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

Vinicius Callegaro[1], Mayler G. A. Martins[2], Renato P. Ribas[1,2] and André I. Reis[1,2]

[1]PPGC, Institute of Informatics, UFRGS
[2]PGMICRO, 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 ISFs, 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 and negative) of a variable appears at most once in the factored expression. For instance, RO algorithms fail when factoring the expression $f=!a \cdot b \cdot d+b \cdot c+a \cdot c$, since the variable '$a$' is binate. The proposed RPO algorithm can factorize such a function into an exact expression $f=(!a \cdot d+c) \cdot (a+b)$, which presents only 5 literals. Moreover, we have investigated the occurrence of RPO functions in circuit designs taking into account the ISCAS'85 benchmark circuits [14]. The results have shown that RPO functions are significantly more frequent than RO functions. The entire flow comprising the unatization of RPO functions and the factoring of ISF in RO expressions has been validated. Our implementation was able to find optimal solutions of functions with up to 16 literals.

This paper is organized as follows. Section II presents the basic concepts and a brief overview of the current state-of-the-art algorithms for factoring RO functions. In Section III, we propose an algorithm for factoring ISF in RO expressions. Section IV presents the proposed domain transformation, *i.e.* the unatization process, as well as the complete algorithm to perform the factoring of ISF in RPO expressions. Experimental results are shown in Section V, whereas the conclusions are outlined in Section VI.

II. PRELIMINARIES

A. Boolean functions

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

$$f: B^n \rightarrow \Upsilon \qquad (1)$$

where $x = [x_1,\dots,x_n] \in B^n$ 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* ($\boldsymbol{X}$) values, in addition to the binary values 0 and 1, *i.e.* $\Upsilon = \{0,1,\boldsymbol{X}\}$ [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 $f \approx g$ *iff* $f^{ON} \cap g^{OFF} = \varnothing$ and $f^{OFF} \cap g^{ON} = \varnothing$.

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 $f(x_1,x_2,\dots,x_n)$, the cofactor of f with respect to x_i is denoted as $f(x_1,\dots,x_i=c,\dots,x_n)$, 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 $f(x_1,\dots,x_i=c,\dots,x_n)$ 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:

$$\alpha = f(x_i=1) \qquad (2)$$

$$\beta = f(x_i=0) \qquad (3)$$

$$\gamma = \alpha + \beta \qquad (4)$$

$$\textit{positive unate:} \quad (\alpha \equiv \gamma) \wedge (\alpha \neq \beta) \qquad (5)$$

$$\textit{negative unate:} \quad (\beta \equiv \gamma) \wedge (\alpha \neq \beta) \qquad (6)$$

$$\textit{don't care:} \quad \alpha \equiv \beta \qquad (7)$$

$$\textit{binate:} \quad \alpha \neq \beta \neq \gamma \neq \alpha \qquad (8)$$

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:

$$f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4 + x_1 \cdot x_3 \cdot x_5 \qquad (9)$$

is a RO function since it can be factored into:

$$f = x_1 \cdot (x_2 + x_3 \cdot (x_4 + x_5)) \qquad (10)$$

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 $f= x_1 \cdot x_2+ x_1 \cdot x_3+ x_2 \cdot x_3$ has $f= x_1 \cdot (x_2+x_3) +x_2 \cdot x_3$ as the minimal solution, in which variables x_2 and x_3 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 $f(x_1,x_2,\dots,x_n)$ a Boolean function with n-input variables, two variables x_i and x_j can be compressed whether their cofactors are equals, *i.e.*, $f(x_i=c) \equiv f(x_j=c)$. 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 φ. The algorithm then looks for compression groups in φ until reaching (if it is possible) an RO form. The time complexity of the JPHI algorithm is $O(n^2K)$, 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.

	Time complexity	Fail fast	Partial RO	Works with ISF
IROF	$O(n\|f\|)$	Yes	No	No
JPHI	$O(n^2K)$	No	Yes	Yes*

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

Table II. Possible *don't care* assignments of Fig. 1 and solutions.

Assignment	ISOP	RO form
00	$x1{\cdot}x4+x2{\cdot}x3$	$x1{\cdot}x4+x2{\cdot}x3$
01	$x1{\cdot}x4+x2{\cdot}x3+x2{\cdot}x4$	**Non-RO**
10	$x1{\cdot}x4+x1{\cdot}x3+x2{\cdot}x3$	**Non-RO**
11	$x1{\cdot}x4+x1{\cdot}x3+x2{\cdot}x3+x2{\cdot}x4$	$(x1 + x2) \cdot (x3 + x4)$

$x1$	$x2$	$x3$	$x4$	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

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:

LogicArrangement
String expression
Set<Assignment> on_spec
Set<Assignment> off_spec
ISF pos_cube_cof
ISF neg_cube_cof

Figure 2. LogicArrangement data structure.

$$la_i.expression = \text{“}x_i\text{”} \tag{11}$$

$$la_i.on\_spec = \{\ (x_i = 1)\ \} \tag{12}$$

$$la_i.off\_spec = \{\ (x_i = 0)\ \} \tag{13}$$

$$la_i.pos\_cube\_cof = f(la_i.on\_spec) \tag{14}$$

$$la_i.neg\_cube\_cof = f(la_i.off\_spec) \tag{15}$$

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=11101X1011X00000. 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_3 and la_4. 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_1 and la_4; la_2 and la_3. 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

```
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 (f, f3.on_spec);
18          f3.neg_cube_cof = cubeCof (f, 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.

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

LA	exp	On	Off	pos_cube_cof	neg_cube_cof
la_1	x_1	x_1	$!x_1$	11101X1011101X10	11X0000011X00000
la_2	x_2	x_2	$!x_2$	1110111011X011X0	1X101X1000000000
la_3	x_3	x_3	$!x_3$	11111X1X11110000	10101010X0X00000
la_4	x_4	x_4	$!x_4$	1111111111XX0000	1100XX0011000000

Table IV. Grouped logic arrangements after first iteration.

LA	exp.	On	Off	pos_cube_cof	neg_cube_cof
la_5	x_1+x_2	x_1	$!x_1 !x_2$	11101X1011101X10	0000000000000000
la_6	x_3+x_4	x_3	$!x_3 !x_4$	1111111111XX0000	0000000000000000
la_7	$x_1 \cdot x_4$	$x_1 x_4$	$!x_1$	1111111111111111	1100XX0011000000
la_8	$x_2 \cdot x_3$	$x_2 x_3$	$!x_2$	1111111111111111	10101010X0X00000

possible to see the results of first iteration of the proposed algorithm.

The algorithm continues grouping logic arrangements whenever it is possible. After the second iteration, two new logic arrangements can be found with the following expressions:

$$(x1 \cdot x4) + (x2 \cdot x3) \tag{16}$$

$$(x1 + x2) \cdot (x3 + x4) \tag{17}$$

Since both (16) and (17) are RO functions, the test in the line (26) in Fig. 3 finds and returns both solutions.

B. ISF2RO empirical analysis

The worst case time complexity of ISF2RO algorithm is $O(2^{2^n})$, where n is the number of variables in f. However, our empirical results are encouraging. The worst runtime observed in the experiments with 16-inputs functions was 500 seconds. The runtime is still lower than *Exact Factor* [8], even being able to deal with higher number of input variables. The *Exact Factor* algorithm takes 600 seconds to factorize an equation with 12 literals, while our ISF2RO algorithm finds minimal equations for functions with 16 literals in less than 500 seconds.

In order to show the time complexity of the proposed algorithm, ISCAS'85 benchmark circuits were synthesized. The functions obtained from the benchmark (72,423 in total) were grouped regarding input count, from 3 up to 16 input variables. The worst case runtime for each group is depicted in Fig. 4. The vertical axis is presented in logarithm scale. This way, it is easy to observe that the algorithm has an exponential time complexity. However, for this benchmark of functions, the worst case runtime was 20 seconds to synthesize a function with 16 inputs. The experiment demonstrates the efficiency of the proposed method.

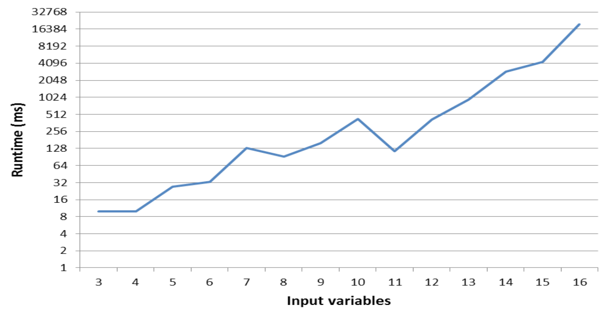


Figure 4. The worst case runtime (in ms) to synthesize functions from *ISCAS'85* benchmark circuits grouped by input count.

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 $f=a \cdot (b+c \cdot (d+e))$ is both RO and RPO, while $f=(a+b) \cdot (!a+!b)$ 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.

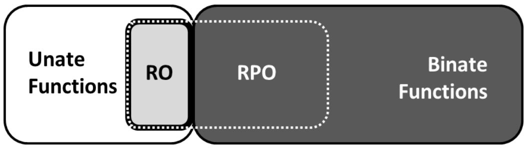


Figure 5. Comparison between *read-once* and *read-polarity-once* functions.

B. Unatization process

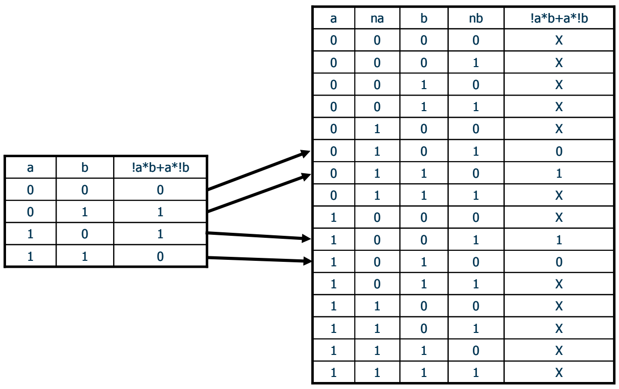
According to Definition 3, if a function can be factored into an RPO expression, each polarity (positive or negative) of a variable appears at maximum once in the factored expression. Hence, an interesting point of investigation is if it is possible to separate the positive and negative literals, and 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 $f=(a+b)\cdot(!a+!b)$. In order to *unatizate* *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 in all of its variables, a new function is obtained. The computation of the cofactors and the new function obtained is presented in Fig. 7. Notice that out of 12 original *don't care* values (shown in Fig. 6), two of them remain unspecified after this process. This means that every possible assignment of the *don't care* (X) values will lead safely to unate functions. This property is further exploited by the *ISF2RO* algorithm, which returns more than one solution if it is possible.

The pseudo-algorithm for the unatization process is shown in Fig. 8. The basic idea is to split the binate variables into independent unate variables. Let x_i be a binate variable of *f*. In order to unatizate x_i, a variable *not*_x_i is inserted into *f*, as shown in the line 4 in Fig. 8. It is important to notice that both variables x_i and *not*_x_i cannot have the same value at the same time. When a Boolean constant *c* is assigned to input x_i, the complemented value has to be assigned to input *not*_x_i. In this sense, the lines where both variables are assigned to the same constant are set to *don´t care* (see



a	b	!a*b+a*!b
0	0	0
0	1	1
1	0	1
1	1	0

a	na	b	nb	!a*b+a*!b
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	X
0	1	0	0	X
0	1	0	1	0
0	1	1	0	1
0	1	1	1	X
1	0	0	0	X
1	0	0	1	1
1	0	1	0	0
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Figure 6. Expanding 2-input exclusive-OR to 4 variables.

a	na	b	nb	!a*b+a*!b	f(a=1)	f(a=0)	f(na=1)	f(na=0)	f(b=1)	f(b=0)	f(nb=1)	f(nb=0)
0	0	0	0	X=0	X	X	x	x	0	x	0	0
0	0	0	1	X=0	1	X	0	x	x	0	0	0
0	0	1	0	X=0	0	X	1	0	0	x	x	0
0	0	1	1	X	X	X	x	x	x	0	x	0
0	1	0	0	X=0	X	X	x	x	1	x	0	x
0	1	0	1	0	X	0	0	x	x	0	0	x
0	1	1	0	1	X	1	1	0	1	x	x	1
0	1	1	1	X=1	X	X	x	x	x	0	x	1
1	0	0	0	X=0	X	X	x	x	0	x	1	0
1	0	0	1	1	1	X	x	1	x	1	1	0
1	0	1	0	0	0	X	1	0	0	x	1	0
1	0	1	1	X=1	X	X	x	x	x	1	1	0
1	1	0	0	X	X	X	x	x	1	x	1	x
1	1	0	1	X=1	X	0	x	1	x	1	1	x
1	1	1	0	X=1	X	1	1	0	1	x	1	1
1	1	1	1	X=1	X	X	x	x	x	1	1	1

Figure 7. *Don't care* terms are set to force the function to become positive unate.

```
1   unatization($f^{ON}$,$f^{DC}$) {
2     for (int i = 0; i < n; i++)
3       if (is_binate($x_i$)){
4         createVariable(not_$x_i$);
5         XNOR = !( $x_i \oplus$ not_$x_i$);
6         $f^{DC} = f^{DC}$ + XNOR;
7         fix_positive_unate($f^{ON}$,$f^{DC}$, $x_i$);
8         fix_positive_unate($f^{ON}$,$f^{DC}$, not_$x_i$);
9       }
10    }
11  }
12
13  fix_positive_unate($f^{ON}$,$f^{DC}$ $x_i$) {
14    PD = positive_cofactor($f^{DC}$ $x_i$);
15    ND = negative_cofactor($f^{DC}$ $x_i$);
16    PC = positive_cofactor($f^{ON}$ $x_i$);
17    NC = negative_cofactor($f^{ON}$ $x_i$);
18    state_0x = !PD · !PC · ND;
19    $f^{DC} = f^{DC}$ · !state_0x;
20    $f^{ON} = f^{ON}$ · !state_0x;
21    state_x1 = PD · !ND · NC;
22    $f^{DC} = f^{DC}$ · !state_x1;
23    $f^{ON} = f^{ON}$ + state_x1;
24  }
```

Figure 8. Pseudo-algorithm for the unatization process.

lines 5-6 in Fig. 8), as these lines represent impossible input conditions. The next step is to guarantee that both variables (x_i and not_x_i) became 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, $\Upsilon=\{0,1,X\}$ and $\{tp_i,tn_i\} \in \Upsilon$ 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:

$$tp_i = 0 \text{ and } tn_i = \mathrm{X} \quad (18)$$

$$tp_i = X \text{ and } tn_i = 1 \quad (19)$$

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 assign 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.

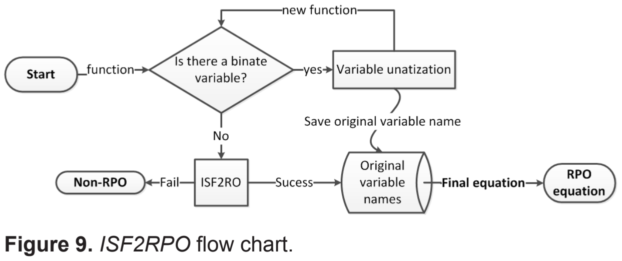


Figure 9. *ISF2RPO* flow chart.

V. EXPERIMENTAL RESULTS

This section presents an investigation of the occurrence of RPO functions over distinct set of functions. The first analysis was performed over the set of functions with up to 5 variables grouped under NPN-equivalency [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 V. Total runtime and number of literals obtained to factor 1,462 RPO functions using different approaches.

	QF [21]	GF [21]	ABC [22]	X-Factor[a] [3][4]	RPO (this paper)
Literals	16,086	15,671	15,981	13,253	13,064
Runtime	1.9s	2.3s	2.0s	7.1s	5.7s

[a] 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.

	RO		RPO	
Inputs	*P-Groups*	*Occurrences*	*P-Groups*	*Occurrences*
2	67%	84%	100%	100%
3	53%	66%	90%	88%
4	44%	54%	85%	71%
5	37%	42%	69%	53%
6	33%	36%	57%	46%
7	34%	36%	52%	47%
8	32%	34%	46%	44%

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

	RO		RPO	
Inputs	*P-Groups*	*Occurrences*	*P-Groups*	*Occurrences*
2	78%	93%	100%	100%
3	59%	63%	87%	94%
4	49%	53%	78%	81%
5	35%	36%	79%	81%
6	41%	39%	63%	60%
7	45%	43%	62%	57%
8	14%	15%	27%	27%

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

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

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

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

Table X. Set of 12 distinct functions given by Motiani *et al* [20] where 10 were RPO functions.

Original	Read-polarity-once form
p0p1!p3+p2!p3+p4p5	((((p1 p0) + p2) !p3) + (p5 p4))
p0p1!p3+!p3!p4+p1p2!p3	((((p2 + p0) p1) + !p4) !p3)
p1p3p6+p0!p2p5+p3p4p6+!p1!p2!p4	(!p4 !p1 + p0 p5) !p2 +p6 p3 (p4 + p1)
!p1!p2p3+!p0!p1p3+p1!p3+p0p2!p3	((((p2 p0) + p1) !p3) + ((!p1 p3) (!p0 + !p2)))
p1!p4+!p0!p3+p1!p3+!p2!p4+!p0!p4+!p2!p3	(((!p0 + p1) + !p2) (!p3 + !p4))
p0p2+p1p2+p3p4	(((p1 + p0) p2) + (p4 p3))
!p1!p2+p4p5+!p6!p7+p0p3	(((!p1 !p2) + (p3 p0)) + ((!p6 !p7) + (p4 p5)))
p1p3p4+p0p2p3p4+!p1!p2!p3p4+!p0!p1!p3p4+!p1!p2p3!p4+p1!p3!p4+p0!p1p2!p4	**Non-RPO**
p0!p3!p5+!p0!p1p2+p2p3p4+p1!p4!p5+p1!p3!p5+p0!p4!p5	(p4 p3 + !p1 !p0) p2 + (!p3 + !p4) !p5 (p0 + p1)
!p0!p1p2+!p0p1!p3+p0p1p2+p0!p1!p3	**Non-RPO**
!p1!p2!p3+p0!p3+!p1!p2!p4+p0!p4	(((!p1 !p2) + p0) (!p4 + !p3))
!p0!p1p4+!p2p4+p0p2!p3+p1p2!p3	((((p1 + p0) p2) + p4) ((!p1 !p0) + (!p3 + !p2)))

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

[1] R. K. Brayton, "Factoring logic functions," *IBM Journal of Research and Development,* vol. 31, no. 2, Mar 1987, pp. 187-98.

[2] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms,* Springer, 2006, p564.

[3] M. C. Golumbic and A. Mintz, "Factoring logic functions using graph partitioning," in *Proceedings of Int'l Conf. on Computer-Aided Design* (ICCAD). 1999, pp. 195-199.

[4] A. Mintz and M. C. Golumbic, "Factoring Boolean functions using graph partitioning," *Discrete Applied Mathematics*, vol. 149, no. 1–3, 2005, pp. 131-53.

[5] T. Stanion and C. Sechen, "Boolean division and factorization using binary decision diagrams," IEEE *Transactions on CAD,* vol. 13, no. 9, Sep. 1994, pp. 1179-84.

[6] E. L. Lawler, "An approach to multilevel Boolean minimization", *Journal of the ACM*, vol.11, no. 3, July 1964. pp. 283-95.

[7] H. Yoshida, M. Ikeda and K. Asada, "Exact minimum logic factoring via quantified Boolean satisfiability," in *Proceedings of IEEE Int'l Conf. on Electronics, Circuits and Systems* (ICECS), Dec. 2006, pp. 1065-68.

[8] H. Yoshida and M. Fujita, "Exact minimum factoring of incompletely specified logic functions via quantified Boolean satisfiability," *IPSJ Trans. on System LSI Design Methodology*, vol. 4, Feb. 2011, pp. 70-79.

[9] M. G. A. Martins, L. S. Rosa Jr, A. B. Rasmussen, R. P. Ribas and A. I. Reis, "Boolean factoring with multi-objective goals," in *Proceedings of IEEE Int'l Conf. on Computer Design* (ICCD), 2010, pp. 229-234.

[10] M. C. Golumbic, A. Mintz and U. Rotics, "Factoring and recognition of read-once functions using cographs and normality," in *Proceedings of Design Automation Conference* (DAC), 2001, pp. 109-14.

[11] M. C. Golumbic, A. Mintz and U. Rotics, "An improvement on the complexity of factoring read-once Boolean functions," *Discrete Applied Mathematics*, vol. 156, no. 10, May 2008, pp. 1633-1636.

[12] T. Lee and C. Wang, "Recognition of fanout-free functions," in *Proceedings of Asia and South Pacific Design Automation Conference* (ASP-DAC), Jan. 2007, pp. 426-31.

[13] V. Callegaro, M.G.A. Martins, R.P. Ribas and A.I. Reis, "Read-polarity-once Boolean functions," in *Proceedings of Integrated Circuits and Systems Design* (SBCCI), 2013, pp. 1-6.

[14] IWLS 2005 Benchmarks: http://iwls.org.

[15] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen and G. D. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA. 1984.

[16] J. P. Hayes, "The fanout structure of switching functions," *Journal of the ACM*, vol. 22, no. 4, Oct. 1975, pp. 551-71.

[17] J. Peer and R. Pinter, "Minimal decomposition of Boolean functions using non-repeating literal trees," in *Proceedings of IFIP,* 1995.

[18] V. Gurvich, "Criteria for repetition-freeness of functions in the algebra of logic," *Soviet Math. Dokl,* vol. 43, no. 3, 1991, pp. 721–726.

[19] V. P. Correia and A. I. Reis, "Classifying n-Input Boolean Functions," in *Proceedings of IBERCHIP*, 2001, pp. 58-66.

[20] D. Motiani, V. Kheterpal and L. T. Pileggi, "Method for the definition of a library of application-domain-specific logic cells," *United States Patent 7784013*, 2010.

[21] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. L. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," *Tech. Rep. UCB/ERL M92/41*, UC Berkeley, Berkeley, 1992.

[22] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 051205, [Online] http://www.eecs.berkeley.edu/~alanmi/abc/.

[23] O. Martinello Jr., F. S Marques, R. P. Ribas and A. I. Reis, "KL-cuts: a new approach for logic synthesis targeting multiple output blocks," in *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE), 2010, pp. 777-82.

[24] V. Callegaro, F. S. Marques, C. E. Klock, L. S. da Rosa Jr., R. P. Ribas and A. I. Reis, "SwitchCraft: a framework for transistor network design," in *Proceedings of the 23rd Symposium on Integrated Circuits and System Design* (SBCCI), 2010, pp. 49-53.