A Domain-Transformation Approach to Synthesize Read-Polarity-Once Boolean Functions

Vinicius Callegaro¹, Mayler G. A. Martins², Renato P. Ribas¹² and André I. Reis¹²

¹PPGC, Institute of Informatics, UFRGS
²PGMICRO, Institute of Informatics, UFRGS
Porto Alegre, RS, Brazil
e-mail: vcallegaro@inf.ufrgs.br

ABSTRACT

Efficient exact factoring algorithms are limited to read-once (RO) functions, where each variable appears exactly once at the final Boolean expression. However, these algorithms present two important constraints: (1) they do not consider incompletely specified Boolean functions (ISFs), and (2) they are not suitable for binate functions. To overcome the first drawback, an algorithm that finds RO expressions for ISF, whenever possible, is proposed. In respect to the second limitation, we propose a domain transformation that splits existing binate variables into two independent unate variables. Such a domain transformation leads to ISF’s, which can be efficiently factored by applying the proposed algorithm. The combination of both contributions gives optimal results for a recently proposed broader class of Boolean functions called read-polarity-once (RPO) functions, where each polarity (positive and negative) of a variable appears at most once in the factored form. Experimental results carried out over ISCAS’85 benchmark circuits have shown that RPO functions are significantly more frequent than RO functions.

Index Terms: Boolean functions, factoring, logic synthesis, read-once, read-polarity-once, digital circuits.

I. INTRODUCTION

Factoring Boolean functions is a fundamental operation in algorithmic logic synthesis [1][2]. Factoring is the process of deriving a parenthesized algebraic expression, or factored form, representing a given logic function, usually provided initially in sum-of-products (SOP) or product-of-sums (POS) forms. For instance, $f = a \cdot b + a \cdot c \cdot d + a \cdot c \cdot e$ can be factored into the logically equivalent equation $f = a \cdot (b + c \cdot (d + e))$.

The task of factoring Boolean functions into shorter, more compact, logically equivalent formulae is one of the basic operations in the early stages of algorithmic logic synthesis [2]. In most design styles, like conventional CMOS gates, the electrical implementation of a Boolean function corresponds almost directly to its factored expression in terms of literals and device count. Generating an optimum factored form, i.e. the shortest length expression, is an NP-hard problem [3]. Hence, heuristic algorithms have been developed in order to obtain good factored solutions [1]-[5]. Among well-known heuristic algorithms is X-Factor [3][4], which provides good results but does not guarantee minimal expressions. In [6], Lawler claims to provide the exact factoring. However, Lawler’s method is not scalable and becomes impractical even for functions with 4 variables. Recently, new approaches have improved the factoring process for exact solutions, but the scalability and runtime still remain the main bottlenecks [7-9]. Efficient and exact algorithms exist for a sub-class of functions known as read-once functions [10]-[12]. A Boolean function is considered read-once (RO) whether it can be represented in a factored form where each variable appears only once [10]. Reviewing the example above, the function $f = a \cdot (b + c \cdot (d + e))$ is RO. This class of functions is of special interest in logic synthesis since they are quite frequent in circuit design [17].

Exact algorithms for RO functions present two important limitations: (1) they do not factorize incompletely specified Boolean functions (ISF), and (2) they are not suitable for functions with binate variables. In order to overcome the first constraint, we propose an algorithm to find RO expressions for ISF, whenever possible. With respect to the second limitation, we propose a domain transformation, named here as unatization process, that splits existing binate variables into two independent unate variables. Such a domain transformation leads to ISF, which can be efficiently factored by the proposed algorithm.

The combination of both contributions gives exact factoring results for a recently proposed and broader class of functions called read-polarity-once (RPO) functions [13], where each polarity (positive
A Domain-Transformation Approach to Synthesize Read-Polarity-Once Boolean Functions
Callegaro, Martins, Ribas & Reis

II. PRELIMINARIES

A. Boolean functions

Let \( B = \{0,1\} \) and \( Y = \{0,1\} \), a completely specified Boolean function (CSF) \( f \) in \( n \)-input variables, \( x_1, \ldots, x_n \), is a function:

\[
f: B^n \rightarrow Y
\]

where \( x = [x_1, \ldots, x_n] \in B^n \) is the input of \( f \).

An incompletely specified Boolean functions (ISF) differs from completely specified functions in the fact that the former may also assume don’t-care (X) values, in addition to the binary values 0 and 1, i.e. \( Y = \{0,1,X\} \) [15].

An element \( m \in B^n \) is called term. The number of terms in \( B^n \) is \( 2^n \). The on-set \( f_{on} \) is defined as the set of terms \( m \in B^n \) such that \( f(m) = 1 \), the off-set \( f_{off} \) such that \( f(m) = 0 \) and the don’t-care set \( f_{dc} \) such that \( f(m) = X \).

Two ISF \( f \) and \( g \) are said equal when \( f \equiv g \), meaning that \( f_{on} = g_{on} \), \( f_{off} = g_{off} \) and \( f_{dc} = g_{dc} \). However, when in an ISF domain, it is often needed to verify if an ISF \( f \) is equivalent to another ISF \( g \). Two ISF \( f \) and \( g \) are equivalent \( f \equiv g \) iff \( f_{on} \cap g_{off} = \emptyset \) and \( f_{off} \cap g_{on} = \emptyset \).

Hereafter, function and Boolean function will have the same meaning in this paper.

B. Cofactor and unateness

In order to identify the unateness behavior of a variable, let us define the cofactor operation as follows. Given an \( n \)-input function \( f(x_1, x_2, \ldots, x_n) \), the cofactor of \( f \) with respect to \( x_i \) is denoted as \( f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \), and it represents the sub-function where variable \( x_i \) is assigned to a Boolean constant \( c \in \{0,1\} \) [12]. For presentation sake, let \( f(x_1, \ldots, x_{i}=c, \ldots, x_n) \) be represented by \( f(x_i=c) \). In this sense, the unateness behavior of a variable \( x_i \) can be obtained according to the following relationships:

\[
\alpha = f(x_i=1) \quad \beta = f(x_i=0) \quad \gamma = \alpha + \beta
\]

\[
\begin{align*}
\text{positive unate:} & \quad (\alpha = \gamma) \land (\alpha \neq \beta) \\
\text{negative unate:} & \quad (\beta = \gamma) \land (\alpha \neq \beta) \\
\text{don’t care:} & \quad \alpha = \beta \\
\text{binate:} & \quad \alpha \neq \beta \neq \gamma \neq \alpha
\end{align*}
\]

We say that a Boolean function is unate if all its variables are either positive or negative unate. When all variables are positive (negative) unate, we say that the function \( f \) is positive (negative) unate. In the case when at least one variable is binate, the function is considered binate.

C. Read-once Boolean functions

Read-once (RO) Boolean functions are well-known for a long time [16], and their special properties still play important role in modern VLSI circuit synthesis flow [4][11]. In [17], an extensive investigation was performed in order to evaluate the occurrence of RO functions in circuit design.

A RO form is a factored form where each variable appears exactly once. A Boolean function is RO if it can be represented by an RO form. For example, the Boolean function represented by:

\[
f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \cdot x_7 \cdot x_8
\]

is a RO function since it can be factored into:

\[
f = x_1 \cdot (x_2 + x_3 \cdot (x_4 + x_5))
\]

If a given function \( f \) can be factored into a RO form, then all variables of \( f \) are either positive or negative unate [10]. This is a necessary but not sufficient condition, since there are functions composed by only unate variables that cannot be factored into a RO form. For example, the unate function \( f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \cdot x_7 \cdot x_8 \) has \( f = x_1 \cdot (x_2 + x_3) + x_4 \cdot x_6 \) as the minimal solution, in which variables \( x_1 \), \( x_4 \) and \( x_6 \) appear more than once. If a function has at least one binate variable, this variable will appear...
with two polarities in the factored form, contradicting the definition of RO function, where each variable appears at most once.

D. Read-once factoring algorithms

The class of RO functions was firstly introduced by Hayes [16] and was called fanout-free functions. Besides the class proposal, Hayes also presented a factoring algorithm to identify and synthesize RO factored forms. His method suffers of high complexity since the algorithm makes intensive calls to a procedure to perform equivalence checking of cofactors.

Peer and Pinter in [17], have also proposed an algorithm to synthesize non-repeating literal trees, another name given to RO functions. The main drawback of their method is that it cannot run in polynomial time. This is due to the fact that their method requires several procedure calls for converting SOP forms into POS forms (and vice-versa). Such a routine requires a non-polynomial time to be executed, making the method very expensive in terms of runtime.

Golumbic, in [10], was the first to propose a polynomial time factoring algorithm for RO functions, called IROF. His method is based on the Gurvich’s work [18]. The IROF algorithm receives as input an irredundant sum-of-products (ISOP) expression and performs graph operations to achieve an RO form. The time complexity of the IROF algorithm is \( O(n | f |) \), where \(| f |\) denotes the length of the ISOP equation of a function \( f \), and \( n \) is the number of variables of \( f \) [11].

In [12], Lee and Wang proposed a new approach, referred herein as JPHI, to overcome the limitations presented in [16]. Without loss of generality, let us consider all variables of \( f \) as positive unate. Being \( f(x_1, x_2, \ldots, x_n) \) a Boolean function with \( n \)-input variables, two variables \( x_i \) and \( x_j \) can be compressed whether their cofactors are equals, i.e., \( f(x_i = c) = f(x_j = c) \). Such compression process is carried out through AND or OR operators if the value of the constant \( c \) is equal to 0 or 1, respectively. The compression of two variables results in a new variable which is reinserted into \( f \), leading to another \((n - 1)\)-input function \( \varphi \). The algorithm then looks for compression groups in \( \varphi \) until reaching (if it is possible) an RO form. The time complexity of the JPHI algorithm is \( O(n^2 K) \), where \( K \) denotes the number of products in the ISOP of a Boolean function \( f \).

Both IROF and JPHI methods factorize RO functions in polynomial time. However, if the entire function is not RO, the IROF method is not able to recognize sub-functions that are RO. The JPHI method, in turn, is able to find RO sub-functions, even if the input function is not completely RO. Furthermore, it is possible to modify the JPHI method to accept incompletely specified Boolean functions as input. Table I summarizes the comparison between IROF and JPHI algorithms.

### Table I. Comparison between IROF and JPHI algorithms.

<table>
<thead>
<tr>
<th></th>
<th>Time complexity</th>
<th>Fail fast</th>
<th>Partial RO</th>
<th>Works with ISF</th>
</tr>
</thead>
<tbody>
<tr>
<td>IROF</td>
<td>( O(n</td>
<td>f</td>
<td>) )</td>
<td>Yes</td>
</tr>
<tr>
<td>JPHI</td>
<td>( O(n^2 K) )</td>
<td>No</td>
<td>Yes</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

\( n \): number of variables in the SOP equation. 
\(| f |\): number of literals in the SOP equation. 
\( K \): number of products in the SOP equation. 
* with modifications.

III. FACTORING INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS INTO READ-ONCE EXPRESSIONS

Several methods for factoring ISF have been proposed in the literature [6][8][9], but only the Exact Factor approach guarantees exactness in the solutions [8]. However, such method can take more than 10 minutes to synthesize an expression with 12 literals, even for RO functions. This way, an efficient method for factoring ISF is still a challenge.

### A. Proposed algorithm to factorize ISF into RO forms (ISF2RO)

This section presents the proposed algorithm to factorize ISF into RO forms, whenever it is possible. Our algorithm is based on the same principle discussed in [12] and in [16] that compare cofactors in order to group variables targeting an RO form. Despite of the efficiency of the IROF algorithm, it cannot be modified in order to deal with ISF, since it depends on an ISOP form as input. That is the main reason of our choice in modifying and extending the JPHI approach [12].

**Observation 1:** Let \( f \) be an ISF. Consider that there is a RO factored form that represents \( f \). Then, there is at least one proper assignment of the don’t-care terms of \( f \) that transforms \( f \) into a CSF \( g \) which is trivially synthesized through algorithms designed for RO functions.

For instance, consider the ISF presented in Fig. 1. Since \( f \) contains two don’t care (\( X \)) terms, it is possible to assign them into four different ways, as seen in Table II.

All don’t care assignments presented in Table II
A Domain-Transformation Approach to Synthesize Read-Polarity-Once Boolean Functions
Callegaro, Martins, Ribas & Reis

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$X$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1. A truth table representing the ISF $f = 1110\ 1X10\ 11X0\ 0000$.

lead to unate functions, but only two of them result in RO forms. Unfortunately, identifying such an ISOP is not a straightforward task. An ISF can present a huge number of ISOP forms, and only the ISOPs that lead to unate functions are of interest. Notice that such condition is necessary but not sufficient, since there are unate functions that do not lead to RO forms.

In this sense, instead of running an exhaustive search for finding the correct ISOP, we have created a method that assigns the don’t care terms using a RO driven approach. Therefore, let us define some basic data structures used in our method.

**Definition 1:** An assignment is a data structure to represent the state when a variable $x_i$ was assigned to a Boolean constant $c$. An assignment is represented by a tuple $<x_i,c>$.

**Definition 2:** A logic arrangement is a data structure used to store the grouping states. This data structure is illustrated in Fig. 2.

Our algorithm, called ISF2RO, receives an ISF $f$ as input. Without loss of generality, we consider that all variables in $f$ are positive unate. The method starts by inserting the input variables into a main list of logic arrangements. Let $x_i$ be a variable in $f$. The logic arrangement $la_i$ of $x_i$ is defined as follows:

$$la_i.expression = \text{"x}_i\text{"}$$ \hspace{1cm} (11)

$$la_i.on_spec = \{ (x_i = 1) \}$$ \hspace{1cm} (12)

$$la_i.off_spec = \{ (x_i = 0) \}$$ \hspace{1cm} (13)

$$la_i.pos_cube_cof = f(la_i.on_spec)$$ \hspace{1cm} (14)

$$la_i.neg_cube_cof = f(la_i.off_spec)$$ \hspace{1cm} (15)

The proposed pseudo-algorithm is shown in Fig. 3. The method starts by filling the main list with logic arrangements representing the input variables of $f$. The initial values for each logic arrangement can be obtained as shown in Equations (11-15). These logic arrangements are cofactored (accordingly to the assignment definition) and compared to each other. The next step consists in finding two logic arrangements that have the same cube cofactor function.

In order to illustrate the idea of the proposed method, let the input ISF be $f=11101X01011X00000$. After filling the main list with the input variables, the expected list is shown in Table III.

In Table III, it is possible to see that $la_1$ and $la_2$ has equivalent positive cofactors. Similar situation occurs between $la_4$ and $la_5$. Hence, according to the lines (8-10) and (17-19) in Fig. 3, these logic arrangements could be grouped through an OR operator. There are also logic arrangements with equivalent negative cofactors: $la_3$ and $la_6$; $la_7$ and $la_8$. Such logic arrangements are grouped by an AND operator in accordance to the lines (12-14) and (17-19) in Fig. 3. In Table IV, it is

```java
1   list = create_logic_arrang_from_input_variables(f);
2   while (TRUE) {
3     for (i=0; i < |list|-1; i++) {
4       f1 = list[i];
5       for (j=i+1; j < |list|; j++) {
6         f2 = list[j];
7           if (f1.pos_cube_cof == f2.pos_cube_cof) {
8             f3.exp = f1.exp + f2.exp;
9             f3.on_spec = min(f1.on_spec,f2.on_spec);
10            f3.off_spec = f1.off_spec U f2.off_spec;
11           } elsif (f1.neg_cube_cof == f2.neg_cube_cof) {
12             f3.exp = f1.exp - f2.exp;
13             f3.on_spec = f1.on_spec U f2.on_spec;
14             f3.off_spec = min(f1.off_spec,f2.off_spec);
15           }
16        if (f3 != null) {
17            f3.pos_cube_cof = cubeCof (f3.on_spec);
18            f3.neg_cube_cof = cubeCof (f3.off_spec);
19            temp_list.add(f3);
20          }
21      }
22  }
23  if (|temp_list| == 0)
24    return FAILURE;
25  ro_instances = find_ro_expressions(temp_list);
26  if (|ro_instances| != 0)
27    return ro_instances;
28  list = list U temp_list;
29  clear(temp_list);
30 }
```

Figure 3. ISF2RO pseudo-algorithm.
A Domain-Transformation Approach to Synthesize Read-Polarity-Once Boolean Functions
Callegaro, Martins, Ribas & Reis

Table III. Expected initial main list for the input function 
\[ f = 11101X1011X00000. \]

<table>
<thead>
<tr>
<th>LA</th>
<th>exp</th>
<th>On</th>
<th>Off</th>
<th>pos_cube_cof</th>
<th>neg_cube_cof</th>
</tr>
</thead>
<tbody>
<tr>
<td>la</td>
<td>x1</td>
<td>x4</td>
<td>x6</td>
<td>1110X1011X01X0</td>
<td>110XX000011X0000</td>
</tr>
<tr>
<td>la</td>
<td>x1</td>
<td>x4</td>
<td>x6</td>
<td>11111X1X100000</td>
<td>10101010X00000</td>
</tr>
<tr>
<td>la</td>
<td>x1</td>
<td>x4</td>
<td>x7</td>
<td>11111111XX0000</td>
<td>100XX001100000</td>
</tr>
</tbody>
</table>

Table IV. Grouped logic arrangements after first iteration.

<table>
<thead>
<tr>
<th>LA</th>
<th>exp</th>
<th>On</th>
<th>Off</th>
<th>pos_cube_cof</th>
<th>neg_cube_cof</th>
</tr>
</thead>
<tbody>
<tr>
<td>la</td>
<td>x1+</td>
<td>x1</td>
<td>110XX1011X01X0</td>
<td>110XX000011X0000</td>
<td></td>
</tr>
<tr>
<td>la</td>
<td>x2</td>
<td>x4</td>
<td>x7</td>
<td>111111111111</td>
<td>110XX001100000</td>
</tr>
<tr>
<td>la</td>
<td>x1</td>
<td>x4</td>
<td>x5</td>
<td>111111111111</td>
<td>10101010X00000</td>
</tr>
</tbody>
</table>

It is easy to observe that the algorithm has an exponential time complexity. However, for this benchmark of 16-inputs functions was 500 seconds. The worst runtime observed in the experiments with the ISCAS’85 benchmark circuits grouped by input count.

IV. FACTORING INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS INTO READ-POLARITY-ONCE EXPRESSIONS

Efficient algorithms exist to perform factoring of RO formulas. Most of them readily discard functions containing binate variables, since the RO functions are always unate. This way, it becomes interesting to obtain a class of functions that comprises both unate and binate functions, as well as to obtain an efficient method to synthesize this class into minimal factored forms. Such class is defined as follows.

A. Read-polarity-once definition

Definition 3: A Boolean function is called read-polarity-once (RPO) if each polarity (positive and negative) of a variable appears at maximum once in the minimum factored form [13].

Lemma 1: a positive (negative) unate variable contributes with at least one positive (negative) literal in a factored form.

Lemma 2: a binate variable contributes with at least two literals (one positive and one negative) in a factored form.

Theorem 1: an RPO function represented by an expression is in minimum form, if each unate variable contributes with exactly one literal and each binate variable contributes with exactly two literals (one positive and one negative).

Proof: straightforward by lemmas 1 and 2, as the RPO factored form contains at most one literal per unate variable and at most two literals (one positive and one negative) per binate variables.

Observation 2: Notice that the RPO class is a superset of the RO class. Every RO function is also an RPO function, while an RPO function is not necessarily an RO function. For instance, the function \( f = (a+b)(c+(d+e)) \) is both RO and RPO, while \( f = (a+b)(c+d+e) \) is RPO but it is not RO. The RPO class contains binate functions as elements of the class, while the RO class contains only unate functions. This relationship is presented in Fig. 5.
B. Unatization process

According to Definition 3, if a function can be factored into an RPO expression, each polarity (positive or negative) of a variable appears at maximum once in the factored expression. Hence, an interesting point of investigation is if it is possible to separate the positive and negative literals, and transform the function into an unate function. Another point, equally interesting, is if that resulting transformation could be treated successfully by RO factoring algorithms.

In this sense, we propose a domain transformation (referred as unatization) that splits existing binate variables into two independent unate variables. This domain transformation leads to ISF, which can be factored efficiently by the algorithm proposed in Section III. The combination of both contributions provides exact results for the recently proposed class of RPO functions [13].

**Example 1:** The unatization method receives as input an ISF and split all binate variables into two independent unate variables. Let \(a\) and \(b\) be binate variables from \(f=(a+b)\cdot(\overline{a}+\overline{b})\). In order to unatize \(f\), we introduce independent variables to represent the negative unate literals. Hence, by introducing variables \(na\) and \(nb\), a domain transformation is performed and the function becomes a 4-input function, with most of the terms appearing as don’t cares, as seen in Fig. 6.

The function represented by the truth table shown in Fig. 6 is not positive unate. By computing the cofactors of the function and setting the don’t care values to force the function to become positive unate

```plaintext
unatization(f\textsuperscript{ ON }, f\textsuperscript{ NC }) {
  for (int i = 0; i < n; i++)
    if (is_binate(x\textsubscript{i}))
      createVariable(not\_x\textsubscript{i});
      XNOR = !(x \oplus not\_x\textsubscript{i});
      f\textsuperscript{ NC } = f\textsuperscript{ NC } + XNOR;
      fix\_positive\_unate(f\textsuperscript{ ON }, f\textsuperscript{ NC }, x\textsubscript{i});
      fix\_positive\_unate(f\textsuperscript{ ON }, f\textsuperscript{ NC }, not\_x\textsubscript{i});
}
```

The domain transformation is shown in Fig. 5. Comparison between read-once and read-polarity-once functions.
lines 5-6 in Fig. 8), as these lines represent impossible input conditions. The next step is to guarantee that both variables \((x_i, not \, x_i)\) become positive unate after the domain transformation.

The fix_positive_unate method is based on the definition of unateness shown in (5). Let \(x_i\) be the variable to be fixed, \(Y = \{0, 1, X\}\) and \(\{tp_i, tn_i\} \in Y\) be a term in the line \(i\) of the truth table when \(f(x_i = 1)\) and \(f(x_i = 0)\), respectively. The following two states need to be fixed:

\[
\begin{align*}
tp_i = 0 \text{ and } tn_i = X & \quad (18) \\
.tp_i = X \text{ and } tn_i = 1 & \quad (19)
\end{align*}
\]

In (18), if \(tn_i\) receives the logic value 1 the function becomes binate. The same happens in (19) if \(tp_i\) receives the value 0. In order to avoid both situations, the method fix_positive_unate (see lines 13 to 24 in Fig. 8) properly assigns values to the don’t care terms that are responsible for these cases.

Empirical results have shown that the unatization runtime is irrelevant in the entire flow to synthesize RPO functions. The runtime of the ISF2RO algorithm is currently the main bottleneck.

C. ISF2RPO: An algorithm to factorize RPO functions

After presenting the ISF2RO algorithm and the unatization process, we are able to describe the entire flow of the RPO factoring algorithm. The complete algorithm proceeds in two main steps. The first step reads an ISF \(f\) and computes the polarity of the variables. Every binate variable is split into two separate positive unate variables according to the unatization process. The second step performs the search (ISF2RO) for an RO expression for the ISF resulting from the domain transformations. If ISF2RO returns an RO output, the expression is then rewritten considering the original variables as they were presented before the domain transformation. The flowchart of the proposed algorithm is presented in Fig. 9.

The time complexity of the RPO algorithm is bounded by the complexity of the ISF2RO algorithm. Experimental results have demonstrated that the RPO algorithm can efficiently find optimal solutions for functions with up to 16 literals.

V. EXPERIMENTAL RESULTS

This section presents an investigation of the occurrence of RPO functions over distinct sets of functions. The first analysis was performed over the set of functions with up to 5 variables grouped under NPN-equivalence [19], where NPN stands for the operations of input negation (first N), input permutation (P) and output negation (second N). We refer to this set of functions as NPN-groups. The second experiment analyzes the occurrence of RPO functions over the ISCAS’85 benchmark circuits [14]. The last experiment was carried out for a set of Boolean functions that are important in the context of logic brick design [20]. The platform used to obtain the results was a Linux system on Intel Core i5 2400 processor with 2GB main memory.

A. Occurrence of RPO functions over 5-inputs NPN-group

The first experiment was carried out over the set of all 5-input Boolean functions, reduced under NPN-equivalence into a set of 616,125 representative functions (denoted as NPN-Group). To run the algorithm for all these functions, the execution time was 4 minutes. The worst case runtime for a single function was 800 ms, while the average case was less than 1 ms.

For the universe of 5-input NPN-group functions, there are 1,462 functions that are classified as RPO, while only 21 functions are classified as RO. This means that there are approximately 70 times more RPO functions compared to RO functions of up to 5-inputs. Our results have demonstrated that the universe of RPO is broader than the universe of RO functions, for which many works have been devoted [10-12][17][18].

Comparative results evaluating the efficiency of the proposed algorithm are shown in Table V, considering the set of 1,462 RPO functions in the 5-input NPN-Group. Our algorithm presented better results in terms of number of literals than Quick Factor (QF) [21], Good Factor (GF) [21], ABC [22] and X-Factor [3] [4] tools. The proposed algorithm gives better results, when compared to previously published approaches, as it finds the exact solution for RPO functions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Literals</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF [21]</td>
<td>16,086</td>
<td>1.9s</td>
</tr>
<tr>
<td>GF [21]</td>
<td>15,671</td>
<td>2.3s</td>
</tr>
<tr>
<td>ABC [22]</td>
<td>15,981</td>
<td>2.0s</td>
</tr>
<tr>
<td>X-Factor *</td>
<td>13,253</td>
<td>7.1s</td>
</tr>
<tr>
<td>RPO (this paper)</td>
<td>13,064</td>
<td>5.7s</td>
</tr>
</tbody>
</table>

* Results of an in-house implementation of the X-Factor algorithm.
B. Occurrence of RPO functions over ISCAS’85 benchmark

We have also performed an investigation of the occurrence of RPO functions over ISCAS’85 benchmark circuits. Such analysis has been carried out in order to figure out the frequency of RPO functions in comparison to RO functions in mapped circuits. We have extracted functions with up to 8 inputs from the benchmarks circuits by using k-cuts [23]. These functions were grouped by equivalence under input permutations (P) [19], and the resulting groups are denoted as P-groups. In Table VI, the term Occurrences represents the number of functions before grouping it into P-groups of equivalence.

We have selected the functions in two ways. Table VI summarizes the results regarding all possible functions with up to 8 inputs in the circuits. In Table VII, the functions represent a circuit cover selected by performing an AIG greedy covering algorithm [23]. Similarly to RO functions, the number of RPO functions decreases as the number of variables increases. This is an expected result, as Boolean functions with more inputs tend to be more complex.

In Table VIII, it is possible to verify the runtime for synthesizing all functions with k-cuts up to k = 8. Table IX shows the runtime of the algorithm to synthesize the functions extracted from a circuit cover. The results show the efficiency of the proposed ISF2RPO algorithm.

Experiments carried out over ISCAS’85 benchmark circuits have demonstrated that RPO functions are significantly more frequent than RO functions.

C. Occurrence of RPO functions over functions for logic brick design

The last experiment was carried out over a benchmark of important functions for logic brick design. According to [20], a set of logic functions can be added to a cell library to significantly improve specific designs. In one of the examples given by Motiani et al., a set of 12 distinct functions were added to a library. Out of the 12 functions added, 6 were RO, 10 were RPO (including the 6 RO that are also RPO) and only 2 are not RPO functions. These functions are presented in Table X. This observation highlights the importance of the RPO class for different technologies, including the logic brick methodology proposed by Montiani et al.

### Table VI. Analysis of functions from ISCAS’85 benchmark circuits.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>RO</th>
<th>P-Groups</th>
<th>Occurrences</th>
<th>RPO</th>
<th>P-Groups</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>67%</td>
<td>84%</td>
<td>100%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>53%</td>
<td>66%</td>
<td>90%</td>
<td>88%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>44%</td>
<td>54%</td>
<td>85%</td>
<td>71%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>37%</td>
<td>42%</td>
<td>69%</td>
<td>53%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>33%</td>
<td>36%</td>
<td>57%</td>
<td>46%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>34%</td>
<td>36%</td>
<td>52%</td>
<td>47%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>32%</td>
<td>34%</td>
<td>46%</td>
<td>44%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table VII. Functions selected from ISCAS’85 benchmark circuits using a greedy covering algorithm [23].

<table>
<thead>
<tr>
<th>Inputs</th>
<th>RO</th>
<th>P-Groups</th>
<th>Occurrences</th>
<th>RPO</th>
<th>P-Groups</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>78%</td>
<td>93%</td>
<td>100%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>59%</td>
<td>63%</td>
<td>87%</td>
<td>94%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>49%</td>
<td>53%</td>
<td>78%</td>
<td>81%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>35%</td>
<td>36%</td>
<td>79%</td>
<td>81%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>41%</td>
<td>39%</td>
<td>63%</td>
<td>60%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>45%</td>
<td>43%</td>
<td>62%</td>
<td>57%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>14%</td>
<td>15%</td>
<td>27%</td>
<td>27%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table VIII. Runtime for synthesizing all k-cut functions (k = 8).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P-Groups</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>680</td>
<td>123.4</td>
<td>0.18</td>
</tr>
<tr>
<td>C17</td>
<td>12</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>1,224</td>
<td>133.1</td>
<td>0.11</td>
</tr>
<tr>
<td>C2670</td>
<td>6,345</td>
<td>284.6</td>
<td>0.04</td>
</tr>
<tr>
<td>C3540</td>
<td>9,275</td>
<td>123.9</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>844</td>
<td>3.2</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>432</td>
<td>98.1</td>
<td>0.23</td>
</tr>
<tr>
<td>C5315</td>
<td>11,350</td>
<td>806.4</td>
<td>0.07</td>
</tr>
<tr>
<td>C6288</td>
<td>142</td>
<td>0.6</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>17,888</td>
<td>8045.1</td>
<td>0.45</td>
</tr>
<tr>
<td>C880</td>
<td>1,691</td>
<td>86.3</td>
<td>0.05</td>
</tr>
</tbody>
</table>

### Table IX. Total runtime for synthesizing functions selected by a greedy covering algorithm [23].

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P-Groups</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>16</td>
<td>3.70</td>
<td>0.23</td>
</tr>
<tr>
<td>C17</td>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>44</td>
<td>2.37</td>
<td>0.05</td>
</tr>
<tr>
<td>C2670</td>
<td>68</td>
<td>39.53</td>
<td>0.58</td>
</tr>
<tr>
<td>C3540</td>
<td>129</td>
<td>1.29</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>25</td>
<td>0.15</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>10</td>
<td>0.89</td>
<td>0.09</td>
</tr>
<tr>
<td>C5315</td>
<td>108</td>
<td>58.07</td>
<td>0.54</td>
</tr>
<tr>
<td>C6288</td>
<td>38</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>130</td>
<td>8.42</td>
<td>0.06</td>
</tr>
<tr>
<td>C880</td>
<td>36</td>
<td>18.08</td>
<td>0.50</td>
</tr>
</tbody>
</table>
VI. CONCLUSIONS

This paper discussed in depth the recently proposed concept of read-polarity once (RPO) functions [13]. The major contributions of this work are: (1) an algorithm for factoring incompletely specified functions into read-once (RO) equations; (2) a domain transformation that splits existing binate variables into two independent unate variables; and (3) a complete algorithm for exact factoring of RPO functions.

The proposed algorithm was implemented and compared to existing factoring algorithms, showing that it guarantees minimal factored forms for the class of RPO functions. Out of the set of 616,125 NPN-grouped functions with up to 5-inputs, 1,462 functions were identified as RPO, while only 21 functions are RO. Moreover, experimental results taking into account ISCAS’85 benchmark circuits have shown that RPO functions are quite more frequent in circuits than RO functions. Furthermore, the RPO class of functions is also important for different technologies, including the logic brick methodology proposed by Motiani et al [20], as demonstrated by the large number of RPO functions (10 out of 12) given as example in [20]. The entire flow to factorize RPO functions has been validated, and our implementation has been able to find optimal solutions of functions with up to 8 binate variables in a reasonable runtime.

Both the ISF2RO and the ISF2RPO algorithms were implemented and integrated in the SwitchCraft [24] framework. SwitchCraft provides a set of tools for switch network and logic gate generation.

ACKNOWLEDGEMENTS

Research partially funded by Nangate Inc. under a Nangate/UFRGS research agreement, by CAPES and CNPq Brazilian funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community’s Seventh Framework Programme under grant 248538 – Synaptic.

REFERENCES


