AOXMIN: A Three-Level Heuristic AND-OR-XOR Minimizer for Boolean Functions

E.V. Dubrova D.M. Miller J.C. Muzio

VLSI Design and Test Group
Department of Computer Science
University of Victoria, P.O.Box 3055
Victoria, B.C., Canada, V8W 3P6
{elena, mmiller, jmuzio}@csr.uvic.ca

Abstract

It was previously shown that an AND-OR-XOR expansion, which is the XOR-sum of two sum-of-products expressions, has a smaller upper bound on the number of products than that of the AND-OR and AND-XOR expansions. In this paper, a new heuristic algorithm for minimizing AND-OR-XOR expansions of incompletely specified Boolean functions is presented. We also hypothesize an upper bound of $2^{n-2}+1$ on the number of products in the AND-OR-XOR expansion, which is tighter than the bounds previously reported.

1 Introduction

This paper considers the logic synthesis of circuits composed of AND, OR, XOR and NOT gates. The goal is to find a minimal circuit realization for a given function in terms of these gates (under some criteria of minimality). Fulfilling this general goal without any limitations on a circuit's structure is known to be an extremely difficult task in terms of computational complexity. To be feasible, practical algorithms put some restrictions on the problem. A common approach is to restrict the circuit to a particular topology (e.g. to a two-level AND-OR circuit). We here consider the case of the XOR sum of two sum-of-products expressions, which is known as an AND-OR-XOR expansion. It has been shown in [2] and [3] that an AND-OR-XOR expansion has a smaller upper bound on the number of products than that of the conventional AND-OR and AND-XOR expansions.

XOR gates have prohibitive cost in some technologies. However, an XOR gate can be justified if it leads to savings elsewhere in the circuit. Also, AND-OR-XOR expansion can be easily realized by an industrially offered three-level Programmable Logic Device (PLD) [4]. The first two levels are implemented by a Programmable Logic Array (PLA) and the third by a set of logic expanders. Each logic expander can be programmed to realize any logic function of two variables, and programming it to implement the XOR operation brings no disadvantage as compared to AND or OR.

Since an AND-OR-XOR expansion has three levels, finding a minimal form is much harder than finding two-level AND-OR and AND-XOR minimal expansions. An exact algorithm for solving this problem most likely has exponential time-complexity $O(2^{2^n})$, since for an n-variable function f there are 2^{2^n} possible

ways to choose two functions, say g_1 and g_2, so that $g_1 \oplus g_2$ is a realization of f. Performing an exhaustive search is out of the question for large functions. Hence, to be feasible, an algorithm for minimizing AND-OR-XOR expansions should balance the quality of the solution against the length of time required to find it. One such algorithm has been developed in [5]. In this paper we present another algorithm for minimizing AND-OR-XOR expansions.

2 Notation and Definitions

Let $f(x_1, \ldots, x_n)$ denote a Boolean function $f : B^n \rightarrow Y$, where $B = \{0, 1\}$ and $Y = \{0, 1, *\}$, with $*$ denoting a don't care value. A point in the domain B^n of the function is called a *minterm*. The *on-set* T, the *don't care-set* D and the *off-set* F of f are the sets of minterms that are mapped by f to 1, $*$ and 0, respectively. The function is specified by the triple $f = (T, D, F)$. If $D = \emptyset$, then the function is a *completely specified* function, otherwise it is *incompletely specified*. A *realization* of an incompletely specified function $g := B^n \rightarrow Y$ is any completely specified function $f : B^n \rightarrow B$ such that for every n-tuple $a \in B^n$, if $g(a) \in B$, then $f(a) = g(a)$.

A *product-term* is a product (AND) of zero or more variables or their complements, such that each variable occurs at most once. We denote product-terms by p_i, $i \geq 0$. A *sum-of-products* is a sum (OR) of product-terms.

An *AND-OR-XOR expansion* is the XOR-sum of two sum-of-products expressions. A *minimal* AND-OR-XOR expansion is the expansion with the smallest number of product-terms. For example, consider the function $f(x_1, x_2, x_3, x_4)$ shown in Figure 1(a). Its minimal AND-OR-XOR expansion consists of 4 product-terms, namely:

$$f(x_1, x_2, x_3, x_4) = (x_1' x_2 + x_3' x_4) \oplus (x_1 x_2' + x_3 x_4')$$

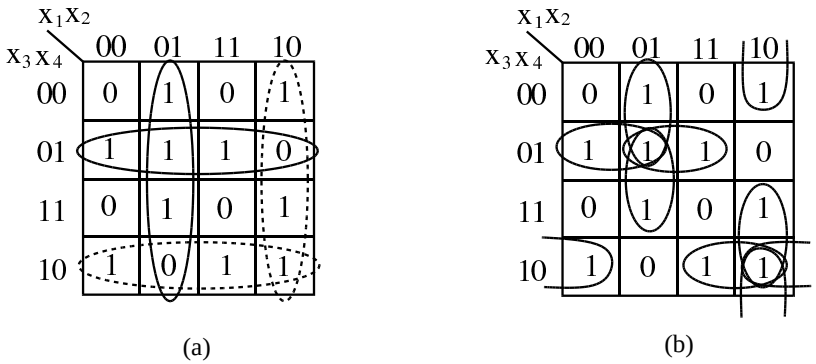

Figure 1: Karnaugh map of the function from the example.

3 An Algorithm for Minimizing AND-OR-XOR Expansions

The problem of finding a minimal AND-OR-XOR expansion can be formulated as follows:

Given a function $f = (T, D, F)$, find two functions g_1 and g_2 in sum-of-products form such that $g_1 \oplus g_2$ realizes f whenever f is defined and the total number of product-terms in g_1 and g_2 is minimized.

The basic steps of our algorithm for solving this problem are as follows:

Algorithm for minimizing AND-OR-XOR expansions (AOXMIN)
input: $f = (T, D, F)$, an integer N_{iter}
output: sum-of-product expressions for g_1 and g_2 such that $g_1 \oplus g_2$ realizes f and the total number of product-terms of g_1 and g_2 is minimized.

1. Use Espresso to minimize f.
2. Fill ON_list and OFF_list with the T and F cubes from the resulting function.
3. Initialize $g1\_best$ and $g2\_best$.
4. Using **DivideEqClasses()**, divide the cubes from ON_list into equivalence classes.
5. Using **SelectPartitioning()**, group the resulting equivalence classes into two sets, T_1 and T_2.
6. Construct $g1\_init = (T_1, F, T_2)$ and $g2\_init = (T_2, F, T_1)$.
7. Starting from $g1\_init$, invoke **SpecifyBoth()** to determine which don't cares in $g1\_init$ and $g2\_init$ should be specified to 1 so that the total number of product-terms in both functions is minimized.
8. Choose which of the pairs of functions (g_1, g_2), (g_1, g_2'), (g_1', g_2) or (g_1', g_2') has the smallest number of product-terms, and save it.
9. Repeat steps 7 and 8 starting from $g2\_init$.
10. Repeat steps 5 - 9 for N_{iter} partitionings.
11. Repeat steps 4 - 10 starting from the OFF_list.

The implementation of the algorithm uses well-known programming methods. Functions are represented dynamically by lists of cubes. Each cube is represented by a structure declared as follows:

```
typedef  struct ListofCubes {
         long int min;                /* minimum minterm value */
         long int max;                /* maximum minterm value */
         int class;                   /* equival. class to which the cube belongs */
         struct ListofCubes *next;    /* pointer to the next cube */
}
```

This approach gives very fast performance of the basic operations, required in the algorithm. Our current implementation only accommodates functions up to 32 variables. However, it can be modified to handle larger functions, by storing each of the *min* and *max* values in two (or more) full words (32 bits), instead of in just one.

In the subsequent sections we explain the basic steps in more detail.

3.1 Dividing the cubes into equivalence classes

Our heuristic is based on the observation that product-terms of a minimal sum-of-products form of a function may give some information about how to partition its on-set T into two sets. To explain this intuition, consider the Boolean function from the previous example (Figure 1(a)). The minimal sum-of-products form of this function consists of 8 product-terms, namely:

$$f(x_1,\ldots,x_4) = x_1'x_2x_3'+x_1'x_2x_4+x_1'x_3'x_4+x_2x_3'x_4+x_1x_2'x_4'+x_1x_2'x_3+x_1x_3x_4'+x_2'x_3x_4'$$

These product-terms are shown in Figure 1(b).

Suppose that we want to represent this function as $f = g_1 \oplus g_2$ and we put an additional constraint that each of the 8 product-terms from the sum-of-products form is entirely contained in either g_1 or g_2. Then we can easily see that if two product-terms, p_i and p_j intersect, i.e. if $p_i \cap p_j \neq 0$, then they both have to belong to either g_1 or g_2, since otherwise $g_1 \oplus g_2 = 0$ for the minterms in which p_i and p_j intersect. For example, $p_1 = x_1'x_2x_3'$ and $p_2 = x_1'x_3'x_4$ overlap, since $x_1'x_2x_3' \cap x_1'x_3'x_4 = x_1'x_2x_3'x_4$. If $p_1 \subseteq g_1$ and $p_2 \subseteq g_2$, when $x_1'x_2x_3'x_4 \not\subseteq g_1 \oplus g_2$, and therefore $g_1 \oplus g_2 \neq f$. This constraint allows us to reduce the search-space for possible partitionings.

Since the algorithm performs partitionings of product-terms, not single minterms, its first step is obtaining a minimal sum-of-products $\hat{f}$ realizing f. The minimization is carried out with Espresso [1]. After minimization, two lists, *ON_list* and *OFF_list*, are formed from the T and F cubes of $\hat{f}$.

The next step is to initialize *g1_best* and *g2_best*. If the number of cubes in *ON_list* is less than the number of cubes in *OFF_list*, then *g1_best* is initialized to $\hat{f}$ and *g2_best* is initialized to the constant zero function, otherwise *g1_best* $= \hat{f}'$ and *g2_best* $= 1$. In this way, for a single-output problem, the number of products generated by AOXMIN is never larger than the one that is produced by Espresso.

In the next step, the procedure **D***ivideEqClasses*() is called to divide the *ON_list* cubes into equivalence classes. A sketch of the code is shown in Figure 2. Two cubes p_i and p_j are in the same class if, and only if,

DivideEqClasses(*list_name*)
input: a list of cubes
output: for each cube the field "class" is filled with the number of the equivalence class to which it belongs

```
for (each cube a from the first to the last in the list list_name)
    for (each cube b from the next after a to the last in the list list_name)
        if (a and b intersect)
            if (b.class is not labeled)
                b.class = a.class
            else
                find all cubes with the same class as b and set them to a.class
```

Figure 2: Implementation of the subroutine **DivideEqClasses()**.

(a) $p_i \cap p_j \neq \emptyset$
or (b) $(p_i \cap p_{i+1} \neq \emptyset)$ and $(p_{i+1} \cap p_{i+2} \neq \emptyset)$ and ... and $(p_{i+k} \cap p_j \neq \emptyset)$

(a) states that p_i and p_j intersects, while (b) states that p_i and p_j are connected through a chain of intersecting cubes. Since our algorithm requires a cube to be entirely included in either g_1 or g_2, it follows that each equivalence class of cubes must be entirely contained in either g_1 or g_2.

The procedure **DivideEqClasses()** takes a list of cubes as its input, computes which cubes are connected, and, for each cube, fills the field "class" with the number of the equivalence class to which it belongs. In the main program, it is run twice - first starting from *ON_list* and then starting from *OFF_list*.

3.2 Obtaining T_1 and T_2

If the result of **DivideEqClasses()** is in more than two equivalence classes, then they should be grouped into two sets to get T_1 and T_2, which are the initial on-sets for g_1 and g_2. This is done by a procedure **SelectPartitioning()**.

The number of possible ways to group k classes into two sets is $N_{all} = 2^{k-1} - 1$. If k is large, trying all possible partitionings may result in an unreasonably long computational time. To avoid this, our program takes as a parameter an integer N_{iter}, indicating the number of partitionings we are willing to try. If N_{iter} is larger than or equals to N_{all}, then **SelectPartitioning()** successively tries all possible partitionings. Otherwise, it generates N_{iter} number of randomly chosen partitionings. As the experimental results section shows, for most practical functions 20 or less iterations are sufficient to obtain good results.

The procedure **SelectPartitioning()** is called only if the number of equivalence classes, obtained by **DivideEqClasses()**, is more than one. Otherwise, AOXMIN returns the functions *g1_best* and *g2_best* as initialized in step 3 of the algorithm. It is our experience that for the functions with just one equivalence class introducing XOR doesn't bring any advantage in terms of the number of products.

3.3 Constructing $g1\_init$ and $g2\_init$

After partitioning the set T into two subsets T_1 and T_2, the functions $g1\_init = (T_1, D_1, F_1)$ and $g2\_init = (T_2, D_2, F_2)$ are constructed, so that:

- T_1 and T_2 are the on-sets obtained after partitioning of the on-set T of f.
- $D_1 = F$ and $D_2 = F$ are the don't care sets, i.e. the off-set of function f is the don't care-set for g_1 and g_2.
- $F_1 = T_2$ and $F_2 = T_1$ are the off-sets, i.e. the on-set of g_1 is the off-set of g_1, and *vice versa*.

Figure 3 shows $g1\_init$ and $g2\_init$ for the function from Figure 1. Here the number of equivalence classes equals two, and therefore there is just one way to choose T_1 and T_2.

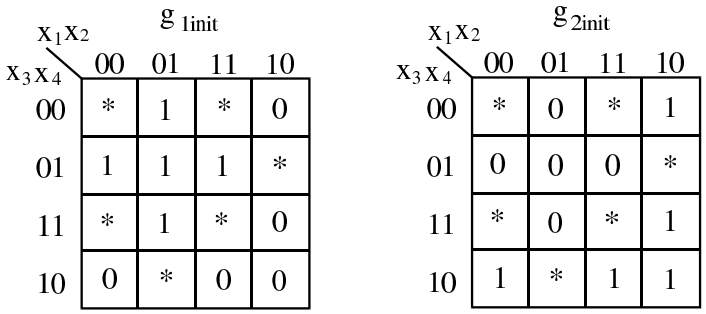

Figure 3: Functions $g1\_init$ and $g2\_init$ for the function from Figure 1.

3.4 Determining common don't cares in $g1\_init$ and $g2\_init$

After $g1\_init$ and $g2\_init$ are obtained, the next step is to determine which don't cares should be specified to be 1 so that the total number of product-terms in both functions is minimized. For this, we invoke the subroutine **SpecifyBoth()**, shown in Figure 4.

First, the function $g1\_init = (T_1, D_1, F_1)$ is minimized using Espresso, giving g_1. Then, it is determined which don't cares from $g1\_init$ were specified to 1 in g_1 by computing the difference $T^* = g_1 - T_1$. These don't cares are specified to 1 in $g2\_init$ and all other don't cares in $g2\_init$ are set to 0. The resulting function is minimized using Espresso.

In the main program, the procedure **SpecifyBoth()** is run twice - first, starting from $g1\_init$ and next starting from $g2\_init$. Each time the result with the smaller number of product-terms in the pairs of functions (g_1, g_2), (g_1', g_2'), (g_1', g_2) or (g_1, g_2') is compared to the "best" result obtained so far, and is saved in case it is less than the "best".

SpecifyBoth$(T_1, D_1, F_1, T_2, D_2, F_2)$
input: two incompletely specified functions $g1\_init = (T_1, D_1, F_1)$ and $g2\_init = (T_2, D_2, F_2)$.
output: two sum-of-products g_1 and g_2, realizing $g1\_init$ and $g2\_init$, correspondently.

$g_1 =$ Espresso (T_1, D_1, F_1);
$T_1^* = g_1 - T_1$;
$T_2 = T_2 \cup T_1^*$;
$F_2 = F_2 \cup (D_2 - T_1^*)$;
$g_2 =$ Espresso (T_2, D_2, F_2);
return (g_1, g_2);

Figure 4: Implementation of the subroutine **SpecifyBoth()**.

3.5 Multiple-output problems

Most digital systems contain multiple outputs. In our current implementation, to handle an m-output problem, we first run AOXMIN for each of the output functions to get its $g1\_best$ and $g2\_best$, and then apply Espresso to the resulting multiple-output problem consisting of $2m$ functions. This exploits common products at least to some degree. Clearly, treating m-outputs simultaneously throughout the algorithm would lead to better results.

4 Experimental Results

We have implemented the algorithm described in the previous section and have applied it to a set of benchmark functions. The benchmark functions are taken from *http://www.cbl.ncsu.edu/pub/benchmark_dirs/LGSynth91/twolexamples/*. We have compared the results of our program with the performance of the two-level AND-OR minimizer Espresso [1] (with and without output phase optimization) as well as with the results reported in [5]. AOXMIN and Espresso were run on a Sun SPARC 20 operating at 50 MHz with 64 MB RAM main storage.

Table 1 shows the number of products in the resulting functions and the time taken in seconds in columns *pr.* and *t*, respectively. The time is user time measured using the UNIX system command *time*. Column 8 shows the number of products obtained by the AND-OR-XOR minimizer from [5] for the benchmarks, reported in [5]. Unfortunately, the running times are not given in [5], so we cannot make a comparison with AOXMIN in terms of time. Columns 2 and 3 give the number of inputs n and the number of outputs m of the functions. The last column N_{iter} refers to the number of iterations, performed by our algorithm. For each function, we tried 1, 10 and 20 iterations and we show the lowest number of iterations for achieving the best result.

In terms of the number of products, for 15 of the 25 benchmarks AOXMIN gives a smaller result than Espresso without output phase optimization. On average the

Table 1: Benchmark results.

Example function	n	m	Espresso [1]				Alg. [5]	AOXMIN		
			without -Dopo		with -Dopo					
			pr.	t,sec	pr.	t,sec	pr.	pr.	t,sec	N_{iter}
5xp1	7	10	65	0.28	64	1.85	47	42	14.8	10
9sym	9	1	86	0.51	86	1.21	73	73	1.7	1
alu4	14	8	575	29.39	359	204.50	-	447	131.9	1
b12	15	9	43	0.69	29	2.72	-	31	9.1	10
bw	5	28	22	0.62	22	2.01	-	24	25.4	10
clip	9	5	120	1.50	120	8.09	92	95	10.5	10
con1	7	2	9	0.1	8	0.04	-	9	1.0	1
cordic	23	2	914	123.34	155	371.74	-	156	231.8	1
duke2	22	29	86	0.93	86	28.1	-	87	20.2	1
ex1010	10	10	284	42.24	279	202.82	-	725	109.7	1
inc	7	9	30	0.18	28	0.65	-	33	6.5	1
misex1	8	7	12	0.04	12	0.15	-	13	11.5	10
misex2	25	18	28	0.07	28	6.18	-	28	6.0	1
misex3	14	14	690	41.86	189	343.79	-	191	112.0	1
misex3c	14	14	197	13.00	199	108.48	-	197	75.1	10
rd53	5	3	31	0.05	22	0.08	-	19	15.7	20
rd73	7	3	127	0.44	93	1.05	83	83	25.5	10
rd84	8	4	255	1.65	186	3.87	-	192	61.1	10
sao2	10	4	58	0.22	37	1.03	33	38	3.7	1
squar5	5	8	25	0.07	23	0.44	-	22	4.4	1
t481	16	1	481	2.86	481	9.42	364	113	557.2	20
table3	14	14	175	3.81	175	124.50	-	176	79.3	1
table5	17	15	158	2.52	158	179.02	-	158	100.7	1
vg2	25	8	110	0.58	110	71.70	-	102	43.4	10
xor5	5	1	16	0.02	16	0.03	-	10	10.3	20

number of products obtained by Espresso is 2.13 times larger than the number of products obtained by our algorithm. However, in terms of time, Espresso without output phase optimization is on average 5.27 times faster than AOXMIN.

When compared to Espresso with output phase optimization (using -Dopo option), for 10 of the 25 benchmarks AOXMIN obtains a result with less products. On average the number of products obtained by our algorithm is 1.21 times smaller, and it is 1.17 times faster.

Compared to the AND-OR-XOR minimizer from [5], AOXMIN generates almost the same results in terms of number of products, with the exception of $t481$ function, for which our algorithm considerably reduces the number of products. We found out that, using AOXMIN, the number of products in $t481$ can be further reduced if more iterations of the algorithm are performed (see Table 2). For 50 iterations the number of products for $t481$ is 18, but the program needs 24.4 min to compute it.

The resulting sum of products expressions for the functions $g1\_best$ and $g2\_best$ are:

$$g1\_best = x'_1x_2x_3x'_4 + x'_6x_7x'_8 + x'_5x_6x_8 + x_5x_7x'_8 + x'_5x_6x'_7 + x_1x'_3 + x_1x_4 + x'_2x'_3 + x'_2x_4$$

$$g2\_best = x'_{13}x_{14}x_{15}x'_{16} + x'_{10}x_{11}x'_{12} + x_9x_{11}x'_{12} + x'_9x_{10}x_{12} + x'_9x_{10}x'_{11} + x'_{14}x_{16} + x_{13}x_{16} + x'_{14}x'_{15} + x_{13}x'_{15}$$

Notice that the functions have disjoint variable sets and interesting symmetry properties.

Table 2: AOXMIN results for $t481$.

products	time,sec	N_{iter}
228	47.4	1
134	295.7	10
113	557.2	20
18	1461.5	50

The experimental results indicate that our algorithm works quite well for functions with embedded XOR logic (like $5xp1, clip, rd53, rd73, sao2, t481$). On the other hand, for benchmarks without embedded XOR logic, like $misex1, misex2$ and $ex1010$, introducing XOR doesn't bring any advantage. The "hardest" function for our program is $ex1010$, for which the program is unable to find a result with less than 725 products for up to 100 iterations.

As can be seen, in general, the time required for AOXMIN to find a solution is quite reasonable, especially taking into account the complexity of the problem. The most time-consuming are the calls to Espresso. We hope in the future to improve the time-performance of AOXMIN by making it an integrated program. Presently the problem is treated in several stages, by first computing minimal AND-OR-XOR expansions for each output functions separately, and then finding common subterms for the resulting functions.

5 Conclusion

This paper presents a new heuristic algorithm for minimizing AND-OR-XOR expansions of incompletely specified Boolean functions. As with any heuristic algorithm, ours doesn't guarantee that a minimal solution is found, but usually obtains a nearly-minimal one. However, as the experimental results show, the algorithm has a satisfactory performance for a class of benchmark functions.

Two major facets of our method require further research. First, as noted above, we use a very simple approach to multiple-output problems with the individual functions treated separately until the final minimization step. As in conventional two-level minimization, we expect to achieve much better results if a method can be found to treat the functions together throughout the complete process. The difficulty is how to extend the notion of product term chains to the multiple-output case.

The second area requiring further work is the selection of partitionings. At present, we select these pseudo-randomly. We need to examine which functional properties can be used to guide the choice of partitionings and also to investigate which search techniques are applicable to this problem.

A further drawback of our algorithm is that it is only capable of finding the AND-OR-XOR expansions which obey the constraint that each product-term from a minimal sum-of-products form of the function is entirely contained in either g_1 or g_2, which is, of course, not necessary in general. Since the minimal sum-of-products form of a function is not unique, the performance of our algorithm could be improved by trying several minimal sum-of-products forms as starting points. But this still doesn't guarantee that the resulting solution is a minimal one.

Since our algorithm is heuristic, it is useful to have a measure for asserting the minimality of the resulting expansion. Previously, an upper bound of $5 \cdot 2^{n-4}$, on the number of product-terms in the AND-OR-XOR expansion has been reported in [3]. This result was later improved to $9 \cdot 2^{n-5}$, in [2]. We hypothesize that the upper bound on the number of product-terms in the minimal AND-OR-XOR expansion is $2^{n-2}+1$, $n > 1$, but we haven't succeeded yet in proving this conjecture.

References

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

[2] D. Debnath, T. Sasao, Minimization of AND-OR-EXOR three-level networks with AND gate sharing, *The Sixth Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '96)*, Fukuoka, Japan, Nov. 1996.

[3] Dubrova, E.V., Miller, D.M., Muzio, J.C., Upper bound on number of product-terms in AND-OR-XOR expansion of logic functions, *Electronics Letters* **31** (1995), 541-542.

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

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