A Comparative Analysis of Different Boolean Function Synthesis Methods

Gabriel Ammes, Walter Lau and Renato P. Ribas
Institute of Informatics
UFRGS
Porto Alegre, Brazil
(gabriel.ammes, wlneto, rpribas)@inf.ufrgs.br

***Abstract*—This paper presents a useful evaluation of Boolean function synthesis methods considering metrics like number of literals, number of operations, circuit logic depth and execution time. Several logic function synthesis approaches have been proposed, but the comparison between them is not clearly defined. Such analysis becomes quite useful for the decision of which one is more suitable for a given application.**

***Index Terms*—Boolean function, logic synthesis, CAD tool, logic optimization, digital circuit.**

I. Introduction

Logic synthesis is a well-established research field, with a direct impact on the quality of digital circuit design [1]. Logic synthesis methods exploit properties of the Boolean function to get a minimized expression concerning a given metric and profile. In particular, the logic function synthesis (*i.e.*, single-output digital circuit) is the basis for more complex, multiple-output, combinational circuit design.

In this way, there are several methods with different synthesis strategies, leading to different solutions. A Boolean function can be synthesized into two-level sum-of-products (SOP) or product-of-sums (POS) structure, as well as into multi-level factored expression. Moreover, these representations of function behavior can be restricted to AND and OR operations as well as can take into account other ones like exclusive-OR (XOR) operation. Each function synthesis approach presents interesting features but also important drawbacks. The choice of which one should be performed is strongly related to the technology and circuit topology targeted. Furthermore, in general, logic synthesis methods present high computing cost.

In this paper, we present a useful evaluation of some widely applied logic function synthesis methods, taken into account metrics such as: (i) number of literals, (ii) circuit logic depth, and (iii) execution time. All methods evaluated here have been developed in the same computer programming platform in order to obtain a fair comparison regarding computation cost.

The rest of this paper is organized as follow. Section II presents the approaches considered in this study, being divided into methods based on SOP and POS structures, the Quine-McCluskey algorithm together to the extension including XOR logic operation, and the presentation of classes of XOR expressions. Section III presents the experimental results and the comparative analysis of the presented methods. The conclusions are outlined in Section IV.

II. Boolean Function Synthesis

Many methods of Boolean Synthesis uses an n-variable truth table as input, as shown in Fig. 1a. Minterms and maxterms can represent a truth table. A minterm is a product of all function variables, where each variable appears once in positive or negative polarity. A maxterm corresponds to a sum of all functions variables, where each variable also appears once and can be in either polarities. The maxterm can be obtained by negating the a minterm and vice versa.

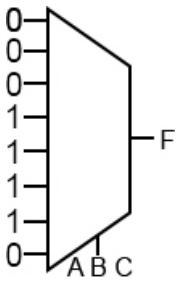
Even though we present a range of different methods, there still others not explored in this work. The approaches discussed in the following represent some of the most applied ones in the logic synthesis of digital circuits and have been implemented in the scope of this work.

A. POS/SOP two-level expression

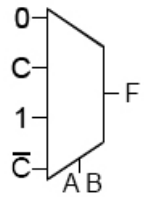
The most straightforward representation of a truth table in a Boolean expression is through its canonical normal form. It can be done into two different ways: at first, the disjunctive canonical normal form, that consists in a sum of literal products, where each product of literals is called a cube; secondly, the conjunctive canonical normal form, which consists in a product of sums of literals. Each sum of literals is known as a clause. Due to the structural characteristics of

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a) Truth Table



(b) MUX 8X1



(c) MUX 4X1

Fig. 1: Representation of the function F = $A \oplus (C \cdot B)$

these forms, the first one is called sum-of-products (SOP) and the second one is called products-of-sums (POS).

Whereas the SOP is composed by the minterms of a truth table (where the function is 1) connected through the OR operator, the POS consists in the truth table maxterms (where the function is 0) connected by the AND operator. For the truth table presented in Fig. 1a, the SOP and POS expressions are the following:

$$F = A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C \quad (1)$$

$$F = A + B + C \cdot A + B + \overline{C} \cdot A + \overline{B} + C \cdot \overline{A} + \overline{B} + \overline{C} \quad (2)$$

B. MUX-based expression

Multiplexer (MUX) is a combinational block that selects one of the input signals to a single-output terminal [2]. In other words, the MUX output corresponds to one of the inputs according to the configuration of the selection signals. Notice that for a MUX with 2^n inputs, there must be n selection signals. A MUX behavior can be logically represented by an SOP where each cube corresponds to the selection condition and the respective input information. Also, a MUX can describe a given truth table. To do so, the MUX inputs are the truth table outputs, while the MUX selector signals are the function variables. Therefore, for a MUX with four inputs, it is possible to represent all 2-variable functions. The implementation of the truth table presented in Fig. 1a in a MUX 8x1 is shown in Fig. 1b and its logic function corresponds to equation 1.

The previous MUX-based representation is quite straightforward but does not implement any minimization. It is possible to implement an n-variable function by using a MUX with 2^{n-1} input signals and $n-1$ selectors. In order to achieve such an optimization, $n-1$ variables are used as selector signals, and the input signal can be then assigned to input variables in any polarity, as well as the constant values 0 and 1. This optimization is shown in Fig. 1c and its logic function expression is the following:

$$F = \overline{A} \cdot B \cdot C + A \cdot \overline{B} + A \cdot B \cdot \overline{C} \quad (3)$$

C. Quine-McCluskey method

The Quine-McCluskey algorithm is one of the most known logic synthesis methods [3]. This algorithm is used to find an SOP with the minimum number of cubes for a given n-variable truth table. It can also be used to find a POS representation with the minimum number of clauses.

The algorithm is divided into two parts: finding all prime implicants of a function and then using these prime implicants to cover the minterms of the truth table. A prime implicant is a cube that can cover or represent one or more minterms and is not covered by any other cube. Moreover, a prime implicant is called an essential prime implicant when a minterm is only covered by such a cube.

The algorithm starts by enumerating the cubes. Then, two cubes that have at most one literal with different polarity can be combined. This combination leads to a new cube without such a different literal, which covers the original cubes. When two cubes are combined, the original ones are removed from the possible solutions. This process of combination is repeated until there are no more left cubes to combine. In the end, the method returns a set of cubes, which are the prime implicants of the function.

In next, the coverage step takes place. The coverage consists of fixing the essential prime implicants and using the others prime implicants to covering the remaining minterms. The output of this algorithm is the cubes that result in coverage with the less number of prime implicants. The result is not unique because it is possible to exist more than one minimum coverage. For the truth table used until now, a possible expression generated by this algorithm is the following:

$$F = A \cdot \overline{B} + \overline{A} \cdot B \cdot C + A \cdot \overline{C} \quad (4)$$

As already stated, it is also possible to synthesize an expression through a POS form. To do so, instead of representing minterms as cubes, the evaluation of variables where the output is 0 is considered. The algorithm flow keeps the same, and the only modification is that the SOP output has to be negated. In the end, the well-known DeMorgan laws are applied over the negated SOP, resulting into a POS expression. For the same truth table, a possible SOP is as follows:

$$F = (A + B) \cdot (\overline{A} + \overline{B} + \overline{C}) \cdot (A + C) \quad (5)$$

A feature that can be implemented in the Quine-McCluskey algorithm is the minimum decision chain (MDC) [4]. The MDC function property corresponds to the minimum number of transistors in the stack needed to implement a given function, which represents a strong correlation with the delay. This number of transistors in the stack can be considered as the number of literals in a cube.

It takes into account the number of literals in a cube as the MDC of this cube and defines which one can be used in this coverage. To define the MDC of the function, the cubes that are not essential prime implicants and have the largest products are removed from the possible solution until finding an essential prime implicant. The product size of this essential prime implicant corresponds to the MDC of the target function. This modification is interesting because there are cases where the original Quine-McCluskey algorithm finds a minimum coverage but exist another solution (coverage), possibly that is not the minimum, that respect the function MDC. A comparison between the solution of the original algorithm and the solution of the modified algorithm can be done through the following expressions:

$$F= (\overline{A} \cdot \overline{B} \cdot \overline{D}) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{D} \cdot \overline{E}) + (A \cdot C \cdot D) + (B \cdot C \cdot \overline{D} \cdot E) \quad (6)$$

$$F= (\overline{A} \cdot \overline{B} \cdot \overline{D}) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{D} \cdot \overline{E}) + (A \cdot C \cdot D) + (\overline{A} \cdot \overline{D} \cdot E) + (\overline{A} \cdot \overline{B} \cdot \overline{D}) \quad (7)$$

An expansion of the Quine-McCluskey algorithm is presented in [5]. This expansion consider that a cube can use exclusive-OR (XOR) operation, besides the usually used AND and OR operations. This expansion is interesting because the use of XOR operation is very common in digital circuits design, and its use in Boolean synthesis methods can improve the solutions [6].

The difference between the original implementation and such an XOR-based expansion is in the cubes combination. Besides the combinations that occur when just one literal has different polarity in two cubes, cubes are also combined when two literals have different polarities. In the case that these literals have different polarities in each cube, the new cube will contain an XOR operation of these two literals. Moreover, a cube has a pair of literals with positive polarity, and the other has the same literals with negative polarity, the new cube will contain an XNOR of these two literals. A result of this algorithm for the truth table already presented is the following:

$$F = C \cdot (B \oplus A) + (A \cdot \overline{C}) \quad (8)$$

D. XOR Expression

As mentioned before, the use of XOR operation can improve the results of Boolean synthesis. Sasao, in [7], presents many classes of expressions that use XOR logic gates, and five of them are being shown here. Each class has specific features and can be used to synthesize an expression. Therefore, expansions are applied over an original expression to build a new expression that can be a reduced expression. This process can generate a better representation than the original expression.

The expressions that can be applied are the positive Davio expansion, presented in Equation 9, the negative Davio expansion, presented in Equation 10, and the Shannon expansion, presented in Equation 11,

$$F = 1 \cdot f_0 \oplus x_1 \cdot f_2 \quad (9)$$

$$F = \overline{x_1} \cdot f_2 \oplus 1 \cdot f_1 \quad (10)$$

$$F = \overline{x_1} \cdot f_0 \oplus x_1 \cdot f_1 \quad (11)$$

Where $F = f(x_1, x_2, x_3, ..., x_n)$, $f_0 = f(0, x_2, x_3, ..., x_n)$, $f_1 = f(1, x_2, x_3, ..., x_n)$and $f_2 = f_0 \oplus f_1$.

The expression classes presented in this work are defined in the next:

1) Positive Polarity Reed-Muller Expression: It is defined when the positive Davio expansion can be applied to each variable. The result expression is unique for a given function and consists of positive polarity literal only. The representation of the Equation 1 is the following:

$$F = A \oplus (B \cdot (A \oplus A)) \oplus (C \cdot (A \oplus (B \cdot (A \oplus A)) \oplus A \oplus B)) \quad (12)$$

2) Fixed Polarity Reed-Muller Expression: Either the positive Davio expansion or the negative Davio expansion can be applied to each variable. Thus, it generates 2^n different expressions for a given function, and its expressions consist of literals with either positive or negative polarity. A representation of the Equation 1 is as follows:

$$F = A \oplus (C \cdot (A \oplus (A \oplus B))) \quad (13)$$

3) Kronecker Expression: Either the positive Davio expansion, the negative Davio expansion or the Shannon expansion can be applied to each variable. There are at most 3^n different expressions for a function, and its literals can appear with any polarity. A representation of the Equation 1 is the following:

$$F = (\overline{C} \cdot A) \oplus (C \cdot (A \oplus B)) \quad (14)$$

4) Pseudo Reed-Muller Expression: When an expansion is applied in expression, an XOR operation with two sub-expressions is generated. In this class, either positive Davio expansion or negative Davio expansion can be applied to each variable and for each sub-expression. Therefore, it is generated a more general expression than the fixed polarity Reed-Muller expression. There are at most 2^{2^n-1} different expressions for a function and its expressions consist of literal with either positive or negative polarity. A representation of the Equation 1 is as follows:

$$F = A \oplus (C \cdot B) \quad (15)$$

5) Pseudo Kronecker Expression: Either positive Davio expansion, negative Davio expansion or Shannon expansion can be applied for each variable and each sub-expression. Therefore, it generates a more general expression than the Kronecker expression. There are at most 3^{2^n-1} different expressions for a function, and its literals can appear with any polarity. A representation of the Equation 1 is the following:

$$F = A \oplus (C \cdot B) \quad (16)$$

III. Implementation and Experimental Results

The methods presented in this work were implemented in Java programming language and validated using the logic synthesis framework KARMA [8]. These methods were evaluated considering the number of literals, number of AND and OR operations, number of XOR operations, circuit logic depth and execution time. Table I shows the experimental results

for all 3-variable functions, Table II presents the results for all 4-variable functions and Table III provides the results for a thousand random generated 5-variable functions. Pseudo-Kronecker can be applied in functions up to 3 variables and pseudo-Reed-Muller in functions up to 4 variables.

TABLE I: All functions of 3 variables.

Method	Literals	AND/OR	XOR	Depth	Time(s)
SOP	12.0	11.0	0.0	2.0	0.01
POS	12.0	11.0	0.0	2.0	0.01
MUX	6.7	5.7	0.0	1.9	0.02
QMSOP	5.0	4.0	0.0	1.8	0.03
QMPOS	5.0	4.0	0.0	1.8	0.03
QMXOR	3.8	2.2	0.6	2.0	0.08
PPRME	6.0	2.4	2.6	3.3	0.09
FPRME	4.9	2.5	1.8	3.0	0.23
KRO	4.5	2.5	1.0	2.0	0.32
PSDRME	4.0	1.8	1.2	1.7	0.32
PSDKRO	4.0	1.8	1.1	1.7	2.80

TABLE II: All functions of 4 variables.

Method	Literals	AND/OR	XOR	Depth	Time(s)
SOP	32	31	0	2	4.1
POS	32	31	0	2	4.2
MUX	18.7	17.8	0	2	16.9
QMSOP	12.7	11.5	0	2	20.3
QMPOS	12.7	11.5	0	2	20.0
QMXOR	8.2	5.45	1.82	2.8	83.6
PPRME	18.8	8	9.7	5.8	12.7
FPRME	15	6.8	7.2	5.75	73.1
KRO	11.5	6.9	3.6	5	277.2
PSDRME	10.6	6.2	3.4	2.3	25200.0

TABLE III: 5 variables random functions.

Method	Literals	AND/OR	XOR	Depth	Time(s)
SOP	59.6	58.6	0	2	0.4
POS	101.3	100.3	0	2	0.5
MUX	38.3	37.3	0	2	2.0
QMSOP	26.5	25.5	0	2	9224.0
QMPOS	26.7	25.7	0	2	23233.8
QMXOR	15.5	11.4	3.1	2.8	10931.0
PPRME	50.3	23.0	26.2	7.7	1.2
FPRME	38.4	18.9	18.5	7.5	5.48
KRO	22.1	14.1	7.0	6.8	29.0

Taking a look at the results, the first two tables one can see that the results between SOP and POS and of Quine-McCluskey using SOP and POS has symmetric values, what does not appear in the third one. It happens because was not possible to use all 5-variables functions, using random generates functions. In this way, one can see that POS has much more literals than SOP and Quine-McCluskey using SOP and POS do not present this difference, what explain the time difference between this two Quine-McCluskey approaches. Also, one can see that the MUX-based method presents better results than SOP and POS based approaches, as expected, and has good results compared to the Quine-McCluskey algorithm if one takes into account the time difference between these approaches. We have also noticed that the effort of applying an expansion in order to minimize a given expression does not look to worth. Even using a multi-level synthesis, the results are worse than the XOR-extended Quine-McCluskey approach that can be considered a 3-level synthesis.

The time increase presented in the comparison between Table II and Table III in the Quine-McCluskey-based methods are bigger than expected. This is due to the coverage step implementation, which is a NP-hard problem [9]. Further optimization in the covering may be exploited in order to speed up the runtime.

IV. Conclusions

Comparing the results presented in Section III, we can conclude that, for a general purpose and disregarding the design cost differences between XOR and AND/OR gates, the Quine-McCluskey expansion considering XOR seems to be the most indicated Boolean synthesis method. As future work, we intend to optimize the Quine-McCluskey algorithm or implement another 2-level method and to optimize the pseudo-Kronecker expression by implementing the method presented in [7], as well as to implement another multi-level synthesis method, such as the factoring one [10].

V. Acknowledgements

This work has been partially supported by CNPq and CAPES Brazilian government agencies.

References

[1] L. T. Wang, Y. W. Chang, K. T. Cheng, Electronic Design Automation: Synthesis, Verification and Test. Morgan Kaufmann, 2009.

[2] Brown, S., Vranesic, Z. "Synthesis of Logic Functions Using Multiplexers", Fundamentals of Digital Logic With Vhdl Design, pp. 323-326, 2000.

[3] E. J. McCluskey, "Minimization of Boolean functions", Bell Syst. Tech. J., vol. 35, no. 5, pp. 1417-1444, 1956.

[4] M. G. A. Martins, V. Callegaro , R. P. Ribas, and A. I. Reis, "Computing Minimum Decision Chains of Boolean Functions " 26th South Symposium on Microelectronics, pp. 35-38, 2011.

[5] B. C. H. Turton, "Extending Quine-McCluskey for Exclusive-Or Logic Synthesis"; IEEE Transactions on Education, Vol. 39, No. 1, Feb. 1996, pp. 81-85.

[6] Sasao, T. "Logic Synthesis with EXOR gates", Logic Synthesis and Optimization, pp 259-285, 1993.

[7] Sasao, T. "AND-EXOR Expressions and their Optimization", Logic Synthesis and Optimization, pp 287-312, 1993.

[8] C. Klock, F. Schneider, M. Gomes, D. Moura, R. Ribas, and A. Reis, "KARMA: A didactic tool for two-level logic synthesis." Microelectronic Systems Education (MSE), IEEE International Conference on, jun. 2007, pp. 59–60

[9] Kumar V. "Finite Boolean Algebra", Discrete Mathematics, pp 150-154, 2002.

[10] Mintz A., Golumbic M. C. "Factoring Boolean functions using graph partitioning", Discrete Applied Mathematics, Vol. 149, pp. 131-153, 2005.