init} + b'_4\alpha^5 + b'_0\alpha^{10} + b'_1\alpha^{20} + b'_2\alpha^9 + b'_3\alpha^{18}\}$

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

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

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

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

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

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

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

6.4.2 Overcome Computational Complexity using RATO

Similar to our improvements in section **??**, RATO [112] 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 **??**.

Example 6.8 *Variable order under RATO is:*

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

Search among all generators of J_{ckt} from Ex.6.7 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 6.9 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

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

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

6.4.3 Solving Linear System for Bit-to-Word Substitution

In example 6.9 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

$$\begin{bmatrix} S \\ S^2 \\ S^{2^2} \\ \vdots \\ S^{2^{k-1}} \end{bmatrix} = \begin{bmatrix} \beta & \beta^2 & \beta^{2^2} & \cdots & \beta^{2^{k-1}} \\ \beta^2 & \beta^{2^2} & \beta^{2^3} & \cdots & \beta \\ \beta^{2^2} & \beta^{2^3} & \beta^{2^4} & \cdots & \beta^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \beta^{2^{k-1}} & \beta & \beta^2 & \cdots & \beta^{2^{k-2}} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{k-1} \end{bmatrix} \tag{6.14}$$

Let $\mathbf{s}$ be a vector of k unknowns $s_0, \ldots, s_{k-1}$, then equation 6.14 can be solved by using Cramer's rule:

$$s_i = \frac{|\mathbf{M_i}|}{|\mathbf{M}|}, \;\; 0 \leq i \leq k-1, |\mathbf{M}| \neq 0 \tag{6.15}$$

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

$$\det M(\alpha_0,\ldots,\alpha_{k-1}) = \begin{vmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{k-1} \\ \alpha_0^p & \alpha_1^p & \cdots & \alpha_{k-1}^p \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{p^{k-1}} & \alpha_1^{p^{k-1}} & \cdots & \alpha_{k-1}^{p^{k-1}} \end{vmatrix} \tag{6.16}$$

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

Moore determinant can be written as an explicit expression

$$\det M(\alpha_0,\ldots,\alpha_{k-1}) = \alpha_0 \prod_{i=1}^{k-1} \prod_{c_0,\ldots,c_{i-1}\in\mathbb{F}_p} (\alpha_i - \sum_{j=0}^{i-1} c_j\alpha_j) \tag{6.17}$$

We use an example to help understanding the notations in Eqn.6.17:

Example 6.10 *Let* $\{\alpha_0,\alpha_1,\alpha_2\}$ *be a set of elements of* $\mathbb{F}_{2^3}$. *Then*

$$\begin{aligned} \det M(\alpha_0,\alpha_1,\alpha_2) &= \begin{vmatrix} \alpha_0 & \alpha_1 & \alpha_2 \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 \\ \alpha_0^4 & \alpha_1^4 & \alpha_2^4 \end{vmatrix} \\ &= \alpha_0 \prod_{i=1}^{2} \prod_{c_0,\ldots,c_{i-1}\in\mathbb{F}_2} (\alpha_i - \sum_{j=0}^{i-1} c_j\alpha_j) \end{aligned} \tag{6.18}$$

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

$$\det M(\alpha_0,\alpha_1,\alpha_2) = \alpha_0\alpha_1(\alpha_1-\alpha_0)\alpha_2(\alpha_2-\alpha_1)(\alpha_2-\alpha_0)(\alpha_2-\alpha_0-\alpha_1) \tag{6.19}$$

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

$$\det M(\alpha_0,\alpha_1,\ldots,\alpha_{k-1}) = 1 \tag{6.20}$$

Proof A: ccording to the definition Eqn.6.17, 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 6.3 to Eqn.6.15 gives

$$s_i = |\mathbf{M_i}|, \;\; 0 \leq i \leq k-1 \tag{6.21}$$

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

...

6.5 Software Implementation of Implicit Unrolling Approach

Our experiment on different size of SMPO designs is performed with both SINGULAR [73] 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.

6.5.1 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 6 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 6. 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.

6.5.2 Architecture in Customized C++ Toolset

6.6 Experimental Results

We have implemented our approach within the SINGULAR symbolic algebra computation system [v. 3-1-6] [73]. Using our implementation, we have performed experiments to verify two SMPO architectures — Agnew-SMPO [109] and the RH-SMPO [110] — 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 [63] is used as the *spec* model. This miter is checked for SAT using the *Lingeling* [113] solver. We also experiment with the Combinational Equivalence Checking (CEC) engine of the ABC tool [62], which uses AIG-based reductions to identify internal AIG equivalences within the miter to efficiently solve verification. The BDD-based VIS tool [114] is also used for equivalence check. The run-times for verification of (unrolled) RH-SMPO against Mastrovito *spec* are given in Table 6.2 – which shows that the techniques fail beyond 23 bit fields.

CEC between unrolled RH-SMPO and Agnew-SMPO also suffers the same fate (results omitted). In fact, both SMPO designs are based on slightly different mathematical concepts and their computations in all clock-cycles, except for the k^{th} one, are also different. These designs have no internal logical/structural equivalencies, and verification with SAT/BDDs/ABC is infeasible. Their dissimilarity is depicted in Table 6.3, where

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

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

N_1 depicts the number of AIG nodes in the miter prior to *fraig_sweep*, and the nodes after *fraiging* are recorded as N_2; so $\frac{N_1-N_2}{N_1}$ reflects the proportion of equivalent nodes in original miter, which emphasizes the (lack of) *similarity* between two designs.

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

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

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

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

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

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

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

6.7 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 7

CONCLUSIONS AND FUTURE WORK

A combinational circuit with k-inputs and k-outputs implements Boolean functions $f : \mathbb{B}^k \rightarrow \mathbb{B}^k$, where $\mathbb{B} = \{0, 1\}$. The function can also be construed as a mapping $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$, where $\mathbb{F}_{2^k}$ denotes the Galois field of 2^k elements. A circuit with differing input and output sizes computes $f : \mathbb{B}^m \rightarrow \mathbb{B}^n$, which can be represented as a function over Galois fields $f : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^n}$. This circuit can also be analyzed as the function $f : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$, where $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$ and $\mathbb{F}_{2^k} \supset \mathbb{F}_{2^m}$.

Every function f over $\mathbb{F}_{2^k}$ is a polynomial function — i.e., there exists a unique, minimal, canonical polynomial $\mathcal{F}$ that describes f. This dissertation presented novel techniques based on computer-algebra and algebraic-geometry to derive the canonical (word-level) polynomial representation from the circuit as $Z = \mathcal{F}(A)$ over $\mathbb{F}_{2^k}$, where A and Z denote, respectively, the input and output bit-vectors of the circuit.

A theory for word-level polynomial abstraction of bit-level circuits over Galois fields is first developed. This theory is derived using techniques from computer-algebra, notably the theory of Gröbner basis. However, due to the computational complexity of computing a Gröbner basis, the solution is not scalable to large designs. In order to overcome these limitations, new symbolic computational algorithms are developed and refined. The algorithms employ techniques from the binomial expansion over $\mathbb{F}_{2^k}$ and $\mathbb{F}_4$-style reduction and can exploit hierarchy in a given circuit. Finally, an efficient implementation of the algorithmic approach is presented.

Experiments show that the proposed approach works exceptionally well for abstracting word-level Galois field arithmetic circuits. It has been shown that the approach can abstract and verify these types of circuits with up to 1024-bit datapaths. Other contemporary techniques cannot to verify these types for circuits beyond 163-bits and fail to abstract them beyond 32-bits.

However, in cases of random logic, the abstraction approach can generate high-degree polynomials:

$$X^{q-1} + X^{q-2} \ldots \tag{7.1}$$

In these cases, the polynomials derived during the computation are dense, and the computational complexity of manipulating such polynomials makes abstraction infeasible.

7.1 Future Work

Due to the modular nature of the proposed solution, there are many potential future research directions that can be explored.

7.1.1 Hardware Acceleration

The first reduction step, $f_Z \xrightarrow{F-f_{z_i},F_0}_{+} r$ is the most computationally complex part of the proposed abstraction approach. This reduction could be implemented using a hardware accelerator. Significant speed-ups have been observed in GPU implementations of circuit simulation algorithms [115]. Furthermore, this work has shown cases where multiple, independent abstractions need to be computed at the same time, such as when abstracting a word-level representation of a composite field multiplier.

These abstractions can be computed in parallel with one another, and this parallelism could then be exploited using a GPU. Furthermore, our approach to compute the abstractions uses an $F4$-style reduction procedure, which performs many complex computations over a large matrix. Operations over matrices can be suitably implemented using a GPU. Lastly, the substitution by $a_i = \mathcal{F}(A)$ is trivially parallelized. Thus, further study is proposed to implement word-level abstraction on a general purpose GPU.

7.1.2 Integration with EDA Tools

The proposed canonical word-level abstraction approach is a full, self-contained solution. It can thus be integrated into other EDA tools. There are direct applications of word-level abstractions to design synthesis. For instance, the approach can compute a functional decomposition of a logic or it could be used in high-level RTL synthesis. Since the derived abstraction is canonical, it can also be used in verification engines such as SMT solvers. The abstraction approach most efficiently handles AND/XOR logic, so

it could be used to complement approaches in the mentioned tools that are efficient over AND/OR logic.

7.1.3 Polynomial Reductions using Data-Structures

The abstraction approach poorly handles chains of OR gates due to their representation as polynomials of Galois fields. Other polynomial-based tools [116] have shown that it can be beneficial to represent polynomials internally as decision diagrams. Thus, it is worthwhile to explore whether it is possible to implement the algorithmic approach in a different data-structure that is better-suited for handling this type of logic. One candidate data-structure is the And-Invert-Graph, as this structure efficiently handles OR gates. The widely-used tool ABC [62] provides a very efficient, flexible, and open-source implementation of the AIG data structure.

Recall that a one-step reduction of the polynomial f by polynomial g, $f \xrightarrow{g} r$, is computed as:

$$r = f - \frac{LT(f)}{LT(g)} \cdot g \tag{7.2}$$

Over Boolean circuits, $\mathbb{B} \equiv \mathbb{F}_2$. Since the leading term of any monomial in $\mathbb{B}$ is 1, and $-1 \equiv +1$, then

$$f - (\frac{LT(f)}{LT(g)} \cdot g) \equiv f + (\frac{LM(f)}{LM(g)} \cdot g) \tag{7.3}$$

which computes an XOR operation

$$f \oplus (\frac{LM(f)}{LM(g)} \cdot g) \tag{7.4}$$

while the $\cdot$ operator acts as an AND operation $\wedge$. So one step of the reduction procedure can be computed as the following AND/XOR operation:

$$r = f \oplus \frac{LM(f)}{LM(g)} \wedge g \tag{7.5}$$

Thus, we propose an investigation into implementing the algorithms presented in this dissertation over AIGs.

7.1.4 Application to Sequential Circuit Verification

Sequential Galois field arithmetic circuits over $\mathbb{F}_{2^k}$ take k-bit inputs and produce a k-bit result after k-clock cycles of operation. Formal verification of sequential arithmetic

circuits with large datapath sizes is beyond the capabilities of contemporary verification techniques. To address this problem, we described a verification method in [26] which uses the presented abstraction approach to implicitly unroll the sequential arithmetic circuit over multiple (k) clock-cycles. The resulting function computed by the state-registers of the circuit is represented canonically as a multi-variate word-level polynomial over $\mathbb{F}_{2^k}$. While directly applicable to sequential Galois field arithmetic circuits, this work needs to be further generalized in order to make applicable to any sequential state machine.

7.1.5 Application to Formal Software Verification

Computer algebra techniques based on Gröbner basis theory have been used in formal software verification [69]. In this work, a Gröbner basis computation is used to derive *loop invariants*. However, the derived invariants are not bit-precise, so not every invariant that is computed can be applied to the verification. As our approach maintains the input-output relationship in the abstraction, it could be applied to find bit-precise invariants.

7.1.6 Application to Integer Arithmetic Circuits

The abstraction approach derives a word-level representation of circuits over Galois fields, $\mathbb{F}_{2^k}$. In order to expand its usability, we conjecture whether it is possible to apply concepts from this approach to **abstract word-level representations of circuits over integer rings**, $\mathbb{Z}_{2^k}$. As any function over a Galois field $\mathbb{F}_{2^k}$ is a polynomial function, there exists a polynomial which describes the word-level function of a given circuit over $\mathbb{F}_{2^k}$. However, not every function over an integer ring $\mathbb{Z}_{2^k}$ is a polynomial function. Thus, a single polynomial which describes the function of a circuit over $\mathbb{Z}_{2^k}$ is not guaranteed to exist. Even though there may not exist a single polynomial which describes the entire function over $\mathbb{Z}_{2^k}$, elimination theory and Gröbner basis still applies over this ring. Thus, it may be possible to modify the theory and implementation of our word-level abstraction approach in order to abstract a set of word-level polynomials over $\mathbb{Z}_{2^k}$.

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] E. Biham, Y. Carmeli, and A. Shamir, "Bug Attacks", *in Proceedings on Advances in Cryptology*, pp. 221–240, 2008.

[2] Thomas R. Nicely, "Pentium FDIV Flaw", `http://www.trnicely.net/pentbug/pentbug.html`.

[3] Douglas N. Arnold, "The Patriot Missile Failure", `http://www.ima.umn.edu/~arnold/disasters/patriot.html`.

[4] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, First edition, 1991.

[5] E. Clarke, O. Grumberg, and D. Peled, *The Temporal Logic of Reactive and Concurrent Systems*, The MIT Press, 1999.

[6] F. Lu, L. Wang, K. Cheng, and R. Huang, "A Circuit SAT Solver With Signal Correlation Guided Learning", *in IEEE Design, Automation and Test in Europe*, pp. 892–897, 2003.

[7] G. Avrunin, "Symbolic Model Checking using Algebraic Geometry", *in Computer Aided Verification Conference*, pp. 26–37, 1996.

[8] 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.

[9] Y. Watanabe and *et al*, "Application of Symbolic Computer Algebra to Arithmetic Circuit Verification", *in IEEE International Conference on Computer Design*, pp. 25–32, October 2007.

[10] W. W. Adams and P. Loustaunau, *An Introduction to Gröbner Bases*, American Mathematical Society, 1994.

[11] J. Lv, *Scalable Formal Verification of Finite Field Arithmetic Circuits using Computer Algebra Techniques*, PhD thesis, Univ. of Utah, Aug. 2012.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word Level Predicate Abstraction and Refinement Techniques for Verifying Rtl Verilog", *in Design Automation Conf.*, 2005.

[13] S. Horeth and Drechsler, "Formal Verification of Word-Level Specifications", *in IEEE Design, Automation and Test in Europe*, pp. 52–58, 1999.

[14] L. Arditi, "*BMDs can Delay the use of Theorem Proving for Verifying Arithmetic Assembly Instructions", in Srivas, editor, *In Proc. Formal methods in CAD*. Springer-Verlag, 1996.

[15] Z. Zeng, P. Kalla, and M. J. Ciesielski, "LPSAT: A Unified Approach to RTL Satisfiability", *in Proc. DATE*, 2001.

[16] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays", *in TACAS 09, Volume 5505 of LNCS*. Springer, 2009.

[17] R. Brant, D. Kroening, and *et al*, "Deciding Bit-Vector Arithmetic with Abstraction", *in Proc. TACAS*, pp. 358–372, 2007.

[18] D. Babic and M. Musuvathi, "Modular Arithmetic Decision Procedure", Technical Report TR-2005-114, Microsoft Research, 2005.

[19] N. Tew, P. Kalla, N. Shekhar, and S. Gopalakrishnan, "Verification of Arithmetic Datapaths using Polynomial Function Models and Congruence Solving", *in Proc. Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 122–128, 2008.

[20] A. Gupta, "Formal Hardware Verification Methods: A Survey", *Formal Methods in System Design*, vol. 1, pp. 151–238, 1992.

[21] J. Smith and G. DeMicheli, "Polynomial methods for component matching and verification", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1998.

[22] J. Smith and G. DeMicheli, "Polynomial Methods for Allocating Complex Components", *in IEEE Design, Automation and Test in Europe*, 1999.

[23] A. Peymandoust and G. DeMicheli, "Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis", *IEEE Transactions CAD*, vol. 22, pp. 1154–11656, 2003.

[24] T. Pruss, P. Kalla, and F. Enescu, "Word-Level Abstraction from Bit-Level Circuits using Gröbner Basis", *in International Workshop on Logic and Synthesis*, 2013.

[25] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases", *in Design Automation Conference*, 2014.

[26] X. Sun, P. Kalla, T. Pruss, and F. Enescu, "Fquivalenceormal Verification of Sequential Galois Field Arithmetic Circuits using Algebraic Geometry", *in Design Automation for Test in Europe*, 2015.

[27] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction from Combinational Circuits for Verification over Galois Fields", *IEEE Transactions on CAD (in review)*, 2015.

[28] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

[29] Randal E. Bryant Karl S. Brace, Richard L. Rudell, "Efficient implementation of a bdd package", *in DAC, pp40-45*, 1990.

[30] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, "Efficient Representation and Manipulation of Switching Functions based on Ordered Kronecker Functional Decision Diagrams", *in Design Automation Conference*, pp. 415–419, 1994.

[31] I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 188–191, Nov. 93.

[32] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping", *in DAC*, pp. 54–60, 93.

[33] E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams - Overcoming the Limitation of MTBDDs and BMDs", *in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 159–163, 1995.

[34] Y-T. Lai, M. Pedram, and S. B. Vrudhula, "FGILP: An ILP Solver based on Function Graphs", *in ICCAD*, pp. 685–689, 93.

[35] R. E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", *in Proceedings of Design Automation Conference*, pp. 535–541, 1995.

[36] R. Dreschler, B. Becker, and S. Ruppertz, "The K*BMD: A Verification Data Structure", *IEEE Design & Test of Computers*, vol. 14, pp. 51–59, 1997.

[37] Y. A. Chen and R. E. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Verification", *in Proc. ICCAD*, 1997.

[38] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data-Flow Designs", *IEEE Transactions on Computers*, vol. 55, pp. 1188–1201, 2006.

[39] N. Shekhar, *Equivalence Verification of Arithmetic Datapaths using Finite Ring Algebra*, PhD thesis, Univ. of Utah, Dept. of Electrical and Computer Engineering, Aug. 2007.

[40] B. Alizadeh and M. Fujita, "Modular Datapath Optimization and Verification based on Modular-HED", *IEEE Transactions CAD*, pp. 1422–1435, Sept. 2010.

[41] A. Jabir and Pradhan D., "MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions", *in IEEE Design, Automation and Test in Europe*, 2004.

[42] A. Jabir, D. Pradhan, T. Rajaprabhu, and A. Singh, "A Technique for Representing Multiple Output Binary Functions with Applications to Verification and Simulation", *IEEE Transactions on Computers*, vol. 56, pp. 1133–1145, 2007.

[43] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vencentelli, F. Somenzi, A. Aziz, S-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, G. Shiple, S. Swamy, and T. Villa, "VIS: A System for Verification and Synthesis", *in Computer Aided Verification*, 1996.

[44] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[45] C. Barrett and C. Tinelli, "CVC3", *in Computer Aided Verification Conference*, pp. 298–302. Springer, July 2007.

[46] L. Moura and N. Bjrner, "Z3: An Efficient SMT Solver.", *in International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963. Springer, 2008.

[47] C. W. Barlett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for bit-Vector Arithmetic", *in DAC*, June 1998.

[48] H. Enderton, *A mathematical introduction to logic*, Academic Press New York, 1972.

[49] T. Bultan and et al, "Verifying systems with integer constraints and boolean predicates: a composite approach", *in In Proc. Int'l. Symp. on Software Testing and Analysis*, 1998.

[50] S. Devadas, K. Keutzer, and A. Krishnakumar, "Design verification and reachability analysis using algebraic manipulation", *in Proc. ICCD*, 91.

[51] Z. Zhou and W. Burleson, "Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions", *in DAC*, 95.

[52] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, "Difference decision diagrams", *in Computer Science Logic*, The IT University of Copenhagen, Denmark, Sep. 1999.

[53] Jesper Møller and Jakob Lichtenberg, "Difference decision diagrams", Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Aug. 1998.

[54] K. Strehl, "Interval Diagrams: Increasing Efficiency of Symbolic Real-Time Verification", *in Intl. Conf. on Real Time Computing systems and Applications*, 1999.

[55] P. Sanchez and S. Dey, "Simulation-Based System-Level Verification using Polynomials", *in High-Level Design Validation & Test Workshop, HLDVT*, 1999.

[56] G. Ritter, "Formal Verification of Designs with Complex Control by Symbolic Simulation", in Springer Verlag LCNS, editor, *Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[57] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL models using Linear Programming and 3-Satisfiability", *in Proc. DAC*, '98.

[58] R. Brinkmann and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming", *in Proc. ASP-DAC*, 2002.

[59] C.-Y. Huang and K.-T. Cheng, "Using Word-Level ATPG and Modular Arithmetic Constraint Solving Techniques for Assertion Property Checking", *IEEE Trans. CAD*, vol. 20, pp. 381–391, 2001.

[60] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 1377–1394, Nov. 2006.

[61] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to Combinational Equivalence Checking", *in Proc. Intl. Conf. on CAD (ICCAD)*, pp. 836–843, 2006.

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

[63] E. Mastrovito, "VLSI Designs for Multiplication Over Finite Fields GF(2^m)", *Lecture Notes in Computer Science*, vol. 357, pp. 297–309, 1989.

[64] C. Koc and T. Acar, "Montgomery Multiplication in GF(2^k)", *Designs, Codes and Cryptography*, vol. 14, pp. 57–69, Apr. 1998.

[65] L. Erkök, M. Carlsson, and A. Wick, "Hardware/Software Co-verification of Cryptographic Algorithms using Cryptol", *in Proc. Formal Methods in CAD (FMCAD)*, pp. 188–191, 2009.

[66] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function Extraction from Arithmetic Bit-level Circuits", *ISVLSI*, 2014.

[67] S. Gao, "Counting Zeros over Finite Fields with Gröbner Bases", Master's thesis, Carnegie Mellon University, 2009.

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

[69] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear Loop Invariant Generation using Grobner Bases", *SIGPLAN Not.*, vol. 39, pp. 318–329, 2004.

[70] L. Lastras-Montaño, P. Meany, E. Stephens, B. Trager, J. O'Conner, and L. Alves, "A new class of array codes for memory storage", *in Proc. Information Theory and Applications Workshop*, pp. 1–10, 2011.

[71] L. Lastras, A. Lvov, B. Trager, S. Winograd, V. Paruthi, A. El-Zhein, R. Shadowen, and G. Janssen, "New Formal Verification Techniques for Algorithms over Finite Fields", Presented at Intl. Workshop on Internation Theory and Applications. Abstract of the paper available at: http://ita.ucsd.edu/workshop/12/talks, 2012.

[72] A. Lvov, L. Lastras-Montaño, V. Paruthi, R. Shadowen, and A. El-Zein, "Formal Verification of Error Correcting Circuits using Computational Algebraic Geometry", *in Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pp. 141–148, 2012.

[73] 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.

[74] 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.

[75] J. Lv, P. Kalla, and F. Enescu, "Efficient Groebner Basis Reductions for Formal Verification of Galois Field Multipliers", *in IEEE Design, Automation and Test in Europe*, 2012.

[76] J. Lv, P. Kalla, and F. Enescu, "Verification of Composite Galois Field Multipliers over GF($(2^m)^n$) using Computer Algebra Techniques", *in IEEE High-Level Design Validation and Test Workshop*, pp. 136–143, 2011.

[77] J. Lv, P. Kalla, and F. Enescu, "Formal Verification of Galois Field Multipliers using Computer Algebra", *in 25th IEEE International Conference on VLSI Design*, 2012.

[78] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Gruel, "An Algebraic Approach to Proving Data Correctness in Arithmetic Datapaths", *in Computer Aided Verification Conference*, pp. 473–486, 2008.

[79] N. Shekhar, P. Kalla, and F Enescu, "Equivalence Verification of Polynomial Datapaths using Ideal Membership Testing", *IEEE Transactions on CAD*, vol. 26, pp. 1320–1330, July 2007.

[80] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G.-M. Greuel, "STABLE: A New QBF-BV SMT Solver for Hard Verification Problems Combining Boolean Reasoning with Computer Algebra", *in IEEE Design, Automation and Test in Europe Conference*, pp. 155–160, 2011.

[81] R. Zippel, "Probabilistic algorithms for sparse interpolation", *in Proc. Symp. Symbolic and Algebraic Computation*, pp. 216–226, 1979.

[82] M. Ben-Or and P. Tiwari, "A deterministic algorithm for sparse multivariate polynomial interpolation", *in Proc. Symp. Theory of Computing*, pp. 301–309, 1988.

[83] S. Javadi and M. Monagan, "On sparse polynomial interpolation over finite fields", *in Intl. Symp. Symbolic and Algebraic Computing*, 2010.

[84] Z. Zilic and Z. Vranesic, "A deterministic multivariate interpolation algorithm for small finite fields", *IEEE Trans. Computers*, vol. 51, Sept. 2002.

[85] Robert J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, 1987.

[86] S. Roman, *Field Theory*, Springer, 2006.

[87] R. Lidl and H. Niederreiter, *Finite Fields*, Cambridge University Press, 1997.

[88] P. Montgomery, "Modular Multiplication Without Trial Division", *Mathematics of Computation*, vol. 44, pp. 519–521, Apr. 1985.

[89] H. Wu, "Montgomery Multiplier and Squarer for a Class of Finite Fields", *IEEE Transactions On Computers*, vol. 51, May 2002.

[90] 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.

[91] D. Singmaster, "On Polynomial Functions (mod m)", *J. Number Theory*, vol. 6, pp. 345–352, 1974.

[92] Z. Chen, "On polynomial functions from Z_n to Z_m", *Discrete Math.*, vol. 137, pp. 137–145, 1995.

[93] Z. Chen, "On polynomial functions from $Z_{n_1} \times Z_{n_2} \times \cdots \times Z_{n_r}$ to Z_m", *Discrete Math.*, vol. 162, pp. 67–76, 1996.

[94] ST Microelectronics, *ST23YLxx series Microcontroller for Smart Cards*.

[95] K. Kobayashi, *Studies on Hardware Assisted Implementation of Arithmetic Operations in Galois Field*, PhD thesis, Nagoya University, Japan, 2009.

[96] S. Morioka and Y. Katayama, "Design methodology for a one-shot reed-solomon encoder and decoder", *in IEEE International Conference on Computer Design*, pp. 60–67, 1999.

[97] Y. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede, "Elliptic-Curve-Based Security Processor for RFID", *IEEE Transactions on Computers*, vol. 57, pp. 1514–1527, Nov. 2008.

[98] Darrel Hankerson, Julio Hernandez, and Alfred Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields", in CetinK. Koc and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 1–24. Springer Berlin Heidelberg, 2000.

[99] V. Miller, "Use of Elliptic Curves in Cryptography", *in Lecture Notes in Computer Sciences*, pp. 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[100] B. A. Forouzan, *Cryptography and Network Security.*, A. R. Apt, 2008.

[101] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n)", *in Proceedings of the Selected Areas in Cryptography*, pp. 201–212, London, UK, UK, 1999. Springer-Verlag.

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

[103] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, PhD thesis, University of Innsbruck, 1965.

[104] 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.

[105] Jimmy K Omura and James L Massey, "Computational method and apparatus for finite field arithmetic", May 6 1986, US Patent 4,587,627.

[106] 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.

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

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

[109] 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.

[110] 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.

[111] 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.

[112] 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.

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

[114] 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.

[115] Z. Feng, Z. Zeng, and P. Li, "Parallel On-Chip Power Distribution Network Analysis on Multicore GPU Platforms", *IEEE Transactions VLSI*, 2011.

[116] Michael Brickenstein and Alexander Dreyer, "Polybori: A Framework for Gröbner Basis Computations with Boolean Polynomials", *Journal of Symbolic Computation*, vol. 44, pp. 1326–1345, September 2009.