A tree search algorithm for low multiplicative complexity logic design

Citation for published version:
https://doi.org/10.1016/j.future.2018.01.063

Digital Object Identifier (DOI):
10.1016/j.future.2018.01.063

Link:
Link to publication record in Heriot-Watt Research Portal

Document Version:
Peer reviewed version

Published In:
Future Generation Computer Systems

Publisher Rights Statement:
© 2018 Elsevier B.V.

General rights
Copyright for the publications made accessible via Heriot-Watt Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
Heriot-Watt University has made every reasonable effort to ensure that the content in Heriot-Watt Research Portal complies with UK legislation. If you believe that the public display of this file breaches copyright please contact open.access@hw.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
A Tree Search Algorithm for Low Multiplicative Complexity Logic Design

Jia Jun Tay, M. L. Dennis Wong, Ming Ming Wong, Cishen Zhang, Ismat Hijazin

Abstract

Low multiplicative complexity logic design is a useful heuristic to achieve low gate count implementation of logic circuit. In this work, we propose a deterministic approach based on the currently known lower and upper bounds of multiplicative complexity for logic minimization problems with not more than five inputs. The proposed tree search algorithm achieves circuit minimization through decomposition of Positive Polarity Reed-Muller expressions. This approach allows low multiplicative complexity logic design to be executed without the consistency issue associated with the randomized approach in the original algorithm. Experimental results show over 85% improvement in computation time compared to solving the same problems using the previous randomized approach. We also demonstrate that the quality of results produce by the proposed algorithm is comparable, and in some cases, better than the results reported in previous works using the same heuristic.

Keywords: Logic design, Multiplicative complexity, Function decomposition, Tree search algorithm

Email address: jtay@swinburne.edu.my (Jia Jun Tay)

1 This work has been supported in part of the Melbourne-Sarawak Research Collaboration Scheme.

2 Early concept of this work appeared in [1].
1. Introduction

Logic design is an exercise of interconnecting basic logic building blocks to perform a specific function. Each logic gate required in a circuit constitutes cost in terms of physical space and money. Thus, it is desirable to reduce area or complexity in logic circuits through the process of logic minimization. However, as proven in [2], logic minimization is a $\sum^P_2$-complete problem. Therefore, circuit designers often rely on a variety of heuristics to achieve economical circuits for specific applications. These heuristics form the backbone for various logic minimization algorithms, trading absolute optimality of the results for practical computation time. Regardless, a good logic minimization algorithm should enable solutions not far from the quality of the optimal case despite the trade-off.

Logic minimization techniques operate over a functionally complete logic basis, i.e., sufficient to describe any arbitrary function. Karnaugh mapping [3] and the Quine-McCluskey algorithm [4] are well known examples of logic minimization techniques that operate over the logic basis (AND, OR, NOT). Variants of the Espresso logic minimizer [5] were introduced to facilitate the use of computers in logic minimization as circuits have grown larger and more complex for the previous algorithms to efficiently compute. The interest in the alternative logic basis (AND, XOR, NOT) started after [6] conjectured that exclusive-or sum-of-products (ESOP) enabled circuit designs that required at most the same number of products over the conventional sum-of-products (SOP). This conjecture is particularly interesting as it directly implies that circuits constructed over the logic basis (AND, XOR, NOT) require less gate count compared to their counterpart designed over logic basis (AND, OR, NOT). However, logic minimization over the logic basis (AND, XOR, NOT) remains a difficult problem even until this day. For a single-output problem with $n$ input variables, there exists $2^{2^n}$ possible functions in ESOP form and there are no known algorithms that can reliably find an optimal circuit construction for the problem.

We observed various logic minimization heuristics developed over the last two decades in the attempt to achieve near-optimal circuit in this logic basis for different applications. Among the proposed algorithms, we find the concept of low multiplicative complexity logic minimization particularly interesting. Multiplicative complexity of logic circuits has been studied exten-
sively in [7]. The same authors then iterated on the concept and proposed a two-step logic minimization heuristic in [8], which we will hereby refer to as the Boyar-Peralta heuristic. The Boyar-Peralta heuristic demonstrated good results on the AES $GF(2^4)$ S-Box in [8] and an efficient construction of the PRESENT S-Box was also attempted in [9] following the same concept. The algorithm relies on an initial AND-minimization step to achieve optimal multiplicative complexity, then solves XOR-minimization as a Shortest Linear Program (SLP) problem to minimize the number of additions.

1.1. Contribution

In this work, we propose a novel approach to achieve low multiplicative complexity circuits based on the decomposition of Positive Polarity Reed-Muller (PPRM) expressions. The proposed approach is a tree search algorithm that leverages the currently known lower bound [23] and upper bound [24] of multiplicative complexity to solve problems with $n \leq 5$ inputs (a limitation associated with the current known upper bound). The contribution of the proposed algorithm is threefold: (a) It allows a deterministic approach to search for low multiplicative complexity implementations without the undesirable characteristics present in the original randomized approach, (b) when experimented on the same set of problems, the proposed algorithm showed over 85% improvement in computation time, and (c) solutions discovered by the proposed algorithm are comparable if not better than previous results.

While the proposed algorithm can be applied directly to solve combinational logic minimization problems with small number of inputs, it works reasonably well on large circuits that make repeated use of smaller components. A good example of this is the substitution-boxes (S-Boxes) of block ciphers. Although most block ciphers operate on large input sizes (ranging from 64 to 128 bits), non-linear transformations of data blocks are typically realized by many instances of S-Boxes with much smaller input sizes. As a case in point, many lightweight block ciphers proposed over the last decade utilize small 4-bit S-Boxes to achieve circuit compactness (see mCrypton [11], PRESENT [26], Piccolo [12], LED [13], PRINCE [14], and Midori [15]). In addition, block ciphers with larger 8-bit S-Boxes such as AES [16] and potentially CLEFIA [17] can also benefit from the proposed algorithm through composite field arithmetic. As demonstrated in [27], by performing the 8-bit Galois field inversion using subfields of four bits, the proposed algorithm can be applied to minimize the resultant 4-bit inversion circuit.
As recently reported in [10], the U.S. National Institute of Standards and Technology (NIST) is actively developing a strategy for the standardization of lightweight cryptographic algorithms. State-of-the-art approaches to hardware optimization of lightweight block ciphers have been relatively confined to architecture-based techniques as reported in [18, 19]. The proposed algorithm contributes to this effort by solving for area-efficient implementations of the non-linear transformations involved in these ciphers.

1.2. Structure of the Paper

The rest of the paper is organized as follows. Section 2 provides the definitions and notations for important terminologies used in this paper and the necessary preliminary knowledge. In Section 3, we describe the problems associated with the current approach in low multiplicative logic design. We discuss the relevance of decomposing Positive Polarity Reed-Muller (PPRM) expressions in low multiplicative complexity logic design in Section 4 and propose a tree search algorithm based on the concept in Section 5. In Section 6, we discuss product sharing between functions in multiple-output problem and expanded the tree search algorithm to include this feature in Section 7. We analyze the time complexity of the proposed algorithm and compared the performance in terms of computation time with the previous approach in Section 8. The quality of the results produced by the proposed algorithm is then evaluated and compared in Section 9. Concluding remarks are then drawn in Section 10 with a brief discussion on potential future work and challenges.

2. Background

In this section, we discuss concepts, definitions and notations that are useful in the context of this work. We assume the readers are familiar with the standard definitions and notations associated with Boolean Algebra. Most of the nomenclatures used in this writing follow their respective definitions in [20].

2.1. Finite Field Arithmetic

Logic operations over the basis (AND, XOR, NOT) are closely related to modulo-2 arithmetic. For instance, logic XOR is equivalent to mod-2 addition and can be expressed in both logical and mathematical notations as
shown in (1). In this case, \( c \) represents the sum resulting from the addition of \( a \) and \( b \).

\[
 a \oplus b = \text{logical} \\
 a + b = \text{mathematical} = c 
\]  

(1)

Similarly, logic AND is equivalent to mod-2 multiplication. The logical and mathematical notations are shown in (2), where \( c \) is the product resulting from the multiplication of \( a \) and \( b \).

\[
 a \odot b = \text{logical} \\
 ab = \text{mathematical} = c 
\]  

(2)

Finally, the logic NOT is mathematically equivalent to mod-2 addition with the value '1'.

\[
 \pi = \text{logical} \\
 a + 1 = \text{mathematical} = c 
\]  

(3)

These comparisons are not trivial, as the proposed approach to low multiplicative complexity logic design revolves heavily upon the mathematical properties of mod-2 arithmetic. Therefore, mathematical notation is used liberally when describing an expression in this work (logical notations are used only when implying actual logic gates). As a result, we emphasize that the use of the '+' sign in this work indicates logical XOR, not to be misinterpreted as logical OR.

2.2. Positive Polarity Reed-Muller Expressions

We first establish basic definitions for key terms required to enable the interpretation of Positive Polarity Reed-Muller (PPRM) expressions.

**Definition 1.** A variable is a symbol representing a single input for a function. For example, \( x_1, x_2, \ldots, x_n \) represent the set of variables for an \( n \)-input function.

**Definition 2.** A literal refers to a variable or its negation. For example, \( x_1 \) and \( \overline{x_1} \) are literals.

**Definition 3.** A cube is the conjunction of one or more literal(s). For example, given a function \( f = x_1x_2x_3 + x_1x_2 + x_1 \), its cubes are \( x_1x_2x_3, x_1x_2 \) and \( x_1 \).
Definition 4. An expression is the disjunction of one or more cube(s). For example, \( f = x_1x_2x_3 + x_1x_2 + x_1 \) is an expression.

Definition 5. A truth vector is the vector representation for a truth table of a function such that \( F = [f(0), f(1), ..., f(2^n - 1)]^T \). For example, the truth vector for the function \( f = x_1x_2 \) is written as \( F = [0, 0, 1]^T \).

PPRM expressions are a form of logic expression to describe the circuit output as a function of the circuit inputs over the logic basis (AND, XOR, NOT) [21]. It is one of the two forms of logic expressions under the category of Fixed Polarity Reed-Muller (FPRM) expression, the other being Negative Polarity Reed-Muller (NPRM) expression. The term "Positive Polarity" implies that all variables in the Reed-Muller expression appear uncomplemented. In this work, we limit our discussion to the use of PPRM. However, the concept of the proposed algorithm can be applied to NPRM expressions without modification.

Definition 6. Given a function with \( n \)-bit inputs \( x_1, x_2, ..., x_n \), its PPRM expressions can be defined as shown in (4) where \( a_0, a_1, ..., a_{2^n-1} \in \{0, 1\} \) and \( \pi_0, \pi_1, \pi_2, \pi_3, ..., \pi_{2^n-1} = 1, x_1, x_2, x_2x_1, x_3, ..., x_nx_{n-1}...x_1 \) fully describe \( f \).

\[
f(x_1, x_2, ..., x_n) = a_0\pi_0 + a_1\pi_1 + ... + a_{2^n-1}\pi_{2^n-1} \tag{4}
\]

For any arbitrary function, there is only one unique representation in PPRM form. This unique PPRM expression can be derived easily by multiplying the truth vector of the function with the \( n \)-variable transform matrix \( T_n \) given in (5) where \( T_0 = [1] \).

\[
T_n = \begin{bmatrix}
T_{n-1} & 0 \\
T_{n-1} & T_{n-1}
\end{bmatrix}
\tag{5}
\]

2.3. Low Multiplicative Complexity Logic Design

Low multiplicative complexity logic design is a heuristic based on the conjecture that given a set of functions \( f_0, f_1, ..., f_i \in \langle x_1, x_2, ..., x_n \rangle \) over the logic basis (AND, XOR, NOT), the circuit representation that requires the minimal number of multiplications usually results in a solution with close to optimal gate count. This approach is not to be confused with works that minimize product terms or minterms in ESOP expressions. The full description of the original algorithm can be referenced in [22].
Definition 7. The multiplicative complexity $c_{\land}(f)$ of a function $f$ is the minimal number of multiplication (AND gates) required to realize the function over the logic basis (AND, XOR, NOT). The same notation can be used on a set of functions of the same $n$-input variables to describe the multiplicative complexity of a multiple-output circuit as a whole, i.e. $c_{\land}(f_0, f_1, ..., f_i)$ where $f_0, f_1, ..., f_i \in \langle x_1, x_2, ..., x_n \rangle$.

Definition 8. The degree of an expression is an integer $d$ which implies the number of literals present in the cube of an expression that has the highest number of literals. For example, the function $f = x_1 x_2 x_3 + x_1 x_2 + x_1$ has a degree of $d = 3$.

Works in [23] and [24] have allowed us to draw Lemmas 1 and 2 respectively. Lemma 1 gives the lower bound for the multiplicative complexity of a function, whereas Lemma 2 gives the upper bound for functions with up to five variables.

Lemma 1. Given a function $f$ of degree $d$, the multiplicative complexity is at least $d - 1$, i.e. $c_{\land}(f) \geq d - 1$.

Lemma 2. Given an $n$-variable function $f$, its multiplicative complexity is at most $n - 1$ as long as $n \leq 5$, i.e. $c_{\land}(f) \leq n - 1$ given $n \leq 5$.

As mentioned previously, the low multiplicative complexity logic minimization heuristic is essentially a two step process. Assuming a problem is described in the form of truth vectors, the proposed logic minimization process is illustrated in Figure 1.

![Figure 1: Processes in low multiplicative complexity logic design.](image)

2.3.1. AND-Minimization

This step is also known as the non-linear step. The objective of this step is to construct an XOR-AND circuit using the least amount of AND gates possible. In this stage, the cost of XOR gates is ignored. In the original approach, the minimization of AND gates is achieved through a randomized
selection algorithm. Given an $n$-variable problem, a sample space is formed initially with the $n$ number of variables. The algorithm alternates between addition and multiplication using two randomly selected elements from the sample space. Each sum or product resulting from the operation is introduced into the sample space if the amount of multiplication required does not exceed the known multiplicative complexity. This process is repeated until a sum or product produces the desired function. In this work, an alternative approach for AND-minimization is proposed (see Section 4).

2.3.2. XOR-Minimization

This step is also known as the linear step. This step takes the multiplication-optimized result from the non-linear step and maximizes the XOR-sharing between the purely linear portion of the circuit. As demonstrated in [8], the process is similar to solving a SLP problem. To achieve this, purely linear expressions are formed from the results of the non-linear step as demonstrated in Example 1. A distance vector indicating the number of additions required by each expression is formed. In each iteration, the algorithm searches for a sum which reduces the magnitude of the distance vector by the maximum amount. The process is repeated until the distance vector is reduced to zero. Since each iteration reduces the distance vector by the maximum amount, the linear step gives the shortest path (minimum number of XOR gates) to construct the linear portion of the circuit. In [25], an additional tie-breaker criterion has been proposed to improve the resultant circuit depth.

While it may seem counterintuitive to minimize the number of multiplication instead of addition (XOR gates are more complex than AND gates), we argue that minimizing the number of multiplication is indirectly beneficial to reducing the number of addition. To begin with, reduction of XOR gates is performed in the linear step of the heuristic. As described in [8], this process is equivalent to solving a NP-hard SLP problem. From Example 1, we observe that each product term translates into a new variable when solving the SLP problem (see $w_5$ and $w_6$). Thus, minimizing the number of AND gates allows SLP to solve for minimal number of variables. Of course, this does not guarantee the resultant circuit to be the absolute optimal solution in terms of gate count. However, this concept is sufficient as a heuristic to obtain near-optimal results in practical time.

Example 1. Given a function $f = x_1 + x_4 + x_1(x_2 + x_3(x_1 + x_4))$ that is optimal in terms of multiplicative complexity with $c^\land(f) = 2$, the function
will be transformed into three purely linear expressions as demonstrated in (6).

\[ f = x_1 + x_4 + x_1(x_2 + x_3(x_1 + x_4)) \]  

\[ z_1 = w_1 + w_4 \]  
\[ z_2 = w_2 + w_5 \]  
\[ z_3 = w_1 + w_4 + w_6 \]  

Where,

\[ w_1 = x_1 \]  
\[ w_2 = x_2 \]  
\[ w_3 = x_3 \]  
\[ w_4 = x_4 \]  
\[ w_5 = x_3(x_1 + x_4) \]  
\[ w_6 = x_1(x_3(x_1 + x_4) + x_2) \]  

3. Problem Statement

The main challenge to low multiplicative complexity logic design is in finding the circuit construction with the absolute minimal number of AND gates. As mentioned in Section 2.3, minimal-AND implementations are achieved through a randomized selection process in the original approach [22]. However, the random nature of the selection algorithm comes with some inevitable compromises:

- For a specific function, there may exist more than one solutions that are optimal in terms of multiplicative complexity but some may be superior in terms of gate count. However, the user has no control over the single result produced by the algorithm in each execution.

- A large number of trials have to be executed to be able to differentiate the better implementations from the rest, thus significantly increasing the true computation time.
• Considering the reliance on randomness to discover the optimal solution, the algorithm may perform worse than exhaustive search in the worst case scenario.

In [25], notable issues including result consistency and execution time of the algorithm are discussed, followed by proposals to improve the algorithm in these regards. However, it is desirable to eliminate the dependency on randomness to achieve low multiplicative complexity logic design in favor of a deterministic approach. Motivated by the potential hardware savings achieved by the heuristic in previous works, we explore a different approach to discover low multiplicative complexity implementations for logic functions.

4. Decomposition of Positive Polarity Reed-Muller Expressions

Function decomposition or factorization can be applied on arithmetic expressions to reduce the complexity of the problem. Since Boolean expressions over the logic basis (AND, XOR, NOT) are essentially modulo-2 arithmetic over $GF(2)$, the same approach can be applied to achieve low multiplicative complexity implementations. Note that similar concept was applied in the MIS heuristic [20] to optimize OR-AND circuits.

In this section, we discuss the relevance of decomposing PPRM expressions on the multiplicative complexity of a function.

**Example 2.** Given $f = x_1x_2 + x_2x_3$,

$$f = x_2(x_1 + x_3)$$  \hspace{1cm} (10)

Example 2 shows a simple case of decomposition to achieve reduction in the number of multiplication. However, for functions with higher degree and number of variables, the reduction in number of multiplication requires further manipulation. For this purpose, we define three important sections for a decomposed expression. Given $f \in \langle x_1, x_2, ..., x_n \rangle$ and let $x_i$ be the literal that is chosen as the factor,

$$f = \left( \frac{x_i}{a} \right) \left( \frac{f_1}{b} \right) + \left( \frac{f_2}{c} \right)$$  \hspace{1cm} (11)

Where,

- $a = \text{multiplier}$
- $b = \text{factored expression}$
- $c = \text{remainder}$
Lemma 3. Given \( n \leq 5 \) and a function in PPRM factored form \( f = (x_i)(f_1) + f_2 \), the multiplicative complexity of \( f_1 \) is less than the multiplicative complexity of \( f \), i.e. \( c_\Lambda(f_1) < c_\Lambda(f) \).

Proof. Let \( f' \) be a disjunction of cubes in \( f \) which contains the literal \( x_i \). \( n \) be the number of variables in \( f' \) and \( d \) be the degree of \( f' \). Since \( f' \) is a part of \( f \), \( c_\Lambda(f') \leq c_\Lambda(f) \).

Factorization of \( x_i \) forms \( f_1 \) with \( n - 1 \) variables and a degree of \( d - 1 \). Lemmas 1 and 2 imply that \( c_\Lambda(f_1) = c_\Lambda(f') - 1 \). Since \( c_\Lambda(f') \leq c_\Lambda(f) \), this allow us to conclude that \( c_\Lambda(f_1) < c_\Lambda(f) \). \( \square \)

Lemma 3 shows that by decomposing the PPRM expressions through the factorization of a literal \( x_i \), a factored expression \( b \) with lower multiplicative complexity can be formed. By repeating the same decomposition process on the factored expression \( b \), we would eventually arrive at a factored expression of degree \( d = 1 \). The number of decomposition steps required to reach this factored expression would give the minimal AND gates required to realize the desired function. Since each decomposition reduces the degree of the expression by one, this agrees with the lower bound rule in Lemma 1.

However, this is procedure is only possible if the remainder \( c \) does not require additional multiplication. In other words, \( c \) has to be either (a) free of multiplication or (b) all multiplications in \( c \) must exist in \( b \) so as to not cost additional AND gates. To achieve this, it is possible to alter the expression in \( c \) without additional AND gates through addition of literals in multiplier \( a \).

\[
f = \frac{x_i + x_j}{a} \left( f_1 \right) + \frac{b}{(f_2) + x_j(f_1)} \quad \text{(12)}
\]

In (12), we observe that the addition of the literal \( x_j \) into \( a \) results in the compensating expression \( x_j(f_1) \) to be added to \( c \) to maintain equality. Due to the cancellation property of XOR, this process allows the manipulation of cubes in \( c \) without introducing any AND gates to the circuit.

Therefore, if a valid compensating expression exists at each stage of the decomposition, the process will result in a minimal-AND solution. In Example 3, we demonstrate the process to obtain one possible optimal solution on the most significant output bit of the PRESENT S-box [26] using the decomposition and remainder manipulation processes.
Example 3. Given $f \in \langle x_1, x_2, ..., x_4 \rangle$ and

$$f = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3 + x_1 + x_2 + x_4 + 1 \quad (13)$$

The multiplicative complexity of the function is known to be $c_\wedge(f) = 2$. We first remove any cubes with a degree of $d \leq 1$ to form $f'$. These cubes are completely linear (no multiplication) and thus can be reintroduced into the equation through additions once $f'$ is optimized. Note that this process is trivial in the actual algorithm and only serves to simplify the current discussion.

$$f' = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3 + x_1 + x_2 + x_4 + 1$$

(14)

From (14), factorizing $x_2$ will result in the expression in (15).

$$f' = x_2 \left( \frac{x_1x_3 + x_1x_4 + x_3}{a} + \frac{x_1x_4x_3 + x_3}{b} \right) \quad (15)$$

Further decomposition of $b$ by factorizing $x_1$ gives

$$f' = x_2 \left( \frac{x_1(x_3 + x_4)}{a} + \frac{x_1x_3x_4}{c} \right) \quad (16)$$

By adding $x_1$ into $a$,

$$f' = \left( \frac{x_2 + x_1}{a} \right) \left( \frac{x_1(x_3 + x_4)}{b} + \frac{x_1x_3x_4 + x_1x_3 + x_1x_4 + x_1x_3}{c} \right) \quad (17)$$

Completing the XOR operations within $c$ results in (18).

$$f' = \left( \frac{x_2 + x_1}{a} \right) \left( \frac{x_1(x_3 + x_4)}{b} + \frac{x_1x_3x_4 + x_1x_3}{c} \right) \quad (18)$$

Since $c$ in (18) still requires additional AND gates after $b$, by adding the literal $x_3$ into $a$,

$$f' = \left( \frac{x_2 + x_1 + x_3}{a} \right) \left( \frac{x_1(x_3 + x_4)}{b} + \frac{x_1x_3x_4 + x_1x_3 + x_1x_4 + x_3}{c} \right) \quad (19)$$

$$f' = \left( \frac{x_2 + x_1 + x_3}{a} \right) \left( \frac{x_1(x_3 + x_4)}{b} + \frac{x_1(x_3 + x_4)}{c} + x_3 \right) \quad (20)$$
Since the \( d > 1 \) cubes in \( c \) is exactly identical to the cubes in \( b \), no extra AND gates are required and the entire expression in (20) can be realized with two AND gates, i.e. optimal in terms of multiplicative complexity. Reintroducing the linear section of the circuit removed earlier in this example gives the optimized expression (21).

\[
f = (x_2 + x_1 + x_3) (x_1 (x_3 + x_4) + x_3) + x_1 (x_3 + x_4) + x_3 + x_1 + x_2 + x_4 + 1 (21)
\]

It is important to note that although the length of (21) may suggest a large logic circuit, a huge portion of the equation are linear strings of XORs which share repeated cubes. As such, the actual logic circuit required to realize the expression after solving the linear step SLP problem is much smaller due to circuit sharing.

5. Tree Search Algorithm for Minimal-AND Implementation

PPRM decomposition and remainder manipulation allow the transformation of a single function to its minimal-AND form. To discover minimal-AND solutions for a given truth vector, we propose a tree search algorithm based on the decomposition of PPRM expressions.

Given a function \( f \in \langle x_1, x_2, ..., x_n \rangle \) as the root of the tree diagram, \( n \) children branches are formed from \( f \). Each child represents a unique set of multipliers \( a \), factored expressions \( b \) and remainders \( c \), formed through the factorization of the literal \( x_i \) as demonstrated in (11). From each child \( f_i \), breadth-first expansion are performed by applying the same decomposition process on its factored expression \( b \). This is repeated until the new factored expression formed is of degree \( d = 1 \). Since each level of branching reduces the degree of the factored expression by one, the height of the resulting tree diagram would be equal to degree \( d - 1 \) of the original function \( f \). Figure 2 shows an example of the tree diagram for a function \( f \) with \( n = 3 \) and \( d = 3 \).

Recall that the lower bound for multiplicative complexity of a function is proved to be \( d - 1 \) (Lemma 1). Each leaf of the tree diagram may achieve this optimal construction if it requires only one AND gate per height level to form the root function \( f \). To determine if each child vertex is able to form the parent vertex with only one AND gate, the algorithm attempts to manipulate the remainder \( c \) of the child vertex to be multiplication free through the addition of literals in \( a \) as demonstrated in (12). This verification process is attempted on every leaf of the expanded tree diagram. If all vertices on the
Figure 2: Tree diagram for an \( n = 3, d = 3 \) function.

path between a leaf and the root satisfy the one AND gate per level rule, a minimal-AND solution is obtained. On the contrary, if at any point on the path where a vertex is found to be unable to form the factored expression of its parent vertex with only one AND gate, the vertex and its descendants are all discarded from the tree diagram.

Let \( \pi_0, \pi_1, \pi_2, \ldots, \pi_{2^n-1} = 0, x_1, x_2, x_1 + x_2, \ldots, x_1 + x_2 + \ldots + x_n \). Algorithm 1 describes the procedure to verify whether the remainder of a vertex can be manipulated to be multiplication-free. Algorithm 2 gives the procedure to check every leaf of the expanded tree for potential solutions.

5.1. On Interchangeable Factors

At every stage of decomposition, the algorithm needs to attempt factorization of all literals to discover all possible solutions. As such, from each vertex, there exists a number of children branches equal to the number of variable remaining in the factored expression \( b \) of the vertex. However, if two or more literals have the same frequencies in every degree \( d > 1 \) cubes in the PPRM expression, they are mutually interchangeable and the algorithm need only attempt factorization for one of the interchangeable literals.

For example, Table 1 gives the frequency spread of literals for (14) in Example 3. Notice that literals \( x_2 \) and \( x_3 \) have the identical frequency spread.
Algorithm 1 Verifying a multiplication-free remainder
1: begin
2: valid = 0
3: for \( i = 0 \) to \( 2^n - 1 \) do
4: \( \text{comp} = (\pi_i)(b) \)
5: \( \text{newc} = c + \text{comp} \)
6: if degree of \( \text{newc} \leq 1 \) then
7: \( \text{valid} = 1 \)
8: else if all \( d > 1 \) cubes in \( \text{newc} \) exist in \( b \) then
9: \( \text{valid} = 1 \)
10: end if
11: end for
12: return valid
13: end

Algorithm 2 Checking each leaf for solution
1: begin
2: for \( i = 1 \) to number of leaves do
3: \( \text{level} = \) depth of leaf
4: while \( \text{level} > 0 \) do
5: run Algorithm 1 on current vertex
6: if \( \text{valid} = 1 \) then
7: \( \text{level} = \text{level} - 1 \)
8: if \( \text{level} = 0 \) then
9: record solution in \( S \)
10: end if
11: else
12: exit
13: end if
14: end while
15: end for
16: return \( S \)
17: end
Table 1: Frequencies of literals in each degree for (14)

<table>
<thead>
<tr>
<th>Literal</th>
<th>Degree</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 3 2 1</td>
<td></td>
</tr>
<tr>
<td>$x_1$</td>
<td>0 3 0 0</td>
<td>3</td>
</tr>
<tr>
<td>$x_2$</td>
<td>0 2 1 0</td>
<td>3</td>
</tr>
<tr>
<td>$x_3$</td>
<td>0 2 1 0</td>
<td>3</td>
</tr>
<tr>
<td>$x_4$</td>
<td>0 2 0 0</td>
<td>2</td>
</tr>
</tbody>
</table>

and are thus mutually interchangeable. Since factorizing $x_2$ first produced the result in (20), by nature of the interchangeable property, we can replace all instances of $x_2$ with $x_3$ in (20) and vice versa to obtain an alternative optimal solution (22).

$$f'(x) = (x_3 + x_1 + x_2)(x_1(x_2 + x_4) + x_2) + x_1(x_2 + x_4) + x_2 \quad (22)$$

As such, when interchangeable literals are present, the algorithm only needs to perform decomposition using one of the literals to effectively discover the associated solutions. This property allows the reduction of number of branches from a vertex when the conditions are met.

6. Enabling Product Sharing in Multiple-Output Problem

A multiple-output problem implies a set of outputs that are described as functions of the same set of inputs. Consider a set of functions $\{f_1, f_2, ..., f_n\} \in F$, it is natural to expect the multiplicative complexity of the problem to be at most the sum of the multiplicative complexity of each individual function, i.e., $c_S(F) \leq \sum_{i=1}^{n} c_S(f_i)$. This upper bound is easily achievable by solving each function individually using the algorithm proposed in Section 5. However, it is possible to achieve a smaller number of AND gates than the upper bound due to product sharing between functions.

The benefit of product sharing is obvious: for each product shared between two functions of a multiple-output problem, the number of AND gates required is reduced by one from the upper bound. Since there usually exist more than one minimal-AND solutions for each function, one way of achieving product sharing is to utilize a matching algorithm to analyze the product set of all discovered solutions between the functions. The algorithm chooses the best solution for each function that maximizes the number of
products shared between all functions. While this approach may seem to be
the easiest method to achieve product sharing, the overall solution for the
multiple-output problem is rarely close to optimal. This is due to the fact
that (a) the product set for each function is only optimal within the scope
of the single-output problem, and (b) the logic minimization algorithm does
not consider utilizing free products from previously solved functions when
minimizing a function.

To facilitate a more effective form of product sharing into the tree search
algorithm, there need to be instances where products from previously solved
functions can be introduced to modify the current function through addition
(not increasing the number of multiplication). For this purpose, we identify
stages during the decomposition process and the remainder manipulation
process where the addition of free products are meaningful.

First, we consider the PPRM decomposition process. Let \( p_1, p_2, ..., p_m \in P \) represent the collective product set obtained from solving previous func-
tions and \( f \) be the function currently undergoing decomposition. It is possible
to change the function-to-be-decomposed completely by adding elements of
\( P \) to the function \( f \). Let \( f' \) be the new function resulting from the addition
of \( p_i \) to the function \( f \).

\[
f' = f + p_i \quad (23)
\]

If the degree \( d_{f'} < d_f \), AND gate savings can potentially be achieved
due to Lemma 1 compared to solving \( f \) without utilizing the product set
\( P \). In the actual approach, we also consider all \( f' \) with \( d_{f'} = d_f \) to be valid
function for decomposition as well, since they are not proven to require more
multiplication at this point. This method of product addition can be applied
on every stage of decomposition to achieve the aforementioned benefit.

Next, we consider the potential use of the collective product set \( P \) in the
remainder \( c \) manipulation process. In Section 4, we discussed the addition
of literals to the multiplier \( a \) in order to manipulate the expressions in the
remainder \( c \) into a form that does not require additional multiplication to
be implemented. The free products in \( P \) allows further manipulation of the
remainder in two ways.

\[
f = (x_i + p_i) (f_1) + (f_2 + p_1 f_1) \quad (24)
\]

Adding \( p_i \) into the multiplier \( a \) results in unique compensating expression
This gives more options to manipulate the remainder $c$ to be multiplication free compared to adding only literals to $a$.

$$f = \left(\prod_{a \in x_i} (f'_2) + (f_2 + p_i) \right)$$

(25)

Alternatively, adding $p_i$ directly to $c$ also achieves the same purpose.

7. Tree Search Algorithm with Product Sharing

We modify the tree search algorithm described in Section 5 to incorporate the use of the collective product set $P$. Between each stage of decomposition, an additional level of branching is introduced for product addition.

Figure 3 briefly illustrates the new tree search algorithm that includes product sharing. Note that due to the inclusion of extra levels of branching, the decomposition process now occurs once per two-depth level of the tree diagram. As the number of AND gates is associated with the levels of decomposition, the cost of AND gates is now represented as one per two-depth level.

From the fully expanded tree diagram, the algorithm examines each leaf to verify if each stage of decomposition can be reversed with one AND gate. The remainder manipulation process is more rigorous than the process described in Section 5 to incorporate the use of free products. Instead of adding only literals into the multiplier $a$, the algorithm uses elements of $P$ as demonstrated in (24) and (25) to further manipulate the remainder $c$, increasing the potential number of good solutions. Let $y$ be an element of the joint set of literals and products to be added to $a$, i.e.

$y_1, y_2, \ldots, y_n, p_1, p_2, \ldots, p_m$,

and

$\pi_0, \pi_1, \pi_2, \pi_3, \ldots, \pi_{2^n + m - 1} = \pi_0, \pi_1, \pi_2, \pi_3, \ldots, \pi_{2^n + m - 1} = 0, y_1, y_2, y_1 + y_2, \ldots, y_1 + y_2 + \ldots + y_n + p_1, \ldots, p_m, p_1, \ldots, p_m$. Algorithm 3 gives the procedure to verify whether the remainder of a vertex can be manipulated to be multiplication-free with consideration for product sharing between functions.

7.1. Discarding Higher Depth Solutions

Due to product sharing, certain branches of the tree diagram will potentially terminate earlier than the others. This occurs when a shared product successfully reduces the multiplicative complexity of a function before decomposition. Therefore, when examining each leaf for potential solution, the
algorithm prioritizes leaves with lower depth. When a leaf is verified to be a solution, all remaining leaves with higher depth can be discarded as they would cost more AND gates to be implemented. Algorithm 4 describes the procedure to check every leaf for potential solutions with prioritization for lower depth leaves.
Algorithm 3 Verifying a multiplication-free remainder in multiple-output problems

1: begin
2: valid = 0
3: for $i = 0$ to $2^{n+m} - 1$ do
4:   $comp = (\pi_i)(b)$
5:   $newc = c + comp$
6: for $j = 0$ to $2^n - 1$ do
7:   $newc = newc + \theta_j$
8:   if degree of $newc \leq 1$ then
9:     valid = 1
10:   else if all $d > 1$ cubes in $newc$ exist in $b$ then
11:     valid = 1
12: end if
13: end for
14: end for
15: return valid
16: end

8. Performance

8.1. Time Complexity

We first determine the time complexity of Algorithm 3 as the sub-algorithm for the proposed algorithm (Algorithm 4). In this case, it is important to note that the time complexity $T(n, m)$ is a function of two variables, where $n$ is the number of input bits and $m$ is the number of elements in the product set. The key operation in the nested loop for Algorithm 3 is the mod-2 addition of expressions. This operation is executed by first concatenating the two addends, followed by sorting the cubes and removing consecutive duplicates through scanning (this simulates the process of XOR-ing cubes between expressions). It has a time complexity of $O(2^n \log(2^n))$. Considering the number of iterations for each loop in Algorithm 3 to be $2^{n+m}$ and $2^m$ respectively, we can deduce the overall time complexity of Algorithm 3 to be $O(2^{2(n+m)} \log(2^n))$.

Remark. A general algorithm to eliminate duplicates through sorting and scanning has a known time complexity of $O(n \log n)$. However, the variable $n$ in this context refers to the number of elements to be sorted. Mod-2
Algorithm 4 Checking each leaf for solution in multiple-output problems
1: begin
2: sort leaves in ascending order of depth
3: next = 1
4: for \( i = 1 \) to number of leaves do
5:  \( level = \) depth of leaf
6:  while \( level > 0 \) do
7:     run Algorithm 3 on current vertex
8:     if \( valid = 1 \) then
9:         \( level = level - 1 \)
10:    if \( level = 0 \) then
11:        record solution in \( S \)
12:        add all used products into \( P \)
13:        next = 0
14:     end if
15:  else
16:     exit
17:  end if
18: end while
19: if all leaves in current depth are done then
20:  if \( next = 0 \) then
21:     exit
22: end if
23: end if
24: end for
25: return \( S \)
26: end

addition in Algorithm 3 involves sorting of cubes. Given \( n \) as the number of input bits by our initial definition, the maximum number of cubes to be sorted is thus \( 2^n \). Hence, the time complexity of mod-2 addition is deduced to be \( O(2^n \log(2^n)) \).

The proposed algorithm runs Algorithm 3 recursively in a two-level nested loop. The number of iterations for each loop is determined by the tree depth and the number of leaves respectively:

- Given the upper bound for multiplicative complexity drawn in Lemma
2, we can determine the tree depth to be \( n - 1 \) in the worst-case.

- From the branching process illustrated in Figure 3, we can deduce the number of leaves to be at most \( m^{n-1}(n!) \).

As a result, the worst-case time complexity of the proposed algorithm is \( O(n(m^{n-1})(n!)(2^{(n+2m)}) \log(2^n))) \). For comparison, Algorithm 2 has a worst-case time complexity of \( O(n(n!)(2^{2n}) \log(2^n)) \) without the product sharing feature.

8.2. Computation Time

Comparison of time complexity with the original non-linear step is difficult as the worst-case time complexity of the original randomized selection algorithm is unbounded. To compare the performance of both approaches, we implemented both algorithm using MATLAB R2012b with the Intel Core i5-4690 processor @ 3.50GHz and 8GB of RAM to compare the computation time required by each algorithm. To reduce the impact of external factors in the experiment, the computation time reported is taken as an average of 100 test cases. We choose the four functions of the PRESENT S-Box to be optimized as a pseudo representation of real world application. We also include a \( c_A(f) = 3 \) function challenged in [8] (labeled as \( f_5 \)). The truth vectors of the functions are given as follow:

\[
\begin{align*}
    f_1 &= [0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0] \\
    f_2 &= [0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0] \\
    f_3 &= [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0] \\
    f_4 &= [1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0] \\
    f_5 &= [0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1] \\
\end{align*}
\]

For performance comparison on multiple-output problems, both algorithms are applied to optimize the full PRESENT S-Box, i.e. \( f_1, f_2, ..., f_4 \in F_{\text{PRESENT}} \). The same tests are also applied to Canright’s \( GF(2^4) \) AES S-Box [27]. The results are reported in Table 2.

8.3. Discussion

From Table 2, we observed over 85% computation time reduction in each case when compared to the original randomized approach. This demonstrates the main advantage of the proposed algorithm. When randomly selecting the
Table 2: Comparison of computational time

<table>
<thead>
<tr>
<th>Function</th>
<th>$c_n(f)$</th>
<th>Computation Time (s)</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>Proposed</td>
</tr>
<tr>
<td>$f_1$</td>
<td>1</td>
<td>0.0638</td>
<td>0.0093</td>
</tr>
<tr>
<td>$f_2$</td>
<td>2</td>
<td>5.3874</td>
<td>0.5107</td>
</tr>
<tr>
<td>$f_3$</td>
<td>2</td>
<td>7.9096</td>
<td>0.4087</td>
</tr>
<tr>
<td>$f_4$</td>
<td>2</td>
<td>7.3758</td>
<td>0.4743</td>
</tr>
<tr>
<td>$f_5$</td>
<td>3</td>
<td>688.4390</td>
<td>26.2880</td>
</tr>
<tr>
<td>$F_{PRESENT}$</td>
<td>4</td>
<td>931.7820</td>
<td>64.5891</td>
</tr>
<tr>
<td>$F_{AES}$</td>
<td>5</td>
<td>4041.3000</td>
<td>147.3512</td>
</tr>
</tbody>
</table>

pair of signals for addition or multiplication, the randomized approach has the possibility of forming a sum or product that already existed in the current sample space. This occurs not only when the same pair of signals are selected repeatedly, but also due to the properties of XOR and AND where different signal pairs may also produce the same sum or product. As discussed in [25], this expands the sample space and reduce the probability to obtain the desired solution. Naturally, this results in longer computation time for the algorithm. This problem does not exist in the proposed algorithm.

As expected, an exponential increase in computation time is observed for both algorithms as the multiplicative complexity of the problem increases. However, due to the much faster computation time of the proposed algorithm, the exponential increase has a lesser impact on the practicality of the algorithm. Most notably, when solving for the full PRESENT and $GF(2^4)$ AES S-Boxes, the proposed algorithm showed a tangible benefit compared to the randomized approach.

We take this opportunity to note an important benefit of the proposed algorithm when solving for multiple-output problems: The proposed algorithm does not need to return to reevaluate previously solved functions. When solving a function using the proposed tree search algorithm, each execution collects all possible solutions discovered (with the least amount of AND gates) and their respective product set. If a product set is determined to be unable to solve the subsequent functions with equal or less AND gates than another product set, the former is discarded and the algorithm proceeds with the remaining product sets. On the contrary, the randomized approach gives one random solution per execution. The product set resulting from this solution
may not be relevant for subsequent functions. However, this problem can only be identified once the algorithm finished solving all functions and compared the number of AND gates against the known minimal. When identified, the algorithm has to return to solve the problem from the first function. This scenario demonstrates another advantage the proposed algorithm has over the randomized approach.

9. Quality of Results

As a heuristic to low multiplicative complexity logic minimization, we note that the proposed approach is generally unable to discover all possible optimal solutions. One important factor is due to the algorithm assuming all factorized expressions to be in their minimal form when factorizing the first literal $x_i$, i.e. given $f = (x_i)(f_1) + f_2, x_i \notin f_1$.

**Example 4.** Given $f = x_1x_2x_3 + x_1x_2 + x_1$,

\[
\begin{align*}
    f &= x_1(x_2x_3 + x_2) + x_1 \quad (26) \\
    f &= x_1(x_1x_2x_3 + x_2) + x_1 \quad (27) \\
    f &= x_1(x_2x_3 + x_1x_2) + x_1 \quad (28)
\end{align*}
\]

From Example 4, (26) represents the minimal form (which the proposed algorithm always assumes) whereas (27) and (28) represents the non-minimal form. When solving for a single-output function, minimal form factorization is strictly better than non-minimal factorization in producing minimal-AND results. However, when a multiple-output problem is concerned, the same statement is no longer true due to the potential of product sharing. The additional AND costs remained in the non-minimal form may be nullified by product sharing from the previous or subsequent functions, resulting in an optimal solution. However, checking for every possible non-minimal factorization introduces another layer of exponential complexity to the algorithm. The added complexity serves to discover optimal solutions which may or may not exist for a multiple-output problem.

Secondly, when attempting the PPRM decomposition method described in Sections 4 and 6, the proposed algorithm only considers compensating expressions generated from adding a combination of input literals and/or elements of the collective product set into the multiplier section $a$. This excludes the compensating expressions formed by adding $2^n$ potential cubes.
Table 3: Comparison of logic gate count

<table>
<thead>
<tr>
<th>S-Box</th>
<th>Implementation</th>
<th>XOR</th>
<th>AND</th>
<th>NOT</th>
<th>OR</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^4)$ AES</td>
<td>Boyar [8]</td>
<td>11</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Boyar [29]</td>
<td>10</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT</td>
<td>Courtois [28]</td>
<td>20</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Courtois [28]</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

into $a$ unless they are elements of the product set. In addition to the obvious increase in algorithm complexity, the reasoning against this approach is that cubes innately requires additional AND gates as a conjunction of literals. As the algorithm has no means of knowing whether a cube can be shared with subsequent functions, adding a cube into $a$ often result in non-optimal solutions. In fact, the proposal to add elements of the product set into $a$ is to slightly compensate for this issue using products or cubes that have already be determined to be required for solved functions.

Despite the points above, when examining the solutions obtained from both algorithms from the experiment in Section 8, we observed that the best results obtained from both algorithms are often identical in terms of gate count, save for the fact that it is the matter of chance whether the randomized approach is capable of discovering the best solution within the limited number of trials. We noted that works in [8] and [28] have applied low multiplicative complexity algorithm to optimize the $GF(2^4)$ AES and PRESENT S-Box respectively. Table 3 compares the smallest results achieved by the proposed algorithm to the aforementioned works.

Comparing the results on the AES S-Box, it appears that our best result achieved the reduction of one XOR gate compared to the implementation proposed in [8]. The proposed algorithm also discovered more than one implementation of the $GF(2^4)$ S-Box using 10 XOR gates and 5 AND gates. Upon further inspection, we noted that the critical path of the implementation in [8] consists of 5 XOR gates and 4 AND gates (depth-9), while the best solution discovered by the proposed algorithm has a critical path of 5 XOR and 3 AND gates (depth-8). The best implementation discovered by the pro-

\[^{3}\text{After further optimization through affine AND-OR replacement.}\]
posed algorithm is described in Figure 4. We also reference the work in [29], where a depth-4 implementation is achieved at a gate count of 10 XOR gates and 7 AND gates. While the implementation is not exactly optimal in terms of multiplicative complexity, it shows that improvement in other metrics can be achieved at the cost of increased gate count when constructed with more AND gates than the absolute minimum.

In [28], the author reported a low multiplicative complexity implementation of the PRESENT S-Box with 25 gates. To further improve on the result, $2^4$ possible cases of replacing some of the AND gates with OR gates (they are affine equivalents) are attempted and a final implementation with 14 gates is reported. When comparing the results, it is observed that the best solution discovered by the proposed algorithm requires one additional NOT gate. However, we note that the same logic gate replacement technique can improve the gate count of our results. In fact, due to $f_3$ and $f_4$ of the PRESENT S-Box being negative functions ($f(0) = 1$), the two NOT gates in our results can be combined with AND or XOR gates in the circuit to be implemented as NAND or XNOR gate respectively as shown below:

\[
a \odot b \oplus 1 = (a \odot b)' = NAND(a, b) \\
a \oplus b \oplus 1 = (a \oplus b)' = XNOR(a, b)
\]

Therefore, a small adjustment can be applied on our result to produce an implementation requiring only 13 gates, the lowest gate count reported for the PRESENT S-Box to date to the best of our knowledge. Both implementations before and after gate replacement are given in Figures 5 and 6 respectively.

Regardless, the purpose of this discussion is to show that the proposed algorithm is able to achieve comparable results with previous works, even when restricted to utilizing only the logic basis (AND, XOR, NOT). However, it should be noted that given infinite number of trials, the randomized

<table>
<thead>
<tr>
<th>$t_1 = x_2 \odot x_4$</th>
<th>$t_2 = x_1 \oplus x_2$</th>
<th>$t_3 = t_1 \oplus t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_4 = x_3 \odot t_3$</td>
<td>$t_5 = x_3 \oplus x_4$</td>
<td>$y_1 = t_4 \oplus t_5$</td>
</tr>
<tr>
<td>$t_6 = t_4 \oplus t_1$</td>
<td>$t_7 = t_6 \odot t_5$</td>
<td>$y_2 = x_4 \oplus t_7$</td>
</tr>
<tr>
<td>$t_8 = t_5 \oplus t_1$</td>
<td>$t_9 = x_1 \odot t_8$</td>
<td>$y_3 = t_2 \oplus t_9$</td>
</tr>
<tr>
<td>$t_{10} = t_1 \oplus y_3$</td>
<td>$t_{11} = t_{10} \odot t_2$</td>
<td>$y_4 = x_1 \oplus t_{11}$</td>
</tr>
</tbody>
</table>

Figure 4: Depth-8 size-15 implementation of $GF(2^4)$ AES S-Box.
\[
\begin{align*}
  t_1 &= x_2 \oplus x_3 \quad t_2 = t_1 \odot x_3 \quad t_3 = x_4 \oplus t_2 \\
y_1 &= x_1 \oplus t_3 \quad t_4 = t_1 \odot t_3 \quad t_5 = x_3 \oplus t_4 \\
t_6 &= x_1 \oplus t_5 \quad t_7 = t_5 \odot x_1 \quad t_8 = t_7 \oplus t_1 \\
t_9 &= y_1 \oplus t_8 \quad y_4 = t'_9 \quad y_2 = t_6 \oplus t_9 \\
t_{10} &= y_2 \odot t_6 \quad t_{11} = t_{10} \oplus t_3 \quad y_3 = t'_{11}
\end{align*}
\]

Figure 5: Size-15 implementation of PRESENT S-Box.

\[
\begin{align*}
  t_1 &= x_2 \oplus x_3 \quad t_2 = t_1 \odot x_3 \quad t_3 = x_4 \oplus t_2 \\
y_1 &= x_1 \oplus t_3 \quad t_4 = t_1 \odot t_3 \quad t_5 = x_3 \oplus t_4 \\
t_6 &= x_1 \oplus t_5 \quad t_7 = t_5 \odot x_1 \quad t_8 = t_7 \oplus t_1 \\
y_4 = y_1 \oplus t_8 \quad y_2 = (t_6 \oplus y_4)' \quad t_9 = (y_2 \odot t_6)'
\end{align*}
\]

Figure 6: Size-13 implementation of PRESENT S-Box after gate replacement.

approach should theoretically achieve the same results (if not potentially better) as the proposed algorithm. We are also unable to prove that no better results exist over the logic basis (AND, XOR, NOT). We only claim that the proposed algorithm is capable of producing sufficiently good results in practical time.

9.1. On Majority Function

A majority function of \( n \) inputs is \textit{true} when at least \( \frac{n}{2} \) of its inputs are true and vice versa in the case for \textit{false}. As reported in [28], finding an optimal representation in terms of multiplicative complexity for a majority function when \( n \) is odd can be difficult.

With \( n = 3 \), our proposed tree search algorithm discovered six optimal solutions for the majority function, each is a size-4 circuit requiring one AND gate and three XOR gates. This is in line with the results reported in [28].

With \( n = 5 \) however, the tree search algorithm discovered a total of 5760 optimal solutions. Most notably, 1440 of the discovered solutions are size-12 (3 AND gates and 9 XOR gates). This is smaller than the optimal solution reported in [28] by one XOR gate. While a single gate is by no means a significant improvement, it is important to show that the proposed algorithm is not only capable of solving for optimal multiplicative complexity.
\begin{align*}
t_1 &= x_3 \oplus x_4 \\
t_2 &= x_1 \oplus x_2 \\
t_3 &= x_2 \oplus x_3 \\
t_4 &= x_4 \oplus x_5 \\
t_5 &= t_1 \odot t_4 \\
t_6 &= t_5 \oplus t_1 \\
t_7 &= t_4 \oplus t_6 \\
t_8 &= t_3 \odot t_7 \\
t_9 &= t_8 \oplus t_6 \\
t_{10} &= x_3 \oplus t_8 \\
t_{11} &= t_2 \odot t_9 \\
y &= t_{11} \oplus t_{10}
\end{align*}

Figure 7: A size-12 implementation of the majority function with $n = 5$.

solutions, but also smaller solutions compared to the alternative approach.
One instance of the size-12 solution is shown in Figure 7.

10. Conclusion and Future Work

In this paper we presented a novel approach to achieve low multiplicative complexity logic design for logic minimization problems with up to five input variables. To solve for economical circuits in practical computation time, a tree search algorithm is proposed utilizing the PPRM decomposition method and knowledge on the lower and upper bounds of multiplicative complexity. To enable further AND gate reduction in a multiple-output problem, product sharing between functions of the same inputs are explored and incorporated into the tree search algorithm. Comparison with the original randomized approach showed over 85% reduction in computation time when solving for both single and multiple-output problems. The results produced are also competitive with the best implementations previously reported using the low multiplicative complexity heuristic even after further optimization.

For further research, there is the potential to reduce the time complexity of the proposed algorithm by exploring properties of the PPRM decomposition process. Through the time complexity analysis of the proposed algorithm, we can identify that the number of leaves in the tree search algorithm is the major contributor to the overall complexity of the algorithm. As such, useful properties of mod-2 arithmetic that allow the algorithm to exclude any number of literals and/or products during the branching process can significantly improve the performance of the algorithm.

There is also the incentive to generalize the algorithm to solve for problems with more than five input variables. The fundamental requirement is that the algorithm has to be independent of the knowledge on the upper bound of multiplicative complexity. One potential approach is to solve a problem strictly using the number of multiplications given by the lower
bound. If no solutions are discovered, the algorithm loosens the criterion and allows higher number of multiplications to be used. The main challenge to this approach lies in finding an algorithm efficient enough to solve for minimal-AND solutions when the multiplicative complexity of a function is not lower bounded. It is also important to note that NP-hardness of the XOR-minimization step may become the bottleneck to the heuristic instead as the number of inputs increases.

References


29


Jia Jun Tay is a Ph.D. student at the Faculty of Engineering, Computing and Science of the Swinburne University of Technology Sarawak Campus. He received his Degree in Electrical and Electronic Engineering from the same university in 2014. His research interests include logic synthesis, integrated circuit design, information security, and lightweight cryptography.
A novel deterministic algorithm for low multiplicative complexity logic design is proposed. The algorithm eliminates reliance on randomness present in the original approach. We show over 85% reduction in algorithm computational time on practical problems. Quality of results are comparable to previous works in desirable metrics.