Boolean Gröbner Basis Reductions on Datapath Circuits using the Unate Cube Set Algebra

Utkarsh Gupta, Priyank Kalla, Vikas Rao
Electrical & Computer Engineering, University of Utah

Abstract—**Recent developments in formal datapath verification make efficient use of symbolic computer algebra algorithms for formal verification. The circuit is modeled as a set of polynomials over Boolean (or pseudo-Boolean) rings, and Gröbner basis (GB) reductions are performed over these polynomials to derive a canonical representation. GB reductions of Boolean polynomials tend to cause intermediate expression swell (term explosion problem) – often making the approach infeasible in a practical setting. To overcome these problems, this paper describes a logic synthesis analogue of GB reductions over Boolean polynomials, using the unate cube set algebra over characteristic sets. By representing Boolean polynomials as characteristic sets using Zero-suppressed BDDs (ZBDDs), implicit algorithms can be efficiently designed for GB-reduction on digital circuits. We show that imposition of circuit-topology based monomial orders on ZBDDs enables an implicit implementation of polynomial division, canceling multiple monomials in one-step. Experiments performed over various finite field arithmetic architectures demonstrate the efficiency of our algorithms and implementations as compared to conventional explicit methods.**

I. Introduction

Automated formal verification and equivalence checking of arithmetic datapath circuits is challenging. Conventional verification techniques, such as those based on binary decision diagrams (BDDs) [1], And-Invert-Graph (AIG) based reductions with SAT or SMT-solvers [2], etc., are infeasible in verifying complex datapath designs. Such designs often implement algebraic computations over bit-vector operands, therefore finite integer rings [3] or finite fields [4] are considered appropriate models to devise decision procedures for verification. For this reason, the verification community has explored the use of algebraic geometry and symbolic algebra algorithms for verification. In such a setting, the circuit is modeled by way of a set of polynomials that generate an ideal, and the verification problem is formulated using Gröbner basis (GB) reduction techniques [5].

The GB problem exhibits high computational complexity. Indeed, computing a GB (using Buchberger's [6] or the F_4 algorithm [7]) for large circuits is practically infeasible. Managing this complexity ought be a major goal of any approach.

State-of-the-art & Limitations: Recent approaches [3] [8] have discovered that particularly for circuit verification problems, the expensive GB computation can be avoided altogether. For arbitrary combinational [3] [8] and sequential circuits [9], a specialized term order $>$ can be derived by analyzing the topology of the given circuit. This term order is derived by performing a reverse topological traversal of the circuit, and in this manuscript we refer to it as the *Reverse Topological Term Order* (RTTO). Imposition of RTTO $>$ on the polynomial ring *renders the set of polynomials of the circuit itself a GB.* Subsequently, the verification problems can be solved solely by way of GB-reduction (using multi-variate polynomial division), without any need to explicitly compute a GB. The techniques of [3] [8] have been extended and improved further to verify integer arithmetic circuits. For instance, [10] and [11] get more insights from the circuit structure that dictate specific rules on the order of polynomials chosen in GB-reduction – by accounting for topological levels, reconvergent fanouts, AND-XOR gates with common inputs, etc. The authors in [12] show that the reduction process can be parallelized by performing reduction for each output bit independently.

A common theme among all these relevant works is that *they move the complexity of verification from one of computing a GB to that of GB-reduction (multivariate polynomial division).* These will benefit greatly by a dedicated, domain-specific implementation of GB-reduction carried out on the given circuit under RTTO $>$. So far, the above techniques [3], [4], [8], [11], [10], [12] use a general-purpose polynomial division approach, together with explicit set representation, for this GB-reduction. While some of these approaches do perform the reduction in some specific ways – e.g., mimicking GB-reduction under RTTO $>$ by substitution [11], or using TEDs to perform input-output signature comparisons [10], or the use of F_4-style GB-reduction on a coefficient matrix [8] – the overall concept of polynomial division is still utilized in its rudimentary form, involving iterative cancellation of monomials "1-step at a time" on explicit data-structures. We show in the sequel, that despite recent efforts, such GB-reductions can still lead to *a worst-case size explosion problem.*

Proposed Solution: To make this GB-reduction on circuits more efficient, this paper describes new techniques and implementations, specifically targeted for circuit verification under RTTO $>$. In particular, we make use of implicit characteristic set representation of Zero-Suppressed BDDs [13]. By analyzing the structure of ZBDDs for polynomial representation under RTTO $>$, we show how this GB-reduction can be efficiently implemented using algorithms that specifically manipulate the ZBDD graph, by interpreting Boolean polynomial manipulation as the algebra of unate cube sets.

Rationale: The algebraic objects used to model the polynomial ideals derived from digital circuits are rings of Boolean polynomials. When Boolean functions are represented in $\mathbb{F}_2$ using AND/XOR expressions, and that too as a canonical Gröbner basis, the representation tends to explode. Polynomial representations employed in computer algebra tools, such as the *dense-distributive data-structure* of the SINGULAR computer algebra tool [14], are inefficient for this purpose. Since addition (mod 2) and multiplication are equivalent to XOR and AND operations, respectively, GB-reduction can be viewed as a specialized *AND/XOR Boolean function decomposition* problem. Clearly, implicit Boolean set representations such as decision diagrams could be employed for this purpose. The decision diagram of choice here is the ZBDD [13], because of its power to represent and manipulate sparse combinatorial problems – particularly "sets of combinations" using the unate cube set algebra framework.

Technical Contributions: We describe when and how the GB-reduction encounters a term-explosion (exponential blow-up) under RTTO $>$, which cannot be easily overcome by explicit representations. We show that ZBDDs can avoid this exponential blow-up – thereby justifying their use. We describe how the rudimentary polynomial division algorithms, that iteratively cancel one monomial in every step, can be implemented on ZBDDs under RTTO $>$. Subsequently, *we show that RTTO $>$ imposes a special structure on ZBDDs that allows us to cancel multiple monomials in every step of polynomial division*, thus improving GB-reduction in both space and time! Finally, experiments conducted on finite field arithmetic (crypto-circuits) benchmarks show an order of magnitude improvement using our implementation of GB-reduction.

Relationship to prior work in Boolean Gröbner Basis: The symbolic algebra community has studied properties of Boolean GB [15] [16] [17]. From among these, the work of PolyBori [17] comes closest to ours, and is a source of inspiration for this work. PolyBori proposed the use of ZBDDs to compute Gröbner bases for Boolean polynomials. PolyBori is a *generic* Boolean GB computational engine that caters to many permissible term orders. Its division algorithm is also based on the conventional concept of canceling one monomial in every step of reduction. In contrast, our algorithms are tailored for GB-reduction under the RTTO $>$. The efficiency of our approach stems from the observation that the RTTO $>$ imposes a special structure on the ZBDDs, which allows for multiple monomials to be canceled in one division-step.

II. Preliminaries: Notation and Background

A. Computer Algebra

This section provides a brief description of the fundamental concepts of commutative algebra including polynomial rings, polynomial division, ideals, Gröbner basis and their application in verification of circuits.

Let $\mathbb{B} = \{0,1\}$ denote the Boolean domain, $\mathbb{F}_2$ the finite field of 2 elements ($\mathbb{B} \equiv \mathbb{F}_2$), and $R = \mathbb{F}_2[x_1,\dots,x_n]$ denote the polynomial ring over variables $x_1,\dots,x_n$ with coefficients in $\mathbb{F}_2$. Operations in $\mathbb{F}_2$ are performed $\pmod 2$, so $-1 = +1$ in $\mathbb{F}_2$. We will use $+,\cdot$ to denote addition and multiplication in R, and $\vee,\wedge$ and $\oplus$ to denote Boolean OR, AND and XOR operations, respectively.

A polynomial $f \in R$ is written as a finite sum of terms $f = c_1X_1 + c_2X_2 + \cdots + c_tX_t$. Here $c_1,\dots,c_t$ are coefficients and $X_1,\dots,X_t$ are monomials, i.e. power products of the type $x_1^{e_1} \cdot x_2^{e_2} \cdots x_n^{e_n}$, $e_i \in \mathbb{Z}_{\geq 0}$. To systematically manipulate the polynomials, a monomial order $>$ (also called a term order) is imposed on the ring such that $X_1 > X_2 > \cdots > X_t$. Subject to $>$, $lt(f) = c_1X_1$, $lm(f) = X_1$, $lc(f) = c_1$, are the *leading term*, *leading monomial* and *leading coefficient* of f, respectively. We also denote tail(f) = $f - lt(f) = c_2X_2 + \cdots + c_tX_t$. In this work, we are mostly concerned with terms ordered lexicographically (*lex*).

Definition II.1. *Let $f = c_1X_1 + \cdots + c_tX_t$ be a polynomial in $\mathbb{F}_2[x_1,\dots,x_n]$ such that the coefficients $c_i \in \{0,1\}$, and monomials $X = x_1^{e_1} \cdot x_2^{e_2} \cdots x_n^{e_n}, e_i \in \{0,1\}$. Then f is called a* **Boolean polynomial**. *For Boolean polynomials $lt(f) = lm(f)$.*

A gate-level circuit can be modeled with Boolean polynomials, where every Boolean logic gate operator is mapped from $\mathbb{B}$ to a polynomial function over $\mathbb{F}_2$:

$$\begin{aligned} z &= \neg a \rightarrow z + a + 1 \pmod 2 \\ z &= a \wedge b \rightarrow z + a \cdot b \pmod 2 \\ z &= a \vee b \rightarrow z + a + b + a \cdot b \pmod 2 \\ z &= a \oplus b \rightarrow z + a + b \pmod 2 \end{aligned} \tag{1}$$

Polynomial Reduction via division: Let f,g be polynomials. If $lt(f)$ is divisible by $lt(g)$, then we say that f *is reducible to* r modulo g, denoted $f \xrightarrow{g} r$, where $r = f - \frac{lt(f)}{lt(g)} \cdot g$. Similarly, f can be *reduced w.r.t. a set of polynomials* $F = \{f_1,\dots,f_s\}$ to obtain a remainder r. This reduction is denoted as $f \xrightarrow{F}_+ r$, and the remainder r has the property that no term in r is divisible by the leading term of any polynomial f_i in F. Algorithm 1 shows the step-by-step procedure to perform this classical reduction.

Algorithm 1 Multivariate Reduction of f by $F = \{f_1,\dots,f_s\}$

```
1:  procedure multi_variate_division($f, \{f_1,\dots,f_s\}, f_i \neq 0$)
2:      $u_i \leftarrow 0$;  $r \leftarrow 0$,  $h \leftarrow f$
3:      while  $h \neq 0$  do
4:          if  $\exists i$ s.t. $lm(f_i) \mid lm(h)$ then
5:              choose $i$ least s.t. $lm(f_i) \mid lm(h)$
6:              $u_i = u_i + \frac{lt(h)}{lt(f_i)}$
7:              $h = h - \frac{lt(h)}{lt(f_i)} f_i$
8:          else
9:              $r = r + lt(h)$
10:             $h = h - lt(h)$
11:     return  $(\{u_1,\dots,u_s\}, r)$
```

The algorithm initializes h with the polynomial f and cancels its leading term by some polynomial f_i. If the leading term $lt(h)$ cannot be canceled by any $lt(f_i)$, then it is added to the final remainder r and process is repeated until all the terms in h are analyzed.

Polynomial Ideals: Given a set of polynomials $F = \{f_1,\dots,f_s\} \in \mathbb{F}_2[x_1,\dots,x_n]$, denote the ideal J generated by F as $J = \langle F \rangle = \langle f_1,\dots,f_s \rangle = \{\sum_{i=1}^{s} h_i \cdot f_i : \ h_i \in R\}$. The ideal J may have many different generators, i.e. it is possible to have $J = \langle f_1,\dots,f_s \rangle = \langle g_1,\dots,g_t \rangle = \cdots = \langle h_1,\dots,h_r \rangle$. A Gröbner basis G of ideal J is one such set of polynomials $G = GB(J) = \{g_1,\dots,g_t\}$ that is a canonical representation of the ideal.

Definition II.2. [**Gröbner Basis**] *[5]: For a monomial ordering $>$, a set of non-zero polynomials $G = \{g_1, g_2, \cdots, g_t\}$ contained in an ideal J, is called a Gröbner basis of J iff*

$\forall f \in J,\ f \neq 0$, there exists $i \in \{1,\cdots,t\}$ such that $lm(g_i)$ divides $lm(f)$.

Gröbner basis G of an ideal $J = \langle f_1,\ldots,f_s\rangle$ is computed using the Buchberger's algorithm [6]. The algorithm initializes the set G with the given generators of J *i.e.* $\{f_1,\ldots,f_s\}$. The algorithm is based on the computation of *Spoly* of pairwise combination of polynomials in G using the following formula,

$$Spoly(f_i,f_j) = \frac{L}{lt(f)}\cdot f - \frac{L}{lt(g)}\cdot f \tag{2}$$

where $L = LCM(f_i,f_j)$. The *Spoly* is then reduced *w.r.t.* the polynomials in G: $Spoly \xrightarrow{G}_{+} h$. If h is non-zero, it is added to G. The algorithm terminates when there are no new non-zero h generated from the set G. The idea of the *Spoly* reductions is to cancel leading terms of polynomials $\{f_i,f_j\}$ to obtain polynomials with new leading terms which provide additional information regarding the basis.

Definition II.3. [**Gröbner Basis Reduction**] *[5]: Let $G = \{g_1,\ldots,g_t\}$ be a Gröbner basis of ideal J, and let f be another polynomial. Then the remainder r obtained by reduction of f modulo G, denoted $f \xrightarrow{G}_{+} r$, is called the* **Gröbner basis reduction (GBR)** *of f. Moreover, the remainder r so obtained by GBR of f is a canonical expression modulo G.*

Proposition II.1. *Given a circuit C, we can represent all the gates using (Boolean) polynomials $F = \{f_1,\ldots,f_s\}$ in $\mathbb{F}_2[x_1,\ldots,x_n]$ by means of Eqn. (1), s.t. ideal $J = \langle F\rangle$. Let z_i, $i = 0,\ldots,k-1$ denote the k-bit primary output variables of the circuit. Compute a Gröbner basis $G = GB(J) = \{g_1,\ldots,g_t\}$ for the polynomials of the circuit, and perform the GBR $z_i \xrightarrow{G}_{+} r_i$ for all $0 \leq i < k$. Then all r_i's are a canonical representation and can be used for formal verification/equivalence checking.*

This verification requires the computation of a Gröbner basis. The Buchberger's algorithm for computation of Gröbner basis has high complexity ($2^{(O(n))}$ in our setting). The work of [8] showed that the GB computation can be avoided.

Definition II.4. [**Product Criterion**] *[18]: For two polynomials f_i,f_j in any polynomial ring R, if the equality $lt(f_i)\cdot lt(f_j) = LCM(lt(f_i),lt(f_j))$ holds, then $Spoly(f_i,f_j) \xrightarrow{G}_{+} 0$.*

Using this criterion we can say that when the leading terms of all polynomials in the basis $F = \{f_1,\ldots,f_s\}$ are relatively prime, then F is already a Gröbner basis ($F = GB(J)$). For a combinational circuit C, a term order $>$ can be derived by analyzing the circuit topology which ensures this property is true [3] [8]:

Proposition II.2. *(From [8]) Let C be any arbitrary combinational circuit. Let $\{x_1,\ldots,x_n\}$ denote the set of all variables (signals) in C. Starting from the primary outputs, perform a reverse topological traversal of the circuit and order the variables such that $x_i > x_j$ if x_i appears earlier in the reverse topological order. Impose a lex term order $>$ to represent each gate as a polynomial f_i, s.t. $f_i = x_i + tail(f_i)$. Then the set of all polynomials $\{f_1,\ldots,f_s\}$ forms a Gröbner basis G, as $lt(f_i) = x_i$ and $lt(f_j) = x_j$ for $i \neq j$ are relatively prime. This term order $>$ is called the* **Reverse Topological Term Order (RTTO)**.

Subsequently, by imposing RTTO on the polynomials of the circuit, the explosive GB computation is avoided, and verification is performed by the canonical GB-reduction: $z_i \xrightarrow{G}_{+} r_i$.

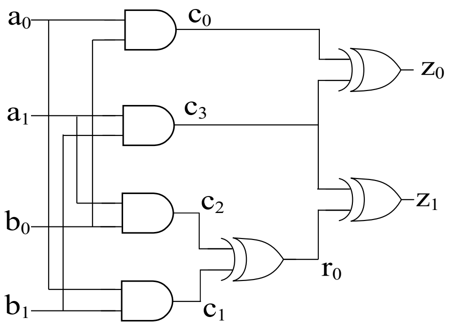

Fig. 1: A 2-bit modulo Multiplier circuit.

$$f_1 : c_0 + a_0\cdot b_0,\ lm = c_0; \quad f_2 : c_1 + a_0\cdot b_1,\ lm = c_1$$
$$f_3 : c_2 + a_1\cdot b_0,\ lm = c_2; \quad f_4 : c_3 + a_1\cdot b_1,\ lm = c_3$$
$$f_5 : r_0 + c_1 + c_2,\ lm = r_0; \quad f_6 : z_0 + c_0 + c_3,\ lm = z_0$$
$$f_7 : z_1 + r_0 + c_3,\ lm = z_1$$

Fig. 2: Polynomials of the circuit under RTTO constitute a GB.

Example II.1. Demonstration of the approach: *Consider the circuit given in Fig. 1. Impose RTTO on the circuit. The primary outputs z_0,z_1 are both at level-0, variables r_0,c_0,c_3 are at level-1, c_1,c_2 are at level-2, and the primary inputs a_0,a_1,b_0,b_1 are at level-3. Order the variables $\{z_0 > z_1\} > \{r_0 > c_0 > c_3\} > \{c_1 > c_2\} > \{a_0 > a_1 > b_0 > b_1\}$. Using this variable order, we impose a lex term order on the monomials. Then all the polynomials extracted from the circuit have relatively prime leading terms, as shown in Fig. 2, and $F = \{f_1,\ldots,f_7\}$ forms a GB.*

Then the GBRs $z_1 \xrightarrow{F}_{+} a_0\cdot b_0 + a_1\cdot b_1$ and $z_0 \xrightarrow{F}_{+} a_0\cdot b_1 + a_1\cdot b_0 + a_1\cdot b_1$ are canonical expressions of the output bits.

B. Unate Cube Sets & Boolean Polynomials

A Boolean *variable* represents a dimension of the Boolean space $\mathbb{B}^n$, a *literal* is an instance of a variable x_i or its complement $\neg x_i$. A *cube* is a product of literals which denotes a set of points in the Boolean space. A *cube set* consists of a number of cubes, each of which is a combination of literals. *Unate cube sets* allow the use of only positive literals, not negative/complemented literals. Each cube in a unate cube set represents a combination, and each literal represents an object selected in the combination.

When cube sets are used to represent Boolean functions, they are usually *binate* cube sets containing negative literals. In binate cube sets, literals x_i and $\neg x_i$ represent $x_i = 1$ and $x_i = 0$, respectively; while the absence of a literal implies a *don't care*. In unate cube sets, literal x_i implies $x_i = 1$ whereas its absence implies $x_i = 0$. For example, the cube set $\{a, bc\}$ represents $(abc) : \{111, 110, 101, 100, 011\}$ in the binate cube set representation, whereas it represents $(abc) : \{100, 011\}$ in the unate cube set representation.

Each monomial of a Boolean polynomial can be viewed as a unate cube – a product of positive literals – and a Boolean polynomial as a unate cube set. Then the GBR $z_i \xrightarrow{G}_+ r_i$ can be interpreted as algebra over unate cube sets, resembling a classical logic synthesis problem, as shown below. Let us (re)consider the one-step division for Boolean polynomials: $f \xrightarrow{g} r$. This division is implemented as:

$$\begin{aligned} f \xrightarrow{g} r &= f - \frac{lt(f)}{lt(g)} \cdot g \;=\; f - \frac{lm(f)}{lm(g)} \cdot g \\ &= f + \frac{lm(f)}{lm(g)} \cdot g \;=\; f \oplus \frac{lm(f)}{lm(g)} \wedge g \end{aligned} \tag{3}$$

We can replace $lt(f)$ with $lm(f)$ as coefficients are either 0 or 1. Notice that $\frac{lm(f)}{lm(g)}$ is a unate product of literals. i.e. a unate cube. The $\oplus$ operation cancels common cubes from f and $\frac{lm(f)}{lm(g)} \cdot g$.

C. Zero Suppressed Binary Decision Diagrams (ZBDDs)

A ZBDD [13] can be obtained from a BDD by eliminating all the nodes whose 1-edge points to 0 terminal node and by sharing all the isomorphic sub-graphs for two nodes. Given the order of the variables, a ZBDD represents a Boolean function canonically.

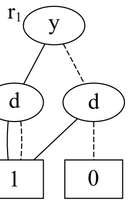

Fig. 3: ZBDD for the polynomial $r_1 = yd + y + d$.

In [19] *Minato* demonstrated that ZBDDs are an efficient data-structure for implicit manipulation (algebra) of unate cube sets. Fig. 3 is a ZBDD for the unate cube set $\{yd, y, d\}$ with the variable order $y > d$. The paths beginning from the root node y and terminating in the 1-terminal node are the cubes of the set. A variable is in a cube if its 1-edge is in the path and is not in the cube if its 0-edge is in the path. The ZBDD can also be interpreted as a polynomial $r_1 = yd + y + d$ where the monomials can obtained the same way we obtain the cubes for the equivalent set.

Based on the above discussion, we will: i) model GBR as the algebra of unate cube sets; ii) use ZBDDs as the implicit data-structure for this GBR; and iii) devise efficient implementation of the GBR by exploiting the special structure imposed by RTTO on the ZBDD graph. For details on the use of unate cube set algebra in classical logic synthesis, and its implementation on ZBDDs, we refer the reader to [13] [19].

III. Theory and Algorithms

Consider the circuit in Fig. 4, impose RTTO: *lex* term order with variable ordering as, $z > y > x > d > c > b > a$. The Boolean polynomials for the circuit are: $f_1 = z + y \cdot d + y + d$, $f_2 = y + x \cdot c + x + c$, $f_3 = x + b \cdot a + b + a$. Under RTTO,

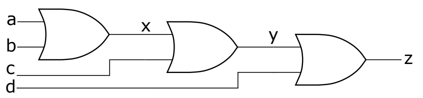

Fig. 4: A chain of OR gates.

f_1, f_2, f_3 forms a GB $G = \langle f_1, f_2, f_3 \rangle$. For verification, we have to reduce the output z modulo G. A classical symbolic algebra reduction using an explicit representation is carried out as:

1) $z \xrightarrow{f_1} yd + y + d$
2) $yd + y + d \xrightarrow{f_2} y + xdc + xd + dc + d \xrightarrow{f_2} xdc + xd + xc + x + dc + d + c$
3) $xdc + xd + xc + x + dc + d + c \xrightarrow{f_3} xd + xc + x + dcba + dcb + dca + dc + d + c \xrightarrow{f_3} xc + x + dcba + dcb + dca + dc + dba + db + da + d + c \xrightarrow{f_3} x + dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c \xrightarrow{f_3} dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c + ba + b + a = r$

In the first step, z is reduced by f_1 just *once* as that's the only term. In the second step, the result of step one is reduced *twice* by f_2 as the result has two terms containing variable y. Similarly, *four* reductions by f_3 are required to reduce the result of step two into an expression containing only primary inputs (which cannot be reduced further).

Observations: i) Notice that the size of the final remainder corresponds to that of the worst case of a Boolean polynomial: i.e. r contains $2^n - 1$ $(= 15)$ monomial terms for n $(= 4)$ variables. ii) Classical division algorithms reduce the polynomials 1-step at a time, where only one monomial is canceled in each step. iii) The number of 1-step reductions can increase exponentially as GBR progresses across the circuit.

It is clear that any data-structure that *explicitly* represents each monomial will encounter space and time explosion: this includes the dense-distributive representation of SINGULAR computer algebra tool [14], or the ones used by [10], [11]. The F_4-style polynomial reduction of [8], [4] simulates division on a matrix M representing the problem. However, each column of M corresponds to monomial generated in the division process, therefore [8], [4] also encounter this size explosion.

The use of ZBDDs can help overcome this explosion. Fig. 5 shows the same reduction of z by f_1, f_2, f_3 using ZBDDs (exact procedure discussed later). The size of the ZBDDs after complete reduction by f_1, f_2, f_3 increases linearly in the number of nodes. Subsequently, the final remainder has $2 \cdot n - 1$ $(= 7)$ nodes (excluding the terminal 1 and 0 nodes) for n $(= 4)$ variables.

ZBDD Representation: The following steps describe the procedure for building ZBDDs for the polynomials of the gates of circuits.

1) Obtain the RTTO for the variables (signals) of the circuits as $x_1 > x_2 > \cdots > x_n$.
2) Impose the same order on the ZBDDs.
3) Declare ZBDDs for each of these variables.
4) Use Eqn. (1) to model the gates of the circuit as Boolean polynomials. Build ZBDDs for these polynomials using the $+$ and $\cdot$ binary operations for modulo 2 sum and product of variables. The $+$ operation can be implemented as $f + g = f \cup g - f \cap g$. However, in order to avoid the large

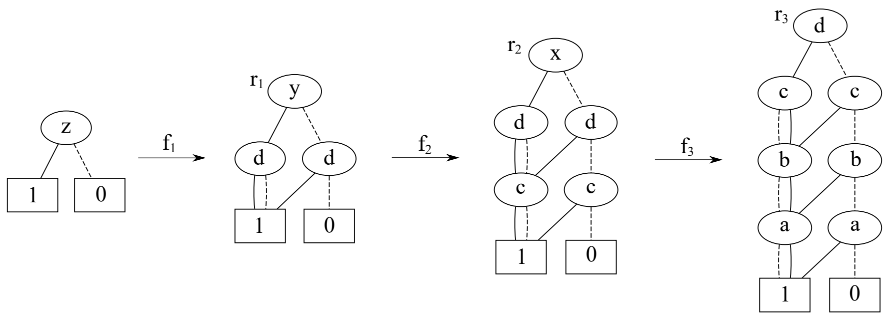

Fig. 5: Reduction of output of the circuit in Fig. 4 by f_1, f_2, f_3.

intermediate ZBDDs for the union we have implemented this operation as presented in Algorithm 1 in [17].

5) Traversing only the solid edges from the root node of a ZBDD to terminal **1** delivers the leading monomial of that polynomial. The child node of the root at the solid edge's end will be referred to as *then* and the other child as *else*.

Once the ZBDDs for the circuit have been built and stored in G, we need to perform the reduction $z_i \xrightarrow{G}_+ r_i$ for each output bit z_i. The polynomial r_i will be a canonical representation of z_i in terms of primary inputs only.

Consider the step 2 of division corresponding to Fig. 4, where the polynomial $r_1 = yd + y + d$ needs to be reduced by f_2. The ZBDDs for r_1 and f_2 are shown in Fig. 6. Checking if $lt(f_2)$ divides $lt(r_1)$ becomes trivial as we just need to compare the indices (each variable has a unique index) of top-most nodes of ZBDDs for the polynomials r_1 and f_2, which in this case are equal. Recall from Proposition II.2 that the $lt(f_i)$ of the polynomials for gates of the circuit will always be x_i.

Division with ZBDDs: Cancel 1 monomial in every step

The algorithm for conventional reduction procedure using ZBDDs is shown in Algorithm 2. The input parameters are the ZBDD of the output bit of the circuit z_i and *poly_list* containing the ZBDDs for the set of polynomials corresponding to the gates of the circuit. The algorithm is based on the classical division procedure (Algorithm 1).

Due to RTTO, the circuit polynomials for each gate are represented as $f_1 = x_1 + else(f_1), \dots, f_s = x_s + else(f_s)$ with variable order $x_1 > \cdots > x_s > \cdots > x_n$ (Prop. II.2). Note that the variables $\{x_{s+1}, \dots, x_n\}$ are primary inputs and are not the output of any logic gate. Then the elements in *poly_list* are ordered $f_1 > f_2 > \cdots > f_s$ *i.e.* $poly\_list[1] = f_1, poly\_list[2] = f_2$ and so on. Populating *poly_list* in this way avoids the search (Line 4, Algorithm 1) required to find a polynomial $g \in poly\_list$ that can divide the leading term of z_i. While iterating over the polynomials $g \in poly\_list$ if a certain polynomial does not divide the leading term of z_i, it will imply the polynomial is not in the logical cone of z_i.

The procedure $leading\_term(g)$ returns the leading term of the ZBDD representation of polynomial g. If g divides f, then the procedure $ZBDD\_Divide(f, g)$ (performs cube division) returns the quotient of the division, else it returns zero. Line 8 iteratively computes $z_i = z_i + \frac{lt(z_i)}{lt(g)} \cdot g$. The polynomial z_i is completely reduced *w.r.t.* the polynomial g in the while loop.

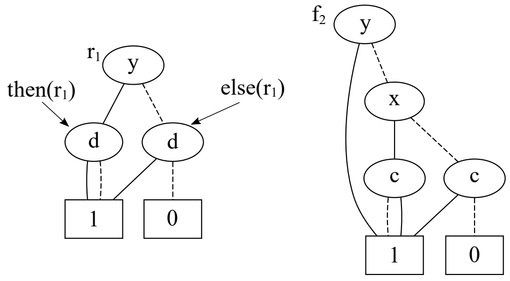

Fig. 6: ZBDDs for polynomial r_1 and f_2.

Algorithm 2 Reduction: Cancel 1 monomial every iteration

```
1: procedure $single\_mon\_red(z_i, poly\_list)$
2:     for each $g \in poly\_list$ do
3:         $lead\_g = leading\_term(g)$
4:         $lead\_z_i = leading\_term(z_i)$
5:         $quotient = ZBDD\_Divide(lead\_z_i, lead\_g)$
6:         while $quotient \neq zero$ do
7:             $prod = quotient \cdot g$
8:             $z_i = z_i + prod$
9:             $lead\_z_i = leading\_term(z_i)$
10:            $quotient = ZBDD\_Divide(lead\_z_i, lead\_g)$
11:     return $z_i$
```

Improved Reduction: Cancel multiple monomials in 1 step: Next, we will show how z_i can be reduced by a polynomial g in one step. In the example of Fig. 5, the primary output z is reduced by f_1 to get r_1. The next step is to reduce r_1 by f_2 to get r_2. To demonstrate our approach we will show how the reduction of r_1 by f_2 can be achieved in one step.

The polynomial $r_1 = yd + y + d$ can be written as $y \cdot (d + 1) + d$. If we perform 1-step reduction of r_1 by f_2 we get the quotient $d + 1$. This quotient is visible as the polynomial represented by the *then*-node of r_1 (Fig. 6). So the reduction can be performed by multiplying $d + 1$ with f_2 and adding this product to $r_1 \pmod 2$:

$$\begin{aligned}&(yd + y + d) + (d + 1) \cdot (y + xc + x + c) \pmod 2\\&= 2 \cdot (yd + y) + d + (d + 1) \cdot (xc + x + c) \pmod 2\\&= d + (d + 1) \cdot (xc + x + c) \pmod 2\end{aligned}$$

Notice that $else(r_1) = d$ and $else(f_2) = xc + x + c$. In addition, we know that $2 \cdot (fd + f) \pmod 2$ is going to be zero. Therefore, in order to reduce number of operations, we directly use the last step as a formula for reduction:

$$\begin{aligned} r_1 \xrightarrow{f_2}_{+} &= d + (d+1)\cdot(xc+x+c) \\ & else(r_1) + then(r_1)\cdot else(f_2) \end{aligned}$$

So the reduction process effectively involves just two operations, a modulo 2 sum and a product. *This has the effect of canceling all the terms in r_1 that can be canceled by $lt(f_2)$ in one-go, implicitly canceling multiple monomials in one step.*

The algorithm for Multiple Monomial Reduction is shown in Algorithm 3, where the notations, z_i and *poly_list*, are same as in Algorithm 2. Unlike in Algorithm 2, however, where we need to find the quotient of $lead\_z_i/lead\_g$, Algorithm 3 only determines if $lead\_g$ can divide z_i at all (in this case the quotient is $then(z_i)$). This can be accomplished by just comparing the indices of top-most nodes of z_i and g. This algorithm significantly reduces the number of iterations, which now exactly equals the size of *poly_list*. For the example of Fig. 4, the number of iterations is 3 using Algorithm 3, whereas 7 iterations are required using Algorithm 2.

Algorithm 3 Reduction: Cancel multiple monomials

```
1: procedure $multi\_mon\_red(z, poly\_list)$
2:     for each $g \in poly\_list$ do
3:         if $index(g) == index(z)$ then
4:             $z = else(z) + then(z)\cdot else(g)$
5:     return $z$
```

We have implemented the above GBR procedures directly using the CUDD package [20]. The circuit under verification is analyzed, RTTO based variable order is imposed on the ZBDDs, and the Boolean polynomials of the circuit are represented as unate cube sets. The polynomials of of the gates of circuit, $f_i \in G$, are inserted in *poly_list* according to the variable order $x_1 > \cdots > x_i > \cdots > x_n$, where $f_i = x_i + else(f_i)$ (this is due to Prop. II.2). To perform GBR $z_i \xrightarrow{G}_{+} r_i$ Algorithm 3 is invoked to obtain the remainder.

IV. Experimental Results

This section presents the results of using our implementation (Algorithm 3) for reducing circuits used in cryptography. We compare our results against F4-style reduction [4], parallelized approach for performing reductions on Galois field multipliers [12], and PolyBori [17] that uses the conventional reduction procedure on top of ZBDDs. The experiments are performed on a 3.5GHz Intel Core^{TM} i7-4770K Quad-Core CPU with 32 GB of RAM.

A. Mastrovito Multipliers

Modular multiplication is an important computation used in cryptography. A Mastrovito multiplier architecture can be employed for performing this computation. Mastrovito multipliers compute $Z = A \times B \pmod P$ where P is a given primitive polynomial for the datapath size k.

The product $A \times B$ is computed using an array multiplier architecture, and then the result is reduced modulo P. The following example demonstrates the Mastrovito multiplier computation [8].

Example IV.1. *Consider the field $\mathbb{F}_{2^4}$. Let the inputs be: $A = a_0 + a_1\cdot\alpha + a_2\cdot\alpha^2 + a_3\cdot\alpha^3$ and $B = b_0 + b_1\cdot\alpha + b_2\cdot\alpha^2 + b_3\cdot\alpha^3$, and the irreducible polynomial be $P(x) = x^4 + x^3 + 1$. The coefficients of $A = \{a_0, \dots, a_3\}, B = \{b_0, \dots, b_3\}$ are in $\mathbb{F}_2 = \{0, 1\}$. First, we perform the multiplication as:*

$$\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
\times & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & a_3\cdot b_0 & a_2\cdot b_0 & a_1\cdot b_0 & a_0\cdot b_0 \\
 & & a_3\cdot b_1 & a_2\cdot b_1 & a_1\cdot b_1 & a_0\cdot b_1 & \\
 & a_3\cdot b_2 & a_2\cdot b_2 & a_1\cdot b_2 & a_0\cdot b_2 & & \\
a_3\cdot b_3 & a_2\cdot b_3 & a_1\cdot b_3 & a_0\cdot b_3 & & & \\
\hline
s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}$$

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, $s_0 = a_0\cdot b_0$, $s_1 = a_0\cdot b_1 + a_1\cdot b_0$, $s_2 = a_0\cdot b_2 + a_1\cdot b_1 + a_2\cdot b_0$, and so on. Here the multiply "$\cdot$" and add "$+$" operations are performed modulo 2, and hence implemented in a circuit using AND and XOR gates. As the coefficients are always reduced modulo $p = 2$, there are no carry-chains in the design. Next, the result is reduced modulo the primitive polynomial $P(x) = x^4 + x^3 + 1$, as:

$$\begin{array}{|cccc|l}
s_3 & s_2 & s_1 & s_0 & \\
\hline
s_4 & 0 & 0 & s_4 & s_4\cdot\alpha^4 \pmod{P(\alpha)} = s_4\cdot(\alpha^3+1) \\
s_5 & 0 & s_5 & s_5 & s_5\cdot\alpha^5 \pmod{P(\alpha)} = s_5\cdot(\alpha^3+\alpha+1) \\
s_6 & s_6 & s_6 & s_6 & s_6\cdot\alpha^6 \pmod{P(\alpha)} = s_6\cdot(\alpha^3+\alpha^2+\alpha+1) \\
\hline
z_3 & z_2 & z_1 & z_0 &
\end{array}$$

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

Table I provides the results for the reductions $z_i \xrightarrow{G}_{+} r_i$ for Mastrovito multipliers for each output bit z_i. The benchmarks are taken from [8] which are optimized using ABC [21] with the same commands and library as mentioned in [12]. Algorithm 3 reduces each output bit independent of other bits. Therefore, we have presented the results obtained by running our reduction algorithm both parallely and sequentially for each output bit. Similarly, the results for implementation in PolyBori are also presented for both cases. The maximum number of parallel processes is decided by the memory usage of each process (*i.e.* reducing one bit) for our implementation and the total available memory. The larger benchmarks are run with fewer parallel processes as they consume more memory.

In the table, the column #T represents the number of parallel processes. (S) and (P) refer to the cases when the experiments are run sequentially and parallely for the output bits z_i, respectively.

TABLE I: Mastrovito Multipliers (Time in seconds); k = Datapath Size, #Gates = No. of gates, #T = No. of threads, Time-Out = 30 hrs, (P): Parallel Execution, (S): Sequential Execution, K = 10^3, M = 10^6, PB: PolyBori, ZR: Algorithm 3

k	**#Gates**	**F4 [4]**	**#T**	**[12](P)**	**PB**		**ZR**	
					(P)	**(S)**	**(P)**	**(S)**
64	11.5K	1.3	20	3.70	3.60	2.21	0.73	**0.27**
128	46K	9.89	20	27.54	23.99	16.76	5.08	**1.63**
163	73.5K	32.61	20	55.96	48.67	33.72	11.41	**3.11**
233	122K	86.30	20	127.61	112.96	77.23	21.77	**3.63**
283	193K	274.68	20	253.05	227.77	157.45	49.89	**11.41**
409	386K	2,528.48	10	716.80	659.64	426.92	163.52	**17.68**
571*	1.6M	TO	3	5,331	CR	CR	2,126.65	**566.39**

The 571-bit multiplier could not be synthesized and mapped with the given memory. Therefore, we have provided results for a structured (but not optimized) 571-bit multiplier benchmark. Our implementation outperforms the explicit approaches of [4] and [12] for Mastrovito multipliers. For the 571-bit multiplier, the implementation of [4] does not finish for the given time period of 30 hours and the PolyBori implementation crashes (CR).

An interesting point to note in the Table I is that our implementation takes less time when we are running it sequentially. There is a certain overhead involved when we declare variables and build ZBDDs for each gate of the circuit. In the case of Mastrovito multipliers benchmarks, this overhead is substantially greater than the reduction time for each output bit. Therefore, when we run these benchmarks parallely this overhead time hampers the overall run time.

B. Montgomery Multipliers

Exponentiation (repeated multiplication) is often required in cryptosystems. For such applications, Montgomery architecture [22] [23] [24] are considered more efficient than Mastrovito multipliers as they do not require explicit reduction modulo P after each step. Fig. 7 shows the structure of a Montgomery multiplier. Each MR block computes $A \cdot B \cdot R^{-1}$, where R is selected as a power of a base (α^k) and R^{-1} is the multiplicative inverse of R in $\mathbb{F}_{2^k}$. As this operation cannot compute $A \cdot B$ directly, we need to pre-compute $A \cdot R$ and $B \cdot R$ as shown in the Fig. 7. We denote the leftmost two blocks as Block A (upper) and B (lower), the middle block as Block C and the output block as Block D.

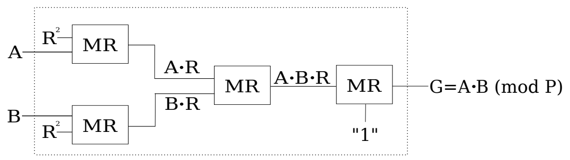

Fig. 7: Montgomery multiplication.

Table II provides the results for flattened (bit-blasted) and optimized Montgomery multipliers for the sequential and parallel executions. The 571-bit benchmark in the table is an unoptimized structured benchmark. We again get a significant improvement over the explicit approaches except for the case of 283 bit multiplier.

TABLE II: Montgomery Multipliers (Time in seconds); k = Datapath Size, #Gates = No. of gates, #T = No. of threads, Time-Out = 30 hrs, (P): Parallel Execution, (S): Sequential Execution, K = 10^3, M = 10^6, PB: PolyBori, ZR: Algorithm 3

k	**#Gates**	**F4 [4]**	**#T**	**[12](P)**	**PB**		**ZR**	
					(P)	**(S)**	**(P)**	**(S)**
64	9.5K	16.29	20	10.69	6.27	9.22	**3.75**	8.37
128	35K	621.90	20	36.19	28.93	34.59	**13.76**	24.73
163	56.5K	2,608.4	20	204.94	167.73	335.24	**141.68**	321.60
233	111K	385.92	20	132.51	119.77	99.36	42.16	**31.88**
283	165K	5,344	20	**704.13**	1,194.2	2,078.1	1,065.3	2,113.0
409	340K	7,104	10	697.91	737.23	722.05	303.91	**299.92**
571*	1.97M	TO	3	TO	CR	CR	**43,813**	99,042

Table III presents the statistics for hierarchical Montgomery multipliers for the blocks A, B, C, and D. The experiment first reduces the outputs of a block modulo the gates of that block, and then reduces the primary outputs modulo these four sets of remainders (ZBDDs), thus exploiting the hierarchy of these circuits. Table III shows the time for reduction of each block and the time for reducing the primary outputs across the four levels. The time for reducing the primary outputs across levels in case of F4 implementation is <1 second, and is not explicitly mentioned in the table. The row labeled *Total* presents the sum of time of reduction across levels and the maximum reduction time for each block (as the reductions for the four levels are independent of each other).

TABLE III: Montgomery Blocks (Time in seconds); k = Datapath Size, #Gates = No. of gates, Time-Out = 30 hrs, Red. = time for reduction, Coll. = time to reduce across the 4 levels. K = 10^3, M = 10^6, PB: PolyBori, ZR: Algorithm 3

k	**#Gates**	**Block**	**F4 [4]**	**PB**		**ZR**	
				Red.	Coll.	Red.	Coll.
163	33K	Block A	25	12	16	1	18
	33K	Block B	25	12		1	
	85K	Block C	73	18		7	
	32K	Block D	24	12		1	
	Total		73	34		**25**	
233	55K	Block A	142	32	5	0.14	4
	55K	Block B	141	33		0.14	
	163K	Block C	408	34		2.1	
	54K	Block D	140	32		0.13	
	Total		408	39		**6.1**	
283	82K	Block A	330	79	26	24	90
	82K	Block B	329	78		23	
	241K	Block C	883	173		118	
	81K	Block D	321	80		23	
	Total		883	**199**		208	
409	168K	Block A	1,322	177	28	0.57	29
	168K	Block B	1,335	175		0.57	
	502K	Block C	4,471	192		14	
	168K	Block D	1,338	176		0.56	
	Total		4,471	220		**43**	
571	330K	Block A	5,371	769	1,341	321	1,412
	330K	Block B	5,421	747		332	
	980K	Block C	37,804	3,605		3026	
	328K	Block D	5,539	751		338	
	Total		37,804	4,946		**4,438**	

C. Point Addition over Elliptic Curves

Point addition is an important operation required for the task of encryption, decryption and authentication in Elliptic Curve Cryptography (ECC). Modern approaches represent the points in a projective coordinate systems, *e.g.*, the López-Dahab (LD) projective coordinate [25] due to which the point addition operation can be implemented as polynomials in the field.

Example IV.2. *Consider point addition in López-Dahab (LD) projective coordinate. 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. We represent point addition over the elliptic curve as* (X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, 1)*. Then* X_3, Y_3, Z_3 *can be computed as follows:*

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

TABLE IV: Point Addition Circuits (Time in seconds); k = Datapath Size, #Gates = No. of gates, Time-Out = 30 hrs, K = 10^3, M = 10^6

k	#Gates	F4 [4]	PB	ZR
64	15.3K	1.78	3.32	**0.72**
128	64K	40.55	27.41	**6.03**
163	104K	130.24	57.57	**13.13**
233	139K	335.60	106.85	**19.62**
283	281K	1,787.96	273.53	**64.48**
409	423K	5,077.50	578.15	**115.20**
571	1.14M	48,162.29	CR	**725.95**

The word-level abstraction approach in [4] presents the results for extracting the above representation for each of $A, B, \ldots, X_3, Y_3, Z_3$. It first performs a bit-level reduction and then a bit to word substitution for the primary input bit variables. Table IV shows the comparison of the bit-level reduction of ECC Point addition circuits as done in [4] against our implementation. This result demonstrates that our implementation can be integrated with that of [4] to improve the overall process.

D. Sequential Galois Field Multipliers

Example IV.3. *Consider a 3-bit RH-SMPO multiplier with output bits* $\{r_2, r_1, r_0\}$ *and input bits* $\{p_2, p_1, p_0, q_2, q_1, q_0\}$. *The multiplier performs the operation* $R = P \cdot Q$, *where* $R = r_0 \cdot \beta + r_1 \cdot \beta^2 + r_2 \cdot \beta^4$, $P = p_0 \cdot \beta + p_1 \cdot \beta^2 + p_2 \cdot \beta^4$ *and* $Q = q_0 \cdot \beta + q_1 \cdot \beta^2 + q_2 \cdot \beta^4$. *Also consider a Mastrovito multiplier performing the operation* $Z = A \cdot B$ *with* $Z = z_0 \cdot + z_1 \alpha + z_2 \cdot \alpha^2$, $A = a_0 \cdot + a_1 \alpha + a_2 \cdot \alpha^2$ *and* $B = b_0 + b_1 \cdot \alpha + b_2 \cdot \alpha^2$ *where* $\{z_2, z_1, z_0\}$ *and* $\{a_2, a_1, a_0, b_2, b_1, b_0\}$ *being the output and input bits respectively. The normal element* β *in terms of standard basis element is* $\beta = \alpha^3$. *The primitive polynomial used in the design of Mastrovito multiplier is* $P = X^3 + X + 1$ *and* α *is the root of this polynomial.*

The RH-SMPO multiplier output R can be written in the notations of standard basis as $R = r_0 \cdot \alpha^3 + r_1 \cdot \alpha^6 + r_2 \cdot \alpha^{12} = (r_0 + r_1 + r_2) + (r_0 + r_2) \cdot \alpha + (r_1 + r_2) \cdot \alpha^2$ *using* $\beta = \alpha^3$ *and* $\alpha^3 = \alpha + 1$. *Similarly,* $P = (p_0 + p_1 + p_2) + (p_0 + p_2) \cdot \alpha + (p_1 + p_2) \cdot \alpha^2$ *and* $Q = (q_0 + q_1 + q_2) + (q_0 + q_2) \cdot \alpha + (q_1 + q_2) \cdot \alpha^2$. *Therefore,*

$$
\begin{aligned}
&z_0 = r_0 + r_1 + r_2; \quad z_1 = r_0 + r_2; \quad z_2 = r_1 + r_2; \\
&a_0 = p_0 + p_1 + p_2; \quad a_1 = p_0 + p_2; \quad a_2 = p_1 + p_2; \\
&b_0 = q_0 + q_1 + q_2; \quad b_1 = q_0 + q_2; \quad b_2 = q_1 + q_2;
\end{aligned}
$$

Solving for p_i *and* q_i *in terms of* a_i *and* b_i *respectively,*

$$
\begin{aligned}
&p_0 = a_0 + a_2; \quad p_1 = a_0 + a_1; \quad p_2 = a_0 + a_1 + a_2; \\
&q_0 = b_0 + b_2; \quad q_1 = b_0 + b_1; \quad q_2 = b_0 + b_1 + b_2;
\end{aligned}
$$

Performing a bit-level reduction on r_0, r_1 *and* r_2 *results in the following remainders,*

$$
\begin{aligned}
r_0 &= p_1 \cdot q_0 + p_0 \cdot q_1 + p_2 \cdot q_1 + p_1 \cdot q_2 + p_2 \cdot q_2 \\
r_1 &= p_0 \cdot q_0 + p_2 \cdot q_0 + p_2 \cdot q_1 + p_0 \cdot q_2 + p_1 \cdot q_2 \\
r_2 &= p_1 \cdot q_0 + p_2 \cdot q_0 + p_0 \cdot q_1 + p_1 \cdot q_1 + p_0 \cdot q_2
\end{aligned}
$$

Using $z_0 = r_0 + r_1 + r_2$, *we get* $z_0 = p_0 \cdot q_0 + p_1 \cdot q_1 + p_2 \cdot q_2$. *Substituting* $\{p_i, q_i\}$ *in terms of* $\{a_i, b_i\}$ *results in* $z_0 = a_0 \cdot b_0 + a_1 \cdot b_2 + a_2 \cdot b_1$, *which is also the remainder if we reduce the* z_0 *bit of the Mastrovito multiplier modulo the polynomials of the gates of the circuit.*

TABLE V: RH-SMPO Multipliers; k = Datapath Size, #Gates = No. of gates, Time-Out = 30 hrs, K = 10^3

k	33	51	65	81	89	99
#Gates	3.5K	8.5K	13.6K	21.4K	25.9K	32K
[26]	112.6	1,129	5,243	20,724	36,096	67,021
PB	0.59	2.02	3.10	5.99	7.72	13.26
ZR	**0.10**	**0.26**	**0.48**	**0.84**	**1.08**	**1.48**

TABLE VI: AG-SMPO Multipliers; k = Datapath Size, #Gates = No. of gates, Time-Out = 30 hrs, K = 10^3

k	36	66	82	89	100
#Gates	3.8K	13K	20K	23.6K	29.8K
[26]	113	3,673	15,117	28,986	50,692
PB	0.48	2.84	6.93	7.29	9.91
ZR	**0.10**	**0.46**	**0.76**	**0.95**	**1.30**

E. Integer Multiplication

We applied our technique to integer arithmetic circuits, which showed an exponential increase in time. Performing detailed analysis of a 7x7 multiplier reveals that, when reducing the z_{13} bit (MSB) and z_{12} bit of this circuit, the maximum number of monomials encountered are 429,889 and 897,955 respectively. However, the modulo-2 sum (XOR) of these ZBDDs contains only 789,604 monomials (during the modulo-2 sum common monomials cancel out) as opposed to 1,327,844 (= 429,889 + 897,955). This modulo-2 sum indicates that reducing all the outputs simultaneously results in monomials that can cancel each other. Therefore, integer multipliers require a word-level decision procedure, as given in [10] [11], that account for the cancellation of these monomials across multiple bits in one word-level expression. The experiment suggests that for integer arithmetic multipliers, integrating the implicit data structure with a word-level representation of the output bit vector can yield significantly better results.

V. Conclusion

This paper has presented an approach to derive a canonical polynomial representation for each output bit z_i of a circuit, by modeling the gates of the circuit as a set of polynomials G over $\mathbb{F}_2$, and performing the reduction $z_i \xrightarrow{G}_+ r_i$. The unate cube set algebra prowess of ZBDDs is exploited to represent the polynomials implicitly. We take further advantage of this data structure to improve the classical Gröbner basis reduction method that relies on canceling only 1 monomial in every iteration of division. Our approach cancels multiple monomials in each step of division, thus speeding up the reduction. The efficiency of our approach is demonstrated by completing the reduction for up to 571-bit modulo multipliers in the allotted time, and significant improvement is achieved over the F4-style reduction, parallelized reductions and PolyBori based techniques. As part of our future work, we will be pursuing word-level implementation of the polynomial reduction for integer arithmetic circuits.

Acknowledgement: The authors wish to thank Cunxi Yu for assistance with optimization of the benchmarks used in the experiments.

References

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

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

[3] 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*, 2008, pp. 473–486.

[4] T. Pruss, P. Kalla, and F. Enescu, “Efficient Symbolic Computation for Word-Level Abstraction from Combinational Circuits for Verification over Finite Fields,” *IEEE Trans. on CAD*, vol. 35, no. 7, pp. 1206–1218, July 2016.

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

[6] B. Buchberger, “Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal,” Ph.D. dissertation, Philosophiesche Fakultat an der Leopold-Franzens-Universitat, Austria, 1965.

[7] *Faugère*, “A new efficient algorithm for computing Gröbner bases (F_4),” *Journal of Pure and Applied Algebra*, vol. 139, pp. 61–88, June 1999.

[8] J. Lv, P. Kalla, and F. Enescu, “Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits,” in *IEEE Trans. on CAD*, vol. 32, no. 9, 2013, pp. 1409–1420.

[9] X. Sun, P. Kalla, and F. Enescu, “Word-level Traversal of Finite State Machines using Algebraic Geometry,” in *Proc. High-Level Design Validation and Test*, 2016.

[10] M. Ciesielski, C. Yu, D. Liu, W. Brown, and A. Rossi, “Verification of Gate-Level Arithmetic Circuits by Function Extraction,” in *Proc. Des. Auto. Conf. (DAC)*, 2015.

[11] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, “Formal verification of integer multipliers by combining gröbner basis with logic reduction,” in *Proc. Design Automation and Test in Europe*, 2016, pp. 1048–1053.

[12] C. Yu and M. Ciesielski, “Efficient parallel verification of galois field multipliers,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 238–243.

[13] S. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in *Design Automation Conference (DAC)*, 1993, pp. 272–277.

[14] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, “Singular 3-1-6 — A computer algebra system for polynomial computations,” http://www.singular.uni-kl.de, 2012.

[15] O. M. Hansen and J.-F. Michon, “Boolean Gröbner basis,” in *Proc. Boolean Functions Cryptography & Applications*, 2006, pp. 185–201.

[16] M. Y. Vardi and Q. Tran, “Groebner Bases Computation in Boolean Rings for Symbolic Model Checking,” in *IASTED*, 2007.

[17] M. Brickenstein and A. Dreyer, “Polybori: A Framework for Gröbner Basis Computations with Boolean Polynomials,” *Journal of Symbolic Computation*, vol. 44, no. 9, pp. 1326–1345, September 2009.

[18] B. Buchberger, “A criterion for detecting unnecessary reductions in the construction of a groebner bases,” in *EUROSAM*, 1979.

[19] S. Minato, “Calculation of Unate Cube Set Algebra using Zero-Suppressed BDDs,” in *Proc. Design Automation Conference (DAC)*, 1994, pp. 420–424.

[20] F. Somenzi, “CUDD: CU Decision Diagram Package Release 3.0.0,” 2015.

[21] Berkeley Logic Synthesis and Verification Group, “ABC: A system for sequential synthesis and verification,” www.eecs.berkeley.edu/alanmi/abc, 2007.

[22] C. Koc and T. Acar, “Montgomery Multiplication in GF(2^k),” *Designs, Codes and Cryptography*, vol. 14, no. 1, pp. 57–69, Apr. 1998.

[23] H. Wu, “Montgomery Multiplier and Squarer for a Class of Finite Fields,” *IEEE Transactions On Computers*, vol. 51, no. 5, May 2002.

[24] 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*, 2008, pp. 77–87.

[25] J. López and R. Dahab, “Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n),” in *Proceedings of the Selected Areas in Cryptography*. London, UK, UK: Springer-Verlag, 1999, pp. 201–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=646554.694442

[26] X. Sun, P. Kalla, T. Pruss, and F. Enescu, “Formal verification of sequential galois field arithmetic circuits using algebraic geometry,” in *Proc. Design, Automation and Test in Europe*, 2015.