A Sufficient Condition for Detecting AND-OR-AND-Type Logic

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

Andrew J. Sullivan
IBM EDA group
Fishkill, N.Y.
USA
sullia@us.ibm.com

Abstract

Three-level logic is shown to have a potential for reduction of the area over two-level implementations, as well as for a gain in speed over multi-level implementations. Algorithms for finding AND-OR-XOR and AND-OR-AND expressions were developed, however an open problem remained which of the algorithms should be used to find an optimal solution for a given function. In this paper we formulate a sufficient condition for a function to have a decomposition of type $f = g \cdot h + r$, with the total number of products in g, h and r smaller or equal than the number of products in f. This condition is used to design an algorithm for deciding whether a function is likely to have a compact AND-OR-XOR expression.

1. Introduction

Three-level logic is shown to be a good trade-off between the speed of two-level logic and the density of multi-level logic. Three-level logic can be implemented by a Programmable Logic Device (PLD) whose simplified logic block consists of two Programmable Logic Arrays (PLAs), implementing the first two levels of logic, and a set of two-input gates, called *logic expanders*, implementing the third level. Each logic expander can be programmed to realize any function of two variables. Such a PLD implements logic expressions of the type:

$$f(x_1, \ldots, x_n) = (P_1 + \ldots + P_k) \circ (P_{k+1} + \ldots + P_r) \quad (1)$$

where P_i, $i \in \{1, \ldots, r\}$ denotes an arbitrary product-term involving some of the variables $x_1, \ldots, x_n$ or their complements, "$\circ$" denotes a binary operation, and $1 \leq k \leq r$.

The first algorithm, addressing the optimization of such PLDs, was presented in 1991 by Malik, Harrison and Brayton in [1]. It first determines a minimal expression (1) for the case of "$\circ$" = AND, and then applies output phase optimization to the logic expander to check suitability of the other choices of "$\circ$". Such a scheme minimizes (1) for all interesting cases except "$\circ$" = XOR, "$\circ$" = XNOR. A modified version of the algorithm [1], aiming to reduce its runtime, was presented in [2]. Several algorithm addressing "$\circ$" = XOR case were designed in [3]-[6].

In many cases, functions which have compact AND-OR-XOR expressions blow up when represented as AND-OR-AND, and vice versa. Since AND-OR-AND algorithms and AND-OR-XOR 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 AND-OR-AND optimization, 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 $f(X) = g(X) \cdot h(X)$ with the total number of products in g and h smaller or equal than the number of products in f. If only a subset of the on-set of f satisfies the condition, then f is represented as $f(X) = g(X) \cdot h(X) + r(X)$. The smaller the "reminder" r, the more likely f to have a compact AND-OR-AND expression. The algorithm we present in this paper estimates how big is r by subsequently enlarging g and h until it is no longer possible. Note, that the purpose of the algorithm is *not* to find an optimal decomposition $f = g \cdot h + r$, but rather to quickly decide whether an AND-OR-AND form with a number of products smaller than the one in two-level AND-OR form exists. It is also interesting to observe 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 [7] requests the intersection of the support sets of g and h to be disjoint. The generalized algebraic division method [8] requests the support sets of g and h to have at least one disjoint variable. In the classi-

cal Boolean decomposition theory [9], [10], the case when g and h have the same support set of is classified as *trivial* non-disjoint decomposition, and is omitted from consideration. Since our algorithm is more general, it has a potential of finding decomposition for functions which cannot be optimized by algebraic and Boolean algorithms.

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 algorithm for checking the condition. 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*. 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 AND-OR-AND optimization, 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-products) 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 a three-level AND-OR-AND expression with a smaller number of product-terms. Unfortunately, it is very hard (if possible) to formulate a property reducing the number of product-terms. We were only able to formulate a condition which guarantees non increasing of them. Furthermore, this condition is sufficient, but not necessary in general.

The theorem formulated below shows how we can substitute a subset F_f^* of the on-set F_f of a function $f : \{0,1\}^n \rightarrow \{0,1,-\}$ by two functions g and h of type $g, h : \{0,1\}^n \rightarrow \{0,1\}$ so that $F_f^* \subseteq F_g \cdot F_h$, $F_g \cap F_h \cap R_f = \emptyset$ and the total number of cubes in F_g and F_h is no greater that in F_f^*.

Theorem 1 *Let $F_f^* = \{c_0, c_1, \ldots, c_{k-1}\}$ be a subset of the one-set F_f of f, with k being an even integer greater than 2. If for all $i \in \{0, 2, 4, \ldots, k-2\}$ and $j \in \{1, 3, 5, \ldots, k-1\}$ it is holds that*

$$sup(c_i, c_{i+1}) \cap sup(c_j, c_{j+1}) \subset (F_f \cup D_f) \qquad (2)$$

where $+$ is the addition modulo k, then $F_f^* \subseteq F_g \cap F_h$ and $F_g \cap F_h \cap R_f = \emptyset$, where

$$\begin{aligned} F_g &= \bigcup_{m=0}^{k/2-1} sup(c_{2m}, c_{2m+1}) \\ F_h &= \bigcup_{m=0}^{k/2-1} sup(c_{2m-1}, c_{2m}) \end{aligned} \qquad (3)$$

with "$+$" and "$-$" being the addition and substitution modulo k, respectively.

Proof: First we show that $F_f^* \subseteq F_g \cap F_h$:

$$\begin{aligned} & F_g \cap F_h = \\ & = \bigcup_{m_1=1}^{k/2-1} sup(c_{2m_1}, c_{2m_1+1}) \cap \bigcup_{m_2=0}^{k/2-1} sup(c_{2m_2-1}, c_{2m_2}) \\ & \{\text{from eq.(3)}\} \\ & = \bigcup_{m_1=0}^{k/2-1} \left(sup(c_{2m_1}, c_{2m_1+1}) \cap \bigcup_{m_2=0}^{k/2-1} sup(c_{2m_2-1}, c_{2m_2}) \right) \\ & \{\text{distributivity of "}\cap\text{" over "}\cup\text{"}\} \\ & \supseteq \bigcup_{m_1=0}^{k/2-1} (sup(c_{2m_1}, c_{2m_1+1}) \cap (sup(c_{2m_1-1}, c_{2m_1}) \cup \\ & \qquad \cup sup(c_{2m_1+1}, c_{2m_1+2}))) \\ & \{\text{selecting } m_2 = m_1 \text{ and } m_2 = m_1 + 1\} \\ & = \bigcup_{m=0}^{k-1} (sup(c_m, c_{m+1}) \cap sup(c_{m-1}, c_m)) \\ & \{\text{substituting } m = 2 \cdot m_1\} \\ & \supseteq \bigcup_{m=0}^{k-1} c_m \\ & \{sup(c_{m-1}, c_m) \cap sup(c_m, c_{m+1}) \supseteq c_m\} \\ & = F_f^* \end{aligned}$$

To show that $F_g \cap F_h \cap R_f = \emptyset$, we observe that the third row in the above proof, namely:

$$\bigcup_{m_1=0}^{k/2-1} \left(sup(c_{2m_1}, c_{2m_1+1}) \cap \bigcup_{m_2=0}^{k/2-1} sup(c_{2m_2-1}, c_{2m_2}) \right)$$

can be further expanded to a union of terms $sup(c_i, c_{i+1}) \cap sup(c_j, c_{j+1})$ over all $i \in \{0, 2, 4, \ldots, k-2\}$ and all $j \in \{1, 3, 5, \ldots, k-1\}$. Since by (2) this union is in $F_f \cup D_f$, we can conclude that $F_g \cap F_h \cap R_f = \emptyset$ holds.

□

Since $F_g \cap F_h$ covers all k cubes in F_f^*, $|F_g| + |F_h| \leq |F_f^*|$. The number of cubes in F_g and F_h can often be further reduced by applying standard two-level AND-OR minimization techniques to g and h.

As an example, consider a 5-variable function f shown in Figure 1. Assume $c_0 = 00000$, $c_1 = 00101$, $c_2 = 01111$,

$\backslash x_1$	0				1			
$x_4x_5 \backslash x_2x_3$	00	01	11	10	10	11	01	00
00	1	0	0	0	1	0	0	0
01	0	1	0	0	0	1	0	0
11	0	0	1	0	0	0	1	0
10	0	0	0	1	0	0	0	1

Figure 1. An example function.

$c_3 = 01010$, $c_4 = 10010$, $c_5 = 10111$, $c_6 = 11111$ and $c_7 = 11000$. Let us check whether condition (2) is satisfied. Since $k = 8$, we have to check all combinations of $i \in \{0, 2, 4, 6\}$ and all $j \in \{1, 3, 5, 7\}$:

$sup(c_0, c_1) \cap sup(c_1, c_2) = c_1$
$sup(c_0, c_1) \cap sup(c_3, c_4) = \emptyset$
$sup(c_0, c_1) \cap sup(c_5, c_6) = \emptyset$
$sup(c_0, c_1) \cap sup(c_7, c_0) = c_0$
$sup(c_2, c_3) \cap sup(c_1, c_2) = c_2$
$sup(c_2, c_3) \cap sup(c_3, c_4) = c_3$
$sup(c_2, c_3) \cap sup(c_5, c_6) = \emptyset$
$sup(c_2, c_3) \cap sup(c_7, c_0) = \emptyset$
$sup(c_4, c_5) \cap sup(c_1, c_2) = \emptyset$
$sup(c_4, c_5) \cap sup(c_3, c_4) = c_4$
$sup(c_4, c_5) \cap sup(c_5, c_6) = c_5$
$sup(c_4, c_5) \cap sup(c_7, c_0) = \emptyset$
$sup(c_6, c_7) \cap sup(c_1, c_2) = \emptyset$
$sup(c_6, c_7) \cap sup(c_3, c_4) = \emptyset$
$sup(c_6, c_7) \cap sup(c_5, c_6) = c_6$
$sup(c_6, c_7) \cap sup(c_7, c_0) = c_7$

Since all resulting intersections are either contained in F_f or empty, condition (2) is satisfied. Thus, f can be represented as $f = g \cdot h$, with $g = \overline{x}_1\overline{x}_2\overline{x}_4 + \overline{x}_1x_2x_4 + x_1x_2\overline{x}_4 + x_1\overline{x}_2x_4$ and $h = \overline{x}_3x_3x_5 + x_1x_3x_5 + \overline{x}_3\overline{x}_4\overline{x}_5 + \overline{x}_3x_4\overline{x}_5 = x_3x_5 + \overline{x}_3\overline{x}_5$.

4. The algorithm

We use Theorem 1 to design an algorithm for deciding whether a function has a compact AND-OR-AND expression. The algorithm searches for a largest subset of F_f satisfying condition (2), represents f as $f = (g \cdot h) + r$ and estimates the size of the "reminder" r. The greater the fraction F_f/F_r, the more likely f to have an AND-OR-AND expression with a smaller number of products than the one in AND-OR expression.

The input is the on-set F_f, don't care-set D_f and off-set R_f of f, and the output is the on-sets F_g, F_h and F_r, of g, h and r, correspondently. The algorithm repeats the following basic steps:

1. Choose an initial pair of cubes, c_0 and c_1, and compute their supercube $sup(c_0, c_1)$;
2. Check whether a cube c_2 can be found, such that $sup(c_0, c_1) \cap sup(c_1, c_2) \subset (F_f \cup D_f)$;
3. Repeat step 2 until either a cube c_{k-1} is found, such that $sup(c_{k-1}, c_0) \cap sup(c_0, c_1) \subset (F_f \cup D_f)$, or until condition (2) is not met for some i and j;
4. Repeat 1, 2 and 3 for all pairs of c_0 and c_1, updating best solution after each iteration;
5. Compute the reminder $F_r = F_f - (F_g \cap F_h)$.

The pseudocode of the algorithm is shown in Figure 2. The subroutine **FindNext** (Figure 3) takes two cubes, c_1 and c_2, and finds a cube c_3 such that $sup(c_1, c_2) \cap sup(c_2, c_3) \subset (F_f \cup D_f)$. c_{in} is the initial cube of the sequence of cubes which we currently check. It is passed as an argument with each recursive call of **FindNext**. **FindNext** terminates when a cube is found whose supercube with the initial cube can "connect" the generated sequence of cubes in a loop. If **FindNext** returns 1, then a sequence of cubes satisfying condition (2) is found. **LoopCost** function checks whether the obtained sequence of cubes has a smaller cost that the sequences found in previous iterations of inner **for**-loop. The cost is computed as $(N_{literals}(F_f) - (N_{literals}(F_g) + N_{literals}(F_h)))/(N_{literals}(F_f))$. If at least one sequence is found for a given c_0, the flag *found_loop* is raised to 1. After all choices of c_1 are tried for a given c_0, the F_g and F_h with the best cost (if found) are unioned with previously selected the F_g and F_h, enlarging the subset of F_f covered by $F_g \cap F_h$. In **FindNext**, the coloring of sup_{next} is always performed not only with respect to the currently created F_g and F_h, but also with respect to the already accepted and stored F_g and F_h. This guarantees that the relations $F_g \cap F_h \supseteq F_f^*$

Check_AND_OR_AND(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: sets of cubes F_g and F_h satisfying condition (2), and F_r such that $(F_g \cap F_h) + F_r \supseteq F_f$

```
for (each c0 ∈ Ff) {
    found_loop = 0;
    for (each c1 ∈ Ff) {
        Replace c0 and c1 by their supercube sup;
        flag = FindNext(c0, sup, sup, c0, c1);
        if (flag = 1) {
            found_loop = 1;
            cost = LoopCost(Fg, Fh, Ff);
            if (cost < best_local_cost) {
                best_local_cost = cost;
                Update best_Fg and best_Fh;
            }
        }
    }
    if (found_loop = 1) {
        best_Fg = best_Fg ∪ Fg;
        best_Fh = best_Fh ∪ Fh;
    }
}
Fr = Ff − (best_Fh ∩ best_Ff);
return(best_Fg, best_Fh, Fr);
```

Figure 2. Pseudocode of the algorithm.

FindNext$(c_{in}, sup_{in}$, sup, $c_1, c_2)$
input: inital cube c_{in} and supercibe sup_{in}, current supercube *sup* of cubes c_1 and c_2
output: returns 1 if a sequence satisfying condition (2) was found, 0 if not

```
/* Termination - sequence closes in a loop */
if(Color(sup_in) = Color(sup)) {
    sup_next = Union(c_in, c2);
    if(can color sup_next in opposite to Color(sup) color) {
        Save sup_next in Fg or Fh, depending on its color;
        return(1);   /* success */
    }
}
/* Recursive step - finding next cube */
for (each c3 ∈ Ff) {
    sup_next = Union(c2, c3);
    if(can color sup_next in opposite to Color(sup) color) {
        Save sup_next in Fg or Fh, depending on its color;
        return(FindNext(c_in, sup_in, sup_next, c2, c3));
    }
}
return(0);   /* failure */
```

Figure 3. Pseudocode of the FindNext().

and $F_g \cap F_h \cap R_f = \emptyset$ are always satisfied, for F_f^* being the union of cubes of F_f contained in current and stored F_g and F_h. The more sequences are found, the larger is the part of F_f is covered by $F_g \cap F_h$. The algorithm terminates after all choices of the initial cube c_0 are tired.

5. Experimental results

Tables 1 and 2 shows the experimental results on some benchmark functions. 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 [11]. $|F_g|$, $|F_h|$ and $|F_r|$ are the sizes of the on-sets of g, h and r, respectively, obtained by the our algorithm and the Algorithm for AND-OR-AND optimization [2]. t_1 and t_2 are user times in seconds measured using the UNIX system command *time*. All programs were run on a Sun Ultra 60 operating with two 360 MHz CPU and with 1024 MB RAM main storage. δ_1 and δ_2 show the improvement of the algorithms over Espresso in terms of the number of cubes, computed as $\delta_1 = \frac{|F_f|}{|F_g|+|F_h|}$ and $\delta_2 = \frac{|F_f|}{|F_g|+|F_h|+|F_r|}$.

Table 1 lists the benchmarks with the reminder F_r smaller that half of the size of the initial on-set cover F_f (computed by Espresso). The condition 2 is sufficient, so we would expect the functions which satisfy it with a small reminder to have a compact AND-OR-AND expression. Since the AND-OR-AND expression obtained by our algorithm is not guaranteed to be the optimal, we also list the solutions of the algorithm for AND-OR-AND optimization [2], Surprisingly, for $ts10$ our solution is much better that the solution of [2]. We can see that all the functions have AND-OR-XOR expressions with at least 25% less products then that in their Espresso cover.

Table 2 shows the cases where the reminder takes the large part of the initial cover (more than $4/5$). In general, because the condition is not necessary, the large reminder should not necessarily imply the bad AND-OR-AND expression. However, we found that it is almost always the case in practice.

6. Conclusion

In this paper we have formulated a sufficient condition for a function f to have decomposition of type $f = g \cdot h + r$, with the total number of products in g, h and r smaller or equal than the number of products in f. Using this condition, we have designed an algorithm for deciding whether a function is likely to have a compact AND-OR-XOR expression.

Our current research includes integrating the new algorithm with the algorithm [2] to reduce the run-time of [2]. We also looking into the ways to relax the condition 2 and, if possible, to formulate a necessary condition.

Table 1. Benchmarks with $|F_r| < 1/2|F_f|$.

Example function	n	m	Espr. $\|F_f\|$	Algorithm [2] $\|F_g\|$	$\|F_h\|$	δ_1	t_1,sec	new Algorithm $\|F_g\|$	$\|F_h\|$	$\|F_r\|$	δ_2	t_2,sec	$\|F_f\|/\|F_r\|$
alu2	10	8	68	33	19	1.31	32	11	11	28	1.36	1.38	2.43
alu3	10	8	66	28	19	1.40	24	11	6	30	1.40	1.76	2.20
b9	16	5	119	49	16	1.83	108	21	21	35	1.55	4.91	3.40
radd	8	5	75	19	20	1.92	14	19	19	31	1.09	0.76	2.42
ryy6	16	1	112	2	5	16.00	379	6	1	0	16.00	0.59	∞
sym10	10	1	210	69	41	1.91	241	29	20	83	1.59	9.84	2.53
t1	21	23	102	43	18	1.67	1146	21	18	48	1.17	13.2	2.13
ts10	22	16	128	54	55	1.17	4830	32	4	39	1.71	1.41	3.28
z4	7	4	59	16	17	1.79	2.1	16	14	23	1.11	0.43	2.57
3-bit adder	6	4	31	14	8	1.41	2.80	11	8	7	1.19	0.23	4.43
4-bit adder	8	5	75	19	20	1.92	14.8	19	19	31	1.09	0.86	2.42
5-bit adder	10	6	167	55	38	1.80	78.6	41	33	73	1.14	7.42	2.29

Table 2. Benchmarks with $|F_r| > 4/5|F_f|$.

Example function	n	m	Espr. $\|F_f\|$	Algorithm [2] $\|F_g\|$	$\|F_h\|$	δ_1	t_1	new Algorithm $\|F_g\|$	$\|F_h\|$	$\|F_r\|$	δ_2	t_2	$\|F_f\|/\|F_r\|$
b10	15	11	100	76	18	1.06	219	8	8	84	1.00	5.65	1.19
bc0	26	11	179	117	57	1.03	5078	8	8	163	1.00	23.4	1.09
gary	15	11	107	80	26	1.01	412	8	8	91	1.00	5.81	1.17
in0	15	11	107	80	26	1.01	389	8	8	91	1.00	5.26	1.18
misex1	8	7	12	9	5	0.86	0.72	0	0	12	1.00	0.01	1.00
sqn	7	3	38	27	7	1.12	6.1	3	3	32	1.00	0.26	1.19

References

[1] A. A. Malik, D. Harrison, R.K. Brayton, "Three-level decomposition with application to PLDs", *IEEE Int. Conference on Computer Design*, 1991, pp. 628-633.

[2] E. Dubrova, P. Ellervee "A fast algorithm for three-level logic optimization", *Proc. Int. Workshop on Logic Synthesis*, Lake Tahoe, May 1999, pp. 251-254.

[3] T. Sasao, "A design method for AND-OR-EXOR three-level networks", *Proc. Int. Workshop on Logic Synthesis*, Lake Tahoe, May 1995.

[4] D. Debnath, T. Sasao, "A heuristic algorithm to design AND-OR-EXOR three-level networks", *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC'98)*, Yokohama, Japan, Feb. 1998.

[5] 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.

[6] A. Jabir, J. Saul, "A Heuristic Decomposition Algorithm for AND-OR-EXOR Three-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. 55-72.

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

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

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

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

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