An Algorithm for Detecting XOR-Type Logic

Elena Dubrova

Department of Electronics
Royal Institute of Technology
Stockholm, Sweden
elena@ele.kth.se

Tomas Bengtsson

School of Engineering
Jönköping University
Jönköping, Sweden
beto@ing.hj.se

Abstract

In this paper we formulate a sufficient condition for Boolean functions to have a decomposition of type $f = (g \oplus h) + r$, with the total number of product-terms in g, h and r smaller than the number of product-terms in f. This condition is used to develop an algorithm for deciding whether a given function is likely to benefit from XOR minimization.

1 Introduction

In many cases, functions with embedded XOR logic have very large expressions in terms of AND, OR and NOT operations. For example, an n-variable parity function needs 2^{n-1} product-terms consisting of n literals each if expressed using AND, OR and NOT. Contrary, it needs only n single-literal product-terms if expressed using AND, XOR and NOT. Vice versa, functions which do not have embedded XOR logic might have larger representation if expressed using AND, XOR and NOT. Since both, AND-OR and AND-XOR, minimization algorithms are quite time-consuming for large functions, it would be attractive to know a priori which of them to use for a given function. We address this problem in this paper. We study what kind of structure a function should have to benefit from XOR minimization, and prove a theorem characterizing one such structure. This theorem formulates a sufficient condition for a given function $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$, to have a decomposition of type $f(X) = (g(X) \oplus h(X)) + r(X)$ with the total number of product-terms in g, h and r smaller than the number of product-terms in f. It is interesting to note that we put no restrictions on the support sets of g and h, i.e. they can be equal, overlapping or disjoint. This differs our method from the existing methods for algebraic and Boolean decomposition. For example, the algebraic division method [1] requests the intersection of the support sets of g and h to be disjoint. The generalized algebraic division method [2] requests the support sets of g and h to have at least one disjoint variable. In the classical Boolean decomposition theory [3], [4], the case when g and h have the same support set is classified as *trivial* non-disjoint decomposition, and is omitted from consideration. Since our approach is more general, it has a potential of finding decomposition for functions which cannot be optimized by algebraic and Boolean algorithms. For example, the function shown in Figure 1 does not pose any non-trivial Boolean disjoint or non-disjoint decomposition. However, it has a trivial Boolean decomposition of type $f(X) = g(X) \oplus h(X)$, namely:

$x_3x_4 \backslash x_1x_2$	00	01	11	10
00	0	1	0	1
01	1	1	1	0
11	0	1	0	1
10	1	0	1	1

Figure 1. An example function.

$$f(x_1, x_2, x_3, x_3) = (\overline{x}_1 x_2 + \overline{x}_3 x_4) \oplus (x_1 \overline{x}_2 + x_3 \overline{x}_4)$$

allowing to reduce the number of product-terms twice compared to the product-terms in the minimal sum-of-product-terms form of this function.

One possible application of the theoretical result of this paper is detecting XOR-type logic. We present a heuristic which use our sufficient condition to quickly estimate whether a given function is likely to benefit from XOR-minimization or not. Note, that the purpose of the heuristic is *not* to find an optimal XOR cover, but rather to quickly decide whether a XOR cover with a number of product-terms smaller than the one in two-level AND-OR form exists.

The paper is organized as follows. Section 2 describes

the basic notation and definitions used in the sequel. Section 3 presents the theorem formulating the sufficient condition. Section 4 describes the heuristic for detecting XOR-logic. Section 5 shows the experimental results. In the final section, some conclusions are drawn and directions for further research are proposed.

2. Notation

Let $f(x_1, x_2, \ldots, x_n)$ be an incompletely specified Boolean function of type $f : \{0,1\}^n \rightarrow \{0,1,-\}$, of the variables $x_1, \ldots, x_n$, where "$-$" denotes a don't care value. We use F_f, R_f and D_f to denote on-set, off-set and don't-care-set of a function f, respectively.

A *product-term* is a Boolean product (AND) of one or more variables $x_1, \ldots, x_n$ or their complements. A convenient representation for a product-term is *cube* [6]. We use the terms *cube* and *product-term* interchangeably.

The *size* of a set A, denoted by $|A|$, is the number of cubes in it. The *complement* of a set A, denoted by $\overline{A}$, is the intersection of the complements for each cube of A. The *intersection* of two sets A and B, denoted by $A \cap B$, is the union of the pairwise intersection of the cubes from A and B. The *union* of two sets A and B, denoted by $A \cup B$, is the union of the cubes from A and B.

A *supercube* of two cubes c_1 and c_2, denoted by $sup(c_1, c_2)$, is the smallest cube containing both c_1 and c_2.

3. Sufficient condition

In this section we examine what kind of structure a function should have to benefit from XOR minimization, and prove a theorem characterizing one such structure.

To optimize a Boolean expression according to some optimization criteria, one normally looks for some property reducing the "cost" of the expression. For example, the number of product-terms in the two-level AND-OR (sum-of-product-terms) expression can be reduced by applying the rule $x \cdot Y + \overline{x} \cdot Y = Y$, where x is a variable and Y is a product-term.

We would like to formulate a rule which could be applied to a two-level AND-OR expression to transform it to an expression of type $f = (g \oplus h) + r$, with the total number of product-terms in g, h and r smaller than the number of product-terms in f. First, we prove the following Lemma.

Lemma 1 *Let $a_1, a_2, \ldots, a_k$, $k > 1$, be cubes from the on-set F_f of a Boolean function $f : \{0,1\}^n \rightarrow \{0,1,-\}$ and let the intersection of $sup(a_1, \ldots, a_k)$ with the off-set R_f be a non-empty set of cubes $sup(a_1, \ldots, a_k) \cap R_f = \cup_{i=1}^{p} c_i$, for some $p \geq 1$. If for each cube c_i we can find a cube $b_i \in F_f$ such that $sup(b_i, c_i) \cap R_f = c_i$ as well as $sup(a_1, \ldots, a_k) \cap sup(b_i, c_i) = c_i$, then*

$$sup(a_1, \ldots, a_k) \oplus \bigcup_{i=1}^{p} sup(b_i, c_i) = \\ = \bigcup_{j=1}^{k} a_j \cup \bigcup_{i=1}^{p} (sup(b_i, c_i) - c_i) \quad (1)$$

Proof: To prove the Lemma, we first show that the following two properties hold. Let X, Y and Z be sets of cubes.

Property 1 *If $X \cap Y = \emptyset$, then $(X \cup Y) \oplus Y = X$.*

Proof: $(X \cup Y) \oplus Y = ((X \cup Y)' \cap Y) \cup ((X \cup Y) \cap Y') = ((X' \cap Y' \cap Y) \cup ((X \cap Y') \cup (Y \cap Y')) = X \cap Y'$. Clearly, $X \cap Y' = X$ since $X \cap Y = \emptyset$. $\square$

Property 2 *If $X \cap Z = \emptyset$, $Y \cap Z = \emptyset$ and $Y \subset X$, then $(X \oplus Y) \cup Z = X \oplus (Y \cup Z)$.*

Proof: On one hand, $(X \oplus Y) \cup Z = (X \cap Y') \cup (X' \cap Y) \cup Z = (X \cap Y') \cup Z$, since $X' \cap Y = \emptyset$ for $Y \subset X$. On the other hand, $X \oplus (Y \cup Z) = (X' \cap (Y \cup Z)) \cup (X \cap (Y \cup Z)') = (X' \cap Y) \cup (X' \cap Z) \cup (X \cap Y' \cap Z') = (X \cap Y') \cup Z$, since $X' \cap Y = \emptyset$ for $Y \subset X$, $X' \cap Z = Z$ for $X \cap Z = \emptyset$, similarly $X \cap Z' = X$ for $X \cap Z = \emptyset$ an thus $X \cap Y' \cap Z' = X \cap Y'$. $\square$

Since $a_i \in F_f, \forall i \in \{1, 2, \ldots, k\}$, and $c_j \in R_f, \forall j \in \{1, \ldots, p\}$, the intersection of the sets $\cup_{i=1}^{k} a_i$ and $\cup_{j=1}^{p} c_j$ is empty. Therefore, by applying Property 1, we can write:

$$\bigcup_{j=1}^{k} a_j = (\bigcup_{j=1}^{k} a_j \cup \bigcup_{i=1}^{p} c_i) \oplus \bigcup_{i=1}^{p} c_i \\ = sup(a_1, \ldots, a_k) \oplus \bigcup_{i=1}^{p} c_i$$

Making union with $\cup_{i=1}^{p} (sup(b_i, c_i) - c_i)$ on both sides, we get:

$$\bigcup_{j=1}^{k} a_j \cup \bigcup_{i=1}^{p} (sup(b_i, c_i) - c_i) = \\ = (sup(a_1, \ldots, a_k) \oplus \bigcup_{i=1}^{p} c_i) \cup \bigcup_{i=1}^{p} (sup(b_i, c_i) - c_i)).$$

Since $sup(a_1, \ldots, a_k) \cap \cup_{i=1}^{p} (sup(b_i, c_i) - c_i) = \emptyset$, $\cup_{i=1}^{p} (sup(b_i, c_i) - c_i) \cup \cup_{i=1}^{p} c_i = \emptyset$ and $\cup_{i=1}^{p} c_i \subset sup(a_1, \ldots, a_k)$, we can to apply Property 2 and get:

$$\bigcup_{j=1}^{k} a_j \cup \bigcup_{i=1}^{p} (sup(b_i, c_i) - c_i) = \\ = sup(a_1, \ldots, a_k) \oplus (\bigcup_{i=1}^{p} c_i \cup \bigcup_{i=1}^{p} (sup(b_i, c_i) - c_i)) \\ = sup(a_1, \ldots, a_k) \oplus \bigcup_{i=1}^{p} sup(b_i, c_i).$$

□

Lemma 1 gives us a condition for substituting a subset F_f^* of the on-set F_f of a function f by two functions g and h of type $g, h : \{0,1\}^n \rightarrow \{0,1\}$ so that $F_f^* = F_g \oplus F_h$ and the total number of cubes in F_g and F_h is smaller that in F_f^*. Next, we prove that this condition is *sufficient*, meaning that, whenever it holds, the number of cubes in f can be reduced by representing it as $f = (g \oplus h) + r$.

Theorem 1 (Sufficient condition) *If a Boolean function f satisfies Lemma 1 for some set of cubes $\{a_1, a_2, \ldots, a_k\}$, $a_i \in F_f$, $\forall i \in \{1,2,\ldots,k\}$, then it can be represented as $f = (g \oplus h) + r$ with the total number of cubes in g, h, r smaller than in f.*

Proof: Suppose a function $f = (F_f, D_F, R_f)$ satisfies Lemma 1. Then, there exist some cubes $a_i, b_j \in F_f$, $i \in \{1,2,\ldots,k\}, j \in \{1,\ldots,p\}$, and $c_j \in R_f$ satisfying eq.(1). All the cubes from the set $\cup_{i=1}^{p}(sup(b_i, c_i) - c_i)$ belong to either on-set F_f or to don't care set D_f. Also, for each i, the set $sup(b_i, c_i) - c_i$ includes at least one cube from the on-set F_f, namely b_i. So, $p+1$ cubes from the left hand side of the equation (1) cover at least $p + k$, $k > 1$, cubes from the on-set F_f, given by the right hand side of (1). If we set $F_g = sup(a_1, \ldots, a_k)$ and $F_h = \cup_{i=1}^{p} sup(b_i, c_i)$, then $|F_g| + |F_h| < |F_f^*|$, with $F_f^* = (\cup_{j=1}^{k} a_j) \cup (\cup_{i=1}^{p}(sup(b_i, c_i) - c_i))$. Defining the "reminder" as $F_r = F_f - F_f^*$, we get a decomposition of f of type $f = (g \oplus h) + r$ with the total number of cubes in g, h, r smaller than in f.

□

As an example, consider a 4-variable function f shown in Figure 2. Let $a_1 = 0000$ and $a_2 = 0101$. Then we have $sup(a_1, a_2) = 0-0-$ and $sup(a_1, a_2) \cap R_f = \{0100, 0001\} = \{c_1, c_2\}$. It is easy to see that $b_1 = 1100$ satisfies $sup(b_1, c_1) \cap R_f = c_1$ and $b_2 = 0011$ satisfies $sup(b_2, c_2) \cap R_f = c_2$. Since $sup(b_1, c_1) - c_1 = b_1$ as well as $sup(b_2, c_2) - c_2 = b_2$, we get

$$sup(a_1, a_2) \oplus (sup(b_1, c_1) \cup sup(b_1, c_1)) = a_1 \cup a_2 \cup b_1 \cup b_2$$

Thus, f can be represented as $f = g \oplus h$, with $g = \overline{x}_1\overline{x}_3$ and $h = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1\overline{x}_2x_4$.

$x_3x_4 \backslash x_1x_2$	00	01	11	10
00	1	0	1	0
01	0	1	0	0
11	1	0	0	0
10	0	0	0	0

Figure 2. An example function.

Check_XOR(F_f, D_f, R_f)
input: on-set F_f, don't care set D_f and off-set R_f of f
output: "YES" if function is likely to benefit from XOR-minimization, "NO" otherwise.

```
counter = 0;
for (every pair of cubes (a1, a2) ∈ Ff × Ff) {
    flag_lemma_satisfied = 1;
    Compute supercube sup(a1, a2);
    Compute sup(a1, a2) ∩ Rf = ∪_{i=1}^{p} ci
    for (each cube ci ∈ ∪_{i=1}^{p}) {
        flag_found_b = 0;
        for (each cube bj ∈ Ff) {
            Compute supercube sup(bj, ci);
            if(sup(bj, ci) ∩ Rf = ci) {
                if(sup(a1, a2) ∩ sup(bj, ci) = ci) {
                    flag_found_b = 1; /* found bj for this ci */
                    break; /* break for-loop */
                }
            }
        }
        if(flag_found_b = 0) { /* failed to find bj for some ci */
                flag_lemma_satisfied = 0;
                break;
        }
    }
    if(flag_lemma_satisfied = 1) /* Lemma is satisfied for (a1, a2) */
        counter++;
}
if(δ > 0)
    return(YES, counter);
else
    return(NO);
```

Figure 3. Pseudocode of the algorithm.

4. Detecting XOR-type logic

One possible application of Theorem 1 is detecting XOR-type logic. The larger is the subset of the on-set of a function f, satisfying Lemma 1, the more f can benefit from XOR minimization. However, there might be several different choices of such a subset. Computing a best one would require first trying all possible subsets $a_1, a_2, \ldots, a_k$ of f to find the ones satisfying Lemma 1, and then solving the covering problem to find which combination of the subsets results in the best XOR-cover for f. Both steps would require exponential time, and therefore the exact algorithm would be too slow for large functions. Instead, we developed a simple heuristic, which gives quickly estimates whether a given function is likely to benefit from XOR-minimization or not.

The pseudocode of our heuristic is shown in Figure 3. The input is the on-set F_f, don't care-set D_f and off-set R_f of f. The output is "YES" if function is likely to benefit from XOR-minimization, "NO" otherwise. The algorithm repeats the following basic steps:

1. Choose a pair of cubes, a_1 and a_2, and compute their

supercube $sup(a_1, a_2)$;

2. Compute the intersection $sup(a_1, a_2) \cap R_f = \cup_{i=1}^{p} c_i$;
3. For each cube c_i, try to find a cube $b_j \in F_f$ such that the supercube $sup(b_j, c_i)$ satisfy conditions $(sup(b_j, c_i) \cap R_f = c_i)$ and $(sup(a_1, a_2) \cap sup(b_j, c_i) = c_i)$ of Lemma 1.
4. Repeat 1, 2 and 3 for all pairs of a_1 and a_2, updating the number of pairs satisfying Lemma 1 (*counter*) after each iteration.
5. If some of the pairs satisfy Lemma 1, return "YES" together with the number of pairs. Otherwise, return "NO".

The main saving in time comes from checking Lemma 1 for only pairs of cubes (a_1, a_2) instead of all possible subsets $(a_1, a_2, \ldots, a_k)$, $n \geq k > 1$. This reduces the number of choices to try from $O(2^{|F_f|})$ to $O(|F_f|^2)$. Another essential saving in time is due to the fact that we are not computing the XOR cover for f at all. Instead, we simply count the number of pairs (a_1, a_2) satisfying Lemma 1. This doesn't cause any problem, because our purpose in not *to find* the minimal XOR cover, but rather *to decide* whether such a cover is likely to be smaller than the OR cover. Since the condition given by Lemma 1 is sufficient, any time we found a pair (a_1, a_2) satisfying Lemma 1, we know that f can be represented as $f = (g \oplus h) + r$ with the total number of cubes in g, h, r smaller than in f. The more pairs satisfy Lemma, the more flexibility we have to select a good XOR cover out of them. However, since Theorem 1 proves only the sufficiency of the condition, not its necessity, there might be cases when the condition is *not* satisfied, but the number of cubes in f can still be reduced by representing it as $f = (g \oplus h) + r$. So, in general, our heuristic might be too pessimistic. To check how often this is the case we have conducted a set of experiment, which are described in the next section.

5. Experimental results

We performed a set of experiments, targeting to determine how pessimistic is our heuristic, i.e. how often it gives the answer "NO" when the right answers is "YES". Table 1 summarizes the results. Columns 2 and 3 give the number of inputs n and the number of outputs m of the function. Column 4 refers to the number F_f of cubes in the cover computed by Espresso [5]. Column 5 gives the total number, N, of pairs (a_1, a_2), satisfying Lemma 1.

The condition given by Lemma 1 is sufficient, so we would expect the functions which satisfy with large N to benefit from XOR minimization. This seems to be confirmed by the experiments. Functions with large N, like *9sym, alu4, b9, ex7, life, rd53, rd73, rd83, sym10, xor5*, are known to have a smaller XOR cover compared to OR cover [7]-[11]. However, since the condition is not sufficient, there is a case, *t481*, when it is not satisfied, but the function can substantially benefit from XOR minimization. *t481* can be represented as $f(X) = g(X) \oplus h(X)$ with only 16 products in g and h in total [7]. We are currently studying the possibilities to relax Lemma 1.

6. Conclusion

In this paper we have formulated a sufficient condition for a function f to have decomposition of type $f = (g \oplus h) + r$, with the total number of product-terms in g, h and r smaller than the number of product-terms in f. Using this condition, we have designed an algorithm for deciding whether a function is likely to benefit from XOR-minimization.

Our current research includes designing an algorithm with the algorithm for finding a good decomposition of type $f = (g \oplus h) + r$, based on Lemma 1. We plan to use the heuristic developed in this paper as a preprocessing step, deciding whether it worth to continue the XOR-minimization or not. Once a decomposition of type $f = (g_1 \oplus h_1) + r_1$ is found, there are several choices: (1) decompose g_1 or h_1, resulting is an iterative decomposition of type $f = (((g_2 \oplus h_2) + r_2) \oplus h_1) + r_1$; (2) decompose r_1, resulting is a multiple decomposition of type $f = (g_1 \oplus h_1) + (g_2 \oplus h_2) + r_2$; (3) or decompose both g_1 or h_1 and r_1, resulting in a general tree-like decomposition.

We also looking into the ways to relax the Lemma 1 and, if possible, to formulate a necessary condition.

References

[1] R.K. Brayton, C. McMullen, "The Decomposition and factorization of Boolean Functions", *Proc. ISCAS-82*, 1982, pp. 49-54.

[2] T. Stanion, C. Sechen, "Quasi-algebraic decomposition of switching functions", *Proc. Int. Conf. Advanced Research in VLSI*, 1995, pp. 358-367.

[3] R. L. Ashenhurst, "The decomposition of switching functions", *Proc. International Symp. Theory of Switching* Part I **29**, 1959, pp. 74-116.

[4] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, Van Nostrand, Princeton, 1962.

[5] R.K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publisher, 1984.

[6] Y. H. Su, P. T. Cheung, "Computer minimization of multi-valued switching functions", *IEEE Trans. Comput.*, vol. C-21, 1972, pp. 995-1003.

[7] E. V. Dubrova, D. M. Miller, J. C. Muzio, "AOXMIN-MV: A Heuristic Algorithm for AND-OR-XOR Minimization", *Proc. 4th International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design*, Victoria, B.C., Canada, August 20-21, 1999, pp. 37-53.

[8] T. Sasao, "A design method for AND-OR-EXOR three-level networks", *Notes of Int. Workshop on Logic Synthesis*, May 1995, pp. 8:11-8:20.

[9] D. Debnath, T. Sasao, "A heuristic algorithm to design AND-OR-EXOR three-level networks", *Proc. Asia and South Pacific Design Automation Conf.*, 1998.

[10] S. Chattopadhyay, S. Roy, P. P. Chaudhuri, KGP-MIN: An efficient multilevel multioutput AND-OR-XOR minimizer, *IEEE Trans. on CAD of ICs and Systems*, **16** No. 3 (1997), 257-265.

[11] A. Jabir, J. Saul, "A Heuristic decomposition Algorithm for AND-OR-EXOR Thre-level Minimization of Boolean Functions", *Proc. 4th International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design*, Victoria, B.C., Canada, August 20-21, 1999, pp. 55-74.

Table 1. Number of pairs satisfying Lemma 1.

Example	n	m	$\lvert F_f \rvert$	N
5xp1	7	10	65	27
9sym	9	1	86	301
alu2	10	8	68	1
alu3	10	8	66	0
alu4	14	8	575	114
amd	14	24	66	7
b2	16	17	106	2
b9	16	5	119	111
b10	15	11	100	3
b12	15	9	43	2
bc0	26	11	179	41
bw	5	28	22	2
clip	9	5	120	28
con1	7	2	9	1
cordic	23	2	155	96
dist	8	5	123	60
duke2	22	29	86	9
ex7	16	5	119	111
exam	10	10	67	0
ex1010	10	10	284	1
f51m	8	8	77	44
gary	15	11	107	11
in0	15	11	107	11
in1	16	17	62	2
in2	19	10	136	11
in5	24	14	62	0
inc	7	9	30	3
life	9	1	84	724
misex1	8	7	12	0
misex2	25	18	28	0
misex3	14	14	690	161
misex3c	14	14	197	86
mlp4	8	8	128	52
newapla2	6	7	7	0
newbyte	5	8	8	0
newcpla1	9	16	38	0
newtpla	15	5	23	0
p1	8	18	55	1
radd	8	5	75	73
rd53	5	3	31	198
rd73	7	3	127	1304
rd84	8	4	255	7590
root	8	5	57	15
ryy6	16	1	112	0
sao2	10	4	58	53
shift	19	16	100	0
squar5	5	8	25	6
sqn	7	3	38	3
sym10	10	1	210	1918
t1	21	23	102	0
t2	17	16	53	5
t481	16	1	481	0
table3	14	14	175	3
table5	14	14	175	3
tial	14	8	581	120
ts10	22	16	128	0
vg2	25	8	110	0
x1dn	27	6	110	0
x9dn	27	7	120	0
xor5	5	1	16	120
z4	7	4	59	54
Z5xp1	7	10	65	23