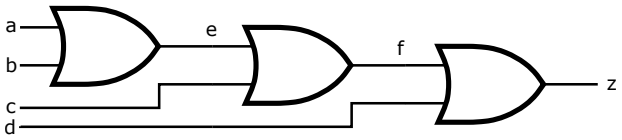

Figure 1: A chain of OR gates

1 Preliminaries

Let $\mathbb{F}_2$ be the field of two elements, $\{0, 1\}$. A polynomial ring, $\mathbb{F}_2[x_1, \ldots, x_d]$, is the set of all polynomials in variables, $x_1, x_2, \ldots, x_d$, and coefficients in the field $\mathbb{F}_2$. A monomial, $M = {x_1}^{\alpha_1} \cdots {x_d}^{\alpha_d}$, is a power product in the variables $x_1, \ldots, x_d$ with $\alpha_i \geq 0$. A polynomial, $P \in \mathbb{F}[x_1, x_2, \ldots, x_d]$, can be written as $P = c_1X_1 + c_2X_2 + \cdots + c_tX_t$, where $c_1, \ldots, c_t$ are the coefficients and $X_1, \ldots, X_t$ are monomials. In this paper, the terms will be ordered lexicographically (LEX).

Definition 1.1. Boolean Polynomials [1]: Let $P \in \mathbb{F}_2[x_1, x_2, \ldots, x_d]/\langle {x_i}^2 - x_i \rangle$ be a Boolean polynomial, such that

$$P = c_1 \cdot {x_1}^{\alpha_{11}} \cdots {x_d}^{\alpha_{1d}} + \cdots + c_t \cdot {x_t}^{\alpha_{t1}} \cdots {x_d}^{\alpha_{td}}$$

where, $c_i \in \mathbb{F}_2$ and $\alpha_{ij} \in \{0, 1\}$.

The circuit under consideration is modeled as a set of Boolean polynomials using the following one-to-one mapping:

$$\begin{gathered} \neg a \rightarrow a + 1 \pmod 2 \\ a \wedge b \rightarrow a \cdot b \pmod 2 \\ a \vee b \rightarrow a \cdot b + a + b \pmod 2 \\ a \oplus b \rightarrow a + b \pmod 2 \end{gathered}$$

where, $a, b \in \{0, 1\}$.

Ideals and Varieties [3]: An *ideal* J generated by polynomials, $P_1, \ldots P_s \in \mathbb{F}_2[x_1, \ldots, x_d]$ is:

$$J = \langle P_1 \ldots P_s \rangle = \{\textstyle\sum_{i=1}^{s} h_i \cdot P_i : h_i \in \mathbb{F}_2[x_1, \ldots, x_d]\}$$

The polynomials, $P_1, \ldots, P_s$, form the basis or generators of the ideal J.

Let $\mathbf{a} = (a_1, \ldots, a_d) \in \mathbb{F}_2^d$ be a point, and $P \in \mathbb{F}_2[x_1, \ldots, x_d]$ be a polynomial. We say that P vanishes on $\mathbf{a}$ if $P(\mathbf{a}) = 0$.

For an ideal $J = \langle P_1 \ldots P_s \rangle$, the *variety* of J over $\mathbb{F}_2$ is:

$$V(J) = \{\mathbf{a} \in \mathbb{F}_2^d : \forall P \in J, P(\mathbf{a}) = 0\}$$

Gröbner basis [3]: There can be many different generators of an ideal J i.e. $J = \langle P_1 \ldots P_s \rangle = \langle Q_1 \ldots Q_s \rangle$, where $\{P_1 \ldots P_s\}$ and $\{Q_1 \ldots Q_s\}$ are two different sets of polynomials. Gröbner basis is a particular kind of generating set of an ideal which allows to solve many polynomial decision problems.

The famous Buchberger's algorithm [2] can compute the Gröbner basis of an ideal but at a very expensive computational complexity. A term order, called

Reverse Topological Term Order (RTTO), derived from the topology of the circuit has been presented in [3]. The polynomials for each gate with this order have the representation of the type, $P_i = x_i + tail(P_i)$. For example, the circuit in Figure 1 can have a variable ordering as, $z > f > e > d > c > b > a$. The polynomials for the three OR gates will be $P_1 = z+f \cdot d+f+d, P_2 = f+e \cdot c+e+c, P_3 = e+b \cdot a+b+a$. As the output of each gate is unique, the leading terms of all the polynomials corresponding to the circuit become relatively prime. Therefore, the RTTO obviates the need for computing a Grobner basis for this set of polynomials as this set is itself a Grobner basis. This term order is used in this paper hereafter.

Polynomial Reduction: Reduction of a polynomial F *w.r.t.* another polynomial G is defined as follows:

$$F \xrightarrow{G} F - \frac{lt(F)}{lt(G)} \cdot G = F + \frac{lt(F)}{lt(G)} \cdot G \tag{1}$$

where $lt(F)$ and $lt(G)$ denote the leading terms of polynomials, F and G, respectively. Note that $-$ can be replaced with $+$ as $-1 \pmod 2 = 1 \pmod 2$. This equation holds only if $lt(G)$ divides $lt(F)$. As we are working with circuits and RTTO, the expression for G will always be like, $G = x_i + tail(G)$. Therefore, F will be completely reduced by G when there is no term in F that contains x_i in it and is represented as $F \xrightarrow{G}_+$.

2 Theory

Our verification procedure is based on reducing the output variables of the circuit *w.r.t* the polynomials of the circuit which result in an expression composed of only input variables. This expression is canonical for a particular output. Next, we explain the reduction with an example and discuss how ZBDDs can improve the process.

Consider, as an example, that we want to reduce the output, $F = z$, of the circuit in Figure 1 using Eqn. 1 (by P_1, P_2, P_3). The steps are:

1. $z \xrightarrow{P_1} fd + f + d$
2. $fd + f + d \xrightarrow{P_2} f + edc + ed + dc + d \xrightarrow{P_2} edc + ed + ec + e + dc + d + c$
3. $edc + ed + ec + e + dc + d + c \xrightarrow{P_3} ed + ec + e + dcba + dcb + dca + dc + d + c \xrightarrow{P_3} ec + e + dcba + dcb + dca + dc + dba + db + da + d + c \xrightarrow{P_3} e + dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c \xrightarrow{P_3} dcba + dcb + dca + dc + dba + db + da + d + cba + cb + ca + c + ba + b + a$

In the first step, F is reduced by P_1 just once as there is only one term in F that has z in it. In the second step, the result of step one is reduced twice by P_2 as the result has two terms that have f in them. Similarly, four reductions by

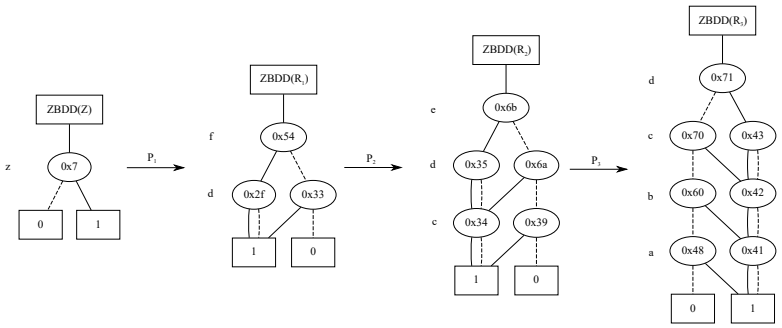

Figure 2: Reduction of output of the circuit in Figure 1 by P_1, P_2, P_3

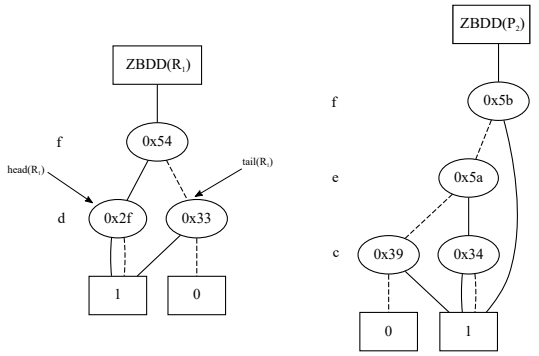

Figure 3: ZBDD for polynomial R_1 and P_2

P_3 are required to reduce the result of step two into an expression containing only primary inputs (which cannot be further reduced). Therefore, each step requires a number of iterations for complete reduction by a single polynomial and the total number of reductions is 7 (one by P_1, two by P_2, four by P_3). Morever, the size of the expression after each step is increasing exponentially in the number of monomials.

Figure 2 shows the same steps of complete reduction by P_1, P_2, P_3 using ZBDDs (exact procedure of reduction using ZBDDs will be discussed later). The size of the ZBDDs after complete reductions by P_1, P_2, P_3 increases linearly in the number of nodes.

Next we will show how reduction can be performed using ZBDDs. In step 2, the polynomial, $P = fd + f + d$, needs to be reduced by P_2. The ZBDDs for P and P_2 are shown in Figure 2 and 3 respectively. The ZBDD for P has three paths (fd, f, d) terminating in the node 1 that correspond to its monomials. The ZBDD manager creates the ZBDDs with the defined monomial order, and therefore, the topmost node in both ZBDDs is f.

The algorithm for conventional reduction procedure using ZBDDs is shown in Algroithm 1. The procedure takes F and $POLY\_LIST$ as input arguments. F

is the polynomial that we need to reduce *w.r.t.* the polynomials (corresponding to the gates of the circuit), $G \in POLY\_LIST$. The variables in the ZBDD manager are declared in the same order as the RTTO. For our example of Figure 1, the first variable declared in the ZBDD manager is z, then f, and so on. If a circuit has polynomials, $P_1 = x_1 + tail(P_1) \cdots P_d = x_d + tail(P_d)$ with variable order $x_1 > \cdots > x_d$, then the first element of the list, $POLY\_LIST$ is P_1, second P_2, and the last P_d. The top-most node in a ZBDD corresponds to the variable that has highest precedence in the RTTO. Populating $POLY\_LIST$ in this way avoids the search required to find a polynomial $G \in POLY\_LIST$ that can divide the leading term of F. While iterating over the polynomials $G \in POLY\_LIST$ if a certain polynomial does not divide the leading term of F, it will imply that polynomial is not in the logical cone of F.

The procedure, $leading\_term(G)$, takes in a ZBDD as an input argument and returns the leading term of the polynomial that the ZBDD represents. The procedure starts at the root node and follows the THEN path until it reaches the terminal node 1. The cube of the variables, whose corresponding nodes are encountered on this path, gives us the leading term of the polynomial. The procedure, $Cudd\_zddDivide(A, B)$, takes two ZBDDs as input argument where B can only be a cube. If B divides A, it returns the quotient of the division, else it returns zero.

The polynomial, F, is completely reduced *w.r.t.* the polynomial G in the while loop. The $\cdot$ operator computes the product of two ZDDs. Also, the $+$ operator is a modulo 2 sum which can be implemented as,

$$A + B = A \cup B - A \cap B, \text{ where } A \text{ and } B \text{ are two ZBDDs}$$

$\cup$ and $\cap$ are set union and set intersection operations respectively. This computation results in large intermediate ZBDDs for the union of A and B. Therefore, this operation and the product is implemented using the Recursive Boolean Addition and Product algorithm as presented in [1].

As the polynomial, F, changes for every iteration of the while loop, the ZBDDs in *lead_F* and *divide* are updated everytime at the end of the loop. When F no longer contains the leading term of a polynomial G in any of its terms, the procedure $Cudd\_zddDivide$ returns zero and the while loop terminates. The total number of iterations of the nested loops for the example of Figure 1 is 7.

Next, we will show how $P = fd + f + d$ can be reduced by P_2 in one step i.e eliminating the need of the while loop in Algorithm 1. If we check the THEN branch of node f in P (Figure 2), we will find that it represents the polynomial, $d + 1$. Therefore, the THEN branch of the top-most node of F gives us all the terms that appear with f. So the reduction can be performed by multiplying $d + 1$ with P_2 and adding this product to $F \pmod 2$,

$$\begin{aligned}
&(fd + f + c) + (d + 1) \cdot (f + ec + e + c) \pmod 2 \\
&= 2 \cdot (fd + f) + c + (d + 1) \cdot (ec + e + c) \pmod 2 \\
&= c + (d + 1) \cdot (ec + e + c) \pmod 2
\end{aligned}$$

Algorithm 1 Single Monomial Reduction

```
1: procedure single_mon_red(F, POLY_LIST)
2:     for each G ∈ POLY_LIST do
3:         lead_G = leading_term(G)
4:         lead_F = leading_term(F)
5:         divide = Cudd_zddDivide(lead_F, lead_G)
6:         while divide ≠ zero do
7:             prod = divide · G
8:             F = F + prod
9:             lead_F = leading_term(F)
10:            divide = Cudd_zddDivide(lead_F, lead_G)
11:        end while
12:    end for
13:    return F
14: end procedure
```

Algorithm 2 Multi-Term Reduction Algo 1

```
1: procedure multi_mon_red_1(F, POLY_LIST)
2:     for i from 0, 1, ..., size(POLY_LIST) − 1 do
3:         G_i = POLY_LIST[i]
4:         index_{G_i} = G_i → index
5:         index_F = F → index
6:         if index_{G_i} == index_f then
7:             F = ELSE(F) + THEN(F) · ELSE(G_i)
8:         end if
9:     end for
10:    return F
11: end procedure
```

Consider the follwing terminologies,

$$head(F) = \text{THEN branch of top-most node of } F = d + 1$$
$$tail(F) = \text{ELSE branch of top-most node of } F = c$$
$$head(G) = \text{THEN branch of top-most node of } G = f$$
$$tail(G) = \text{ELSE branch of top-most node of } G = ec + e + c$$

In the above example, 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, $F \xrightarrow{G}_{+}$,

$$= c + (d + 1) \cdot (b + a)$$
$$tail(F) + head(f) \cdot tail(G)$$

The data structure for a ZBDD node has two pointers for the THEN child and ELSE child, respectively. Therefore, $head(F)$, $tail(F)$, and $tail(G)$ can be acquired by just accessing the respective pointers. So the reduction process effectively involves two operations, a modulo 2 sum and a product.

The algorithm for Multiple Monomial Reduction is shown in Algorithm 2, where the notations, F and $POLY\_LIST$, are same as that for the Single Monomial Reduction algorithm. Unlike in Algorithm 1, where we need to find the quotient of $lead\_F/lead\_G$, in Algorithm 2 we only need to determine if $lead\_G$ can divide F at all (in this case the quotient is $THEN(F)$). This can be accomplished by just comparing the indices of top-most nodes of F and G. This algorithm significantly cut down the number of iterations which are now equal to the size of $POLY\_LIST$ for each output. For the example Figure 1, the number of iterations is 3 using Algorithm 2.

The AND-XOR heuristic presented in [4] can also be implemented on the ZBDD framework. VAR_LIST is the list of the variables in the polynomial model. We create another list, $XVAR\_LIST$, that contain all the variables except the inputs and outputs variables of XOR gates and primary inputs and outputs in the circuits. While iterating over all the polynomials, G_i, in the $POLY\_LIST$, we check if the ELSE part, e, of G_i contain any variable from the $XVAR\_LIST$. This can be done using the sub-routines for finding the support of a ZBDD and checking if any of these variables are contained in the $XVAR\_LIST$. If it does, we reduce e with the polynomial that has that variable as the leading term using a variant of Multi-Term Reduction algorithm, presented in Algorithm 4. This algorithm can reduce the ZBDD of a polynomial either by another polynomial or a cube depending on the value of the parameter, par. The polynomial e is then reduced by the cube of pair of AND-XOR variables which have been already extrated from the circuit topology and stored in the list, AX_LIST. Reducing a polynomial by a cube of type $x_i \cdot x_j$ is equivalent to replacing all the monomials containing this cube with zero.

A similar algorithm for applying the fan-out heuristic is also implemented. The algorithm will take a list, instead of $XVAR\_LIST$, that contains all the variables except the the ones that occur in the *tail* of more than one polynomial and primary inputs and outputs.

Algorithm 3 AND-XOR Elimination

```
1: procedure and_xor(POLY_LIST,XVAR_LIST)
2:     for G_i ∈ POLY_LIST do
3:         e = ELSE(G_i)
4:         s = SUPPORT(e)
5:         for each var ∈ s do
6:             if var ∈ XVAR_LIST then
7:                 e = multi_mon_red_2(e, POLY_LIST[var → index], 0)
8:                 for each p ∈ AX_LIST do
9:                     e = multi_mon_red_2(e, p, 1)
10:                end for
11:            end if
12:        end for
13:        G_i = ITE(VARS(i), THEN(G_i), e)
14:    end for
15: end procedure
```

Algorithm 4 Multi-Term Reduction Algo 2

```
1: procedure multi_mon_red_2(G_i, R, par)
2:     if par == 0 then
3:         divide = Cudd_zddDivide(G_i, VAR_LIST[R → index])
4:     else
5:         divide = Cudd_zddDivide(G_i, R)
6:     end if
7:     if divide ≠ 0 then
8:         return F + G_i · divide
9:     else
10:        return F
11:    end if
12: end procedure
```

References

[1] Michael Brickenstein and Alexander Dreyer. Polybori: A framework for grbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[2] B. Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *SIGSAM Bull.*, 10(3):19–29, August 1976.

[3] J. Lv, P. Kalla, and F. Enescu. Efficient gröbner basis reductions for formal verification of galois field multipliers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 899–904, March 2012.

[4] A. Sayed-Ahmed, D. Groe, U. Khne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1048–1053, March 2016.