Output Grouping Method Based on a Similarity of Boolean Functions

Petr Fišer, Pavel Kubalík, Hana Kubátová
Department of Computer Science and Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2
e-mail: fiserp@fel.cvut.cz, xkubalik@fel.cvut.cz, kubatova@fel.cvut.cz

Abstract

A method allowing us to efficiently group the multi-output Boolean function outputs is presented in this paper. Some kind of decomposition is usually involved in the logic synthesis process. Here the circuit has to be repeatedly divided into subcircuits, until these subcircuits become technological library elements or technology-dependent components, in general. Our output-grouping method can be used to compute which functions of a set of Boolean functions could share most of logic. This idea can be exploited in single-level decomposition, where a two-level multi-output circuit is divided into several stand-alone blocks, while retaining their two-level nature. The total size of these circuits should be kept minimal. After such single-level decomposition the circuits are further processed by a common synthesis process.

Obtained results are shown in this paper, for a set of standard MCNC and ISCAS benchmarks. Two general problems to be solved were considered for tests: an output-grouping based decomposition and a parity predictor synthesis, which is used in an on-line diagnostic design.

1. Introduction

It is always necessary to perform some kind of decomposition when designing complex VLSI circuits, with respect to available components. Most of up to now proposed methods start with a two-level Boolean network (sum-of-products) and try to decompose it into a multi-level network. The Boolean function is being manipulated so as to extract subfunctions common to more of its parts. This is being done either algebraically, by finding the function's common divisors (kernels) [1], by using computationally demanding Boolean methods [2, 3], or by using a functional decomposition [4, 5], lately based on BDDs [6, 7]. Nowadays, a functional bi-decomposition plays a big role, for it is generally usable for most of applications [8, 9, 10].

Most of the previously mentioned methods are primarily intended for single-output functions, even when they can be extended to multi-output functions. Nevertheless, there is no method strictly determining relations between the multi-output Boolean function's outputs. Our partitioning method is based on grouping the *output* variables. There can be a relationship between several outputs of the function found. The proposed method is based on computing the measure of a "similarity" of functions. When two Boolean functions are similar, there is a big chance they may efficiently share a lot of logics. Thus, grouping those "similar" functions together could be advantageous, when output decomposition is needed. If proper output grouping is found, the resulting logic of the overall design is significantly reduced.

The method naturally found its next application in the on-line BIST (Built-in Self-Test) design [11, 12]. Here the functions are grouped together to form parity bits of the parity predictor. The parity groups are generated from the original circuit outputs, by successively XOR-ing them. A choice of outputs to be XORed plays an important role in the resulting area overhead.

The paper is structured as follows: the principles of the single-level partitioning are given in Section 2, the principles of our output grouping method based on a similarity of functions is described in Section 3. Its application to on-line BIST is shown in Section 4, Section 5 concludes the paper.

2. Output Grouping

Let us consider a need to divide a circuit into several stand-alone blocks having a limited number of outputs (into e.g., PLAs, PALs, GALs). These blocks have to be synthesized separately then, since they cannot share internal signals. The blocks can share input variables only. Such a case of decomposition will be denoted as a *single-level partitioning*, since the number of levels of the circuit remains unchanged, see Fig. 1.

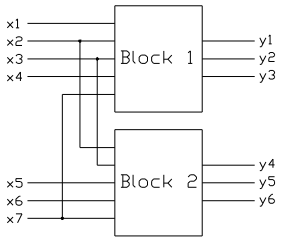

Figure 1: Single-level partitioning

Our task is to determine which outputs should be grouped together, i.e., finding an *output grouping*. If the outputs that are to be grouped together are properly determined, the complexity of the individual blocks is reduced.

Since the blocks cannot share the group terms (or subfunctions when they are designed as multi-level), the total complexity of the overall design must be naturally increased, in comparison to the all-in-one design. When our technique is used, this cost of the decomposition is reduced to minimum.

3. Similarity-Based Output Grouping

The method is based on joining functions (i.e., outputs of multi-output functions) that are somehow similar. Then there should be a big chance that the two functions will share a lot of logics. The task is to determine how to compute the measure of similarity of two Boolean functions.

The main idea is based on these straightforward observations:

(I) Two equal functions are “very” similar
(II) Two inverse functions are very similar too, since they could differ at most by one inverter in the final (multilevel) design

These two criteria could be combined together to form a new, general one:

(III) Two functions are similar, if a change of a value of one input variable induces a change of values of *both* the functions, or the values of *both* functions do not change. This should be checked for all possible input variable changes.

To quantify a vague term of a “similarity” of two Boolean functions, a *scoring function* is introduced. To compute the value of the scoring function, all the functions’ minterms are processed. For each minterm each input variable value is switched and values of the outputs of the two functions are observed. If both values remain unchanged, the scoring function is increased by one, since this represents the same behavior of these two functions. If both values change, regardless the logic values, the scoring function is increased by one as well. In other cases, when one output value changes and one not, the score remains unchanged.

The complexity of this algorithm is $O(n.2^n)$, where n is the number of input variables. For all the 2^n minterms n variable swaps are explored. The actual complexity can by reduced to $\frac{1}{2}\, n.2^n$. Only $0 \rightarrow 1$ swaps can be considered, since all the reverse swaps will yield the same result, thus just doubling the score.

Two equal functions will obtain the highest score by this algorithm. Two inverse functions will obtain the highest score as well.

Such an approach is very straightforward and apparently inefficient, due to its prohibitively high complexity. However, we have used this approach in our experiments, since functions described by minterms were needed for on-line BIST design [12], see Section 4.

Nevertheless, the scoring function can be computed in a much more efficient way: by using a Boolean difference function [16]. A Boolean difference of a function $f(x_0, \ldots x_n)$, with respect to an input variable x_i can be computed as:

$$\frac{\partial f(x_0,...,x_n)}{\partial x_i} = f(x_0,...,x_i = 1,...,x_n) \oplus f(x_0,...,x_i = 0,...,x_n) \quad (1)$$

As a result we obtain a Boolean function which is equal to 1, if a change of x_i induces a change of $f(x_0, \ldots x_n)$. Thus, the size of the Boolean difference function (i.e., number of its 1-minterms) describes the number of 1-minterms of $f(x_0, \ldots x_n)$ for which a change of x_i induces a change of $f(x_0, \ldots x_n)$. To derive the cubes for which two functions simultaneously change their value by changing a value of x_i, we compute Boolean differences of these two functions with respect to x_i and compute their intersection, i.e., a Boolean product. The size of this product will correspond to the number of minterms, for which both functions will change a value, if x_i changes.

Similarly, the number of minterms, for which both functions will not change a value, if x_i changes can be computed using a Boolean indifference, i.e., a negation of a Boolean difference:

$$\frac{\overline{\partial} f(x_0,...,x_n)}{\overline{\partial} x_i} = f(x_0,...,x_i = 1,...,x_n) \Leftrightarrow f(x_0,...,x_i = 0,...,x_n) \quad (2)$$

As a result, the scoring function for functions $f(x_0, \ldots x_n)$ and $g(x_0, \ldots x_n)$ is computed as:

$$s = \sum_{i=0}^{n-1} \left(\left| \frac{\partial f(x_0,...,x_n)}{\partial x_i} \cdot \frac{\partial g(x_0,...,x_n)}{\partial x_i} \right| + \left| \frac{\overline{\partial} f(x_0,...,x_n)}{\overline{\partial} x_i} \cdot \frac{\overline{\partial} g(x_0,...,x_n)}{\overline{\partial} x_i} \right| \right) \quad (3)$$

Notice that the complexity of the computation of the score is polynomial with n, thus it can be used for any problem sizes. Such a method is applicable to functions described by any algebraic expression.

3.1. Example

A very simple example of the score computation is shown here, to illustrate the principles of the method. The score for the two following functions is to be computed:

$$\begin{aligned} f_1 &= a + bc \\ f_2 &= ab\bar{c} + \bar{a}c + a\bar{b}c \end{aligned} \quad (4)$$

Boolean differences with respect to all input variables are computed for both functions and sizes of their products are summed:

$\frac{\partial f_1}{\partial a} = (1 + bc) \oplus (bc) = \bar{b} + \bar{c}$	$\frac{\partial f_1}{\partial b} = (a + c) \oplus a = \bar{a}c$	$\frac{\partial f_1}{\partial c} = (a + b) \oplus a = \bar{a}b$
$\frac{\partial f_2}{\partial a} = (b\bar{c} + \bar{b}c) \oplus c = b$	$\frac{\partial f_2}{\partial b} = (a\bar{c} + \bar{a}c) \oplus (\bar{a}c + ac) = a$	$\frac{\partial f_2}{\partial c} = (\bar{a} + a\bar{b}) \oplus ab = 1$
$\frac{\partial f_1}{\partial a} \cdot \frac{\partial f_2}{\partial a} = (\bar{b} + \bar{c})b = b\bar{c}$	$\frac{\partial f_1}{\partial b} \cdot \frac{\partial f_2}{\partial b} = 0$	$\frac{\partial f_1}{\partial c} \cdot \frac{\partial f_2}{\partial c} = \bar{a}b$
$\lvert b\bar{c} \rvert = 2$	$\lvert 0 \rvert = 0$	$\lvert \bar{a}b \rvert = 2$

Figure 2: Difference computation example

The total score obtained from the difference computation is 2 + 0 + 2 = 4.

Now the indifference score has to be computed and added, to obtain the total score:

$\frac{\overline{\partial} f_1}{\overline{\partial} a} = (1 + bc) \Leftrightarrow (bc) = bc$	$\frac{\overline{\partial} f_1}{\overline{\partial} b} = (a + c) \Leftrightarrow a = a + \overline{c}$	$\frac{\overline{\partial} f_1}{\overline{\partial} c} = (a + b) \Leftrightarrow a = a + \overline{b}$
$\frac{\overline{\partial} f_2}{\overline{\partial} a} = \left(b\overline{c} + \overline{b}c\right) \Leftrightarrow c = \overline{b}$	$\frac{\overline{\partial} f_2}{\overline{\partial} b} = \left(a\overline{c} + \overline{a}c\right) \Leftrightarrow \left(\overline{a}c + ac\right) = \overline{a}$	$\frac{\overline{\partial} f_1}{\overline{\partial} c} = \left(\overline{a} + a\overline{b}\right) \Leftrightarrow ab = 0$
$\frac{\overline{\partial} f_1}{\overline{\partial} a} \cdot \frac{\overline{\partial} f_2}{\overline{\partial} a} = (bc).\overline{b} = 0$	$\frac{\overline{\partial} f_1}{\overline{\partial} b} \cdot \frac{\overline{\partial} f_2}{\overline{\partial} b} = \overline{a}\,\overline{c}$	$\frac{\overline{\partial} f_1}{\overline{\partial} c} \cdot \frac{\overline{\partial} f_2}{\overline{\partial} c} = 0$
$\lvert 0 \rvert = 0$	$\lvert \overline{a}\,\overline{c} \rvert = 2$	$\lvert 0 \rvert = 0$

Figure 3: Indifference computation example

The total score obtained from the indifference computation is 0 + 2 + 0 = 2. Thus, by summing these two scores we obtain the total score of 6.

3.2. Scoring Matrix

By computing the score for each pair of output variables we obtain a *scoring matrix*. It is a symmetric matrix of dimensions (*m*, *m*), where *m* is the number of output variables. The value in a cell [*i*, *j*] represents a scoring function value for variables *i* and *j*. The multi-output function's outputs are grouped together according the scoring matrix values. The output-grouping algorithm proceeds as follows:

1. Assign the first output variable to the first block. Since there is no relationship between outputs and blocks yet, it can be freely done.
2. Find the maximum scoring matrix value, corresponding to outputs *i* and *j*. These outputs should be grouped together, since their "similarity value" is the highest one.
3. If one of these outputs is already assigned to a block, append the second one, if possible (maximum number of block's outputs is not exceeded).
4. If none of them is assigned, try to find an empty block and assign both outputs to this block.
5. If no free block is available, try to put them both into some block.
6. If there is not enough place to put both the outputs into one block, assign them randomly.

This simple algorithm yields an assignment of all output variables to the blocks, while the function's similarity is exploited. Such an algorithm could be further refined by analyzing the scoring matrix more thoroughly, in order to find groups of outputs more precisely. This will be the aim of our further research.

3.3. Experimental Results

We have evaluated the efficiency of the algorithm on some of the MCNC benchmarks. For each of the benchmark circuits we have performed three experiments:

- First, the respective benchmark circuit has been minimized by BOOM [13]. This experiment has been done to estimate the circuit size when no partitioning is used.
- In the second group of experiments we have divided the circuit into several blocks (*b*), while all the output variables were assigned to the individual blocks *purely at random*. Then the circuit has been minimized by BOOM. 100 experiments were performed and an average value was taken, to ensure good statistical values.
- Finally the similarity-based output grouping method was used. We have made an experiment similar to the previously described one, but the output variables were assigned to the blocks using the proposed method.

These three experiments will show the differences between the all-in-one implementation of the benchmark circuit, the circuit divided into several blocks with randomly assigned outputs and our method. The number of blocks was selected accordingly the number of the circuit's outputs, to be somehow balanced with the number of outputs. However, any circuit may be divided into an arbitrary number of blocks, without loosing the efficiency of the algorithm.

The benchmark results are shown in Table 1. After the benchmark name the numbers of the primary inputs (*i*) and outputs (*o*) of the circuit are presented. The next column gives the number of gate equivalents [14] of the minimized circuit. Next, there is the number of blocks, into which the circuit is being decomposed. The numbers of outputs of all the blocks were kept equal. The "*Random output*

grouping" columns shows the minimization results, for the experiment where the outputs are assigned to the blocks randomly. The "*Similarity-based output grouping*" labeled columns describe the results obtained by our newly proposed method. The last column, "*impr.*" shows the improvement against the previous (random) method.

Table 1. Output grouping results

			No decomp.		Random output grouping	Similarity-based output grouping	
bench	*i*	*o*	*GEs*	*blocks*	*GEs*	*GEs*	*impr.*
al2	16	47	206.5	5	244.0	218.0	10.7%
amd	14	24	334.5	3	460.0	429.0	6.7%
b2	16	17	989.5	4	2018.5	1807.5	10.5%
b7	8	31	81.0	4	105.0	88.5	15.7%
b11	8	31	81.0	4	105.5	87.5	17%
br1	12	8	130.0	3	215.0	186.5	13%
dk17	10	11	70.5	3	84.0	72.5	13.7%
exps	8	38	910.0	4	1473.0	1256.0	14.7%
luc	8	27	162.5	3	244.0	228.0	6.6.%

4. Application to On-Line BIST

The above-described algorithm can be very efficiently applied to on-line BIST (Built-in Self-Test). The parity predictor is used to generate proper output parity, see Figure 2. The parity predictor is designed by duplicating an original combinational circuit. The output nets of the duplicate circuit are XORed together to obtain output check bits. The predictor outputs are being gradually XORed in the design process, until one or more parity bits are obtained (see Fig. 4). Two nets are XORed together in each step, by using the scoring function [see Fig. 5]. Thus, the number of outputs (now the parity bits) is being gradually decreased, until the required number of parity bits is obtained. The scoring matrix is recomputed after each step, to reflect the newly obtained output function.

The number of final parity bits (check bits) selected influences the area overhead, together with the dependability parameters [12]. Thus, proper number of parity bits has to be chosen, according to the designer's needs.

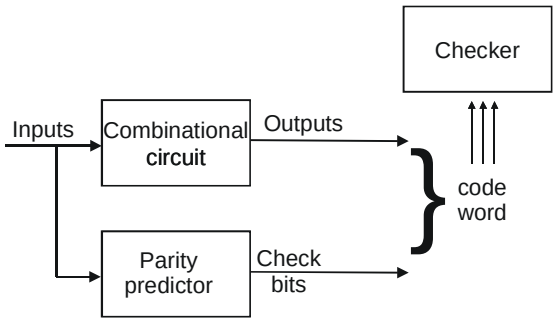

Figure 4: The on-line diagnostics design

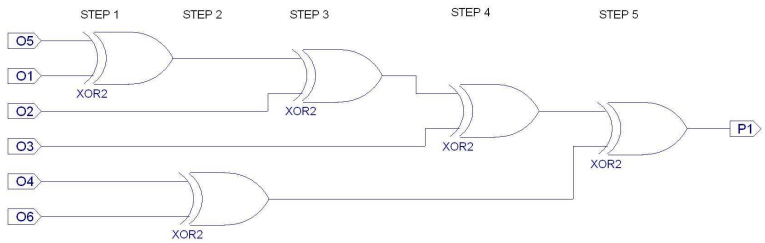

Figure 5: The parity prediction

4.1. Parity Bits Grouping

An algorithm used for grouping the circuit's outputs to form the parity bits is described here. The selection of outputs to be joined by a XOR gate is of a key importance for the final design area overhead.

Since the parity predictor is constructed by gradually joining the original circuit's outputs by 2-input XOR gates, our primary task is to properly choose the two outputs to be joined in each step. The function similarity-based approach can be exploited very well. The basic idea of the algorithm is based on these facts and assumptions:

(1) When two equal functions are joined by a XOR gate, the resulting value will be '0' for all minterms. If the values of two functions will differ in a couple of minterms only, there will be only several '1' values in the resulting XORed function. Experiments show that a low number of '1's at the output is very advantageous for the subsequent minimization process (Fig. 6).
(2) Two inverse functions, when XORed, yield a '1' value for each minterm. If the output values of two functions are inverse but a few minterms, there will be only few '0' values in the result. This is advantageous for the minimization too (Fig. 6).
(3) And, consequently, if two functions are "similar", there is a big probability that they will share a lot of logic in the implemented design. If these functions are joined together, there is a big chance for an overall area reduction.

The first two statements were based on an assumption, that it is advantageous for the minimization, when function values are either '0's or '1's for most of minterms. This is documented in Fig. 6. A typical dependency of an area overhead on the number of '1' values in the output is shown. 100-input and 20-output functions with 100 terms defined were minimized in this experiment. The number of '1's in the output was changing from 10% to 90% while the number of gate equivalents [14] of the circuit obtained after a minimization using BOOM [13] was measured. We can see that low or high values of the ratio of '1's to '0's involve best solutions.

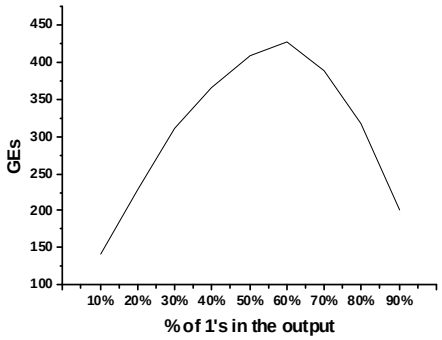

Figure 6. Dependency of the area overhead on the ratio of output '1's

The functions are described by values of all minterms, i.e., functionally, not by a netlist. Thus, the final checker design has to be synthesized "from scratch". This brings us an advantage, since the synthesis process is able to recognize the similarity of functions and design the decoder efficiently.

4.2. On-Line BIST Experimental Results

Like in the previous set of experiments, our method is compared with a purely randomized method here. All the circuit complexity values are measured as gate equivalents [14], obtained after the synthesis. An area reduction obtained by the proposed method, with respect to the random method, is shown in the "*Red.*" column. The random assignments were run 500-times and the values averaged.

Two-bit parity has been chosen for these experiments, thus the outputs were gradually XORed, until only two remained.

Sometimes there can be observed a very significant improvement with respect to the random method, up to more than 90%.

Table 2. Comparison results

Circuit	*Random [GEs]*	*Similarity [GEs]*	*Red.*
alu1	967	156	83.9 %
apla	128	76	40.6 %
b11	36	21	41.7 %
br1	80	68	15 %
alu2	418	40	90.4 %
alu3	433	320	26.1 %
s1488	364	241	33.8 %
s386	87	73	16.1 %

5. Conclusions

A novel circuit decomposition and output grouping method is presented in this paper. It is based on an evaluation of a "similarity" of Boolean functions. Functions that are found to be "similar" share a lot of logic, thus, when they are grouped together, many resources are spared. The output grouping retains a two-level nature of the circuit, hence we call it a single-level partitioning.

A very efficient application of the method to an on-line BIST design is proposed. Here the circuit outputs are joined together by XOR gates, to form a parity predictor. The parity predictor outputs are compared with the outputs of the original circuit, and thus the proper circuit's function is checked. The proposed method helps to reduce the parity predictor logic overhead to minimum. The area overhead reduction sometimes reaches more than 90% with when compared to a random method.

The results obtained by using our method are presented and compared with a random-based approach. Standard MCNC and ISCAS benchmarks were used.

Acknowledgement

This research has been supported by MSMT under research program MSM 6840770014 and by a grant GA102/04/0737

References

[1] R. K. Brayton, C. T. McMullen: The Decomposition and Factorization of Boolean Expressions, In Proc. of the IEEE International Symposium on Circuits and Systems, pp. 49-54, 1982

[2] S. Muroga, Y. Kambayashi, J. C. Lai, J. N. Culliney: The Transduction Method – Design of Logic Networks Based on Permissible Functions, IEEE Trans. on Computers, C-38(10), pp. 1404-1424, 1989

[3] T. Stanion, C. Sechen: Boolean Division and Factorization using Binary Decision Diagrams, IEEE Trans on CAD, CAD-13(9), pp. 1179-1184, 1994

[4] R. L. Ashenhurst: The Decomposition of Switching Functions, In Proc. of International Symposium on the Theory of Switching, pp. 74-116, 1957

[5] J. P. Roth, R. M. Karp: Minimization over Boolean Graphs, IBM Journal of Research and Development, Vol. 6, No. 2, pp. 227-238, 1962

[6] R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. on Computers, C-35(8), pp. 677-691, 1986

[7] Y. T. Lai, M. Pedram, S. Vrudhula: BDD Based Decomposition of Logic for Functions with Applications to FPGA Synthesis, In Proc. Design Automation Conference, pp. 642-647, 1993

[8] T. Sasao, J. T. Butler: On Bi-Decompositions of Logic Functions, ACM/IEEE International Workshop on Logic Synthesis, Tahoe City, California, 1997

[9] A. Mischenko, B. Steinbach, M. Perkowski: An Algorithm for Bi-decomposition of Logic Functions, In Proc. of Design Automation Conference, pp. 103-108, 2001

[10] L. Jozwiak, S. Bieganski: Information Trans-coders in Information-driven Circuit Synthesis, Proc. 30th Euromicro Symposium on Digital Systems Design (DSD'04), Rennes (FR), 31.8. - 3.9.04, pp. 288-297

[11] P. Kubalík, P. Fišer, H. Kubátová: Minimization of the Hamming Code Generator in Self Checking Circuits, Proceedings of the International Workshop on Discrete-Event System Design - DESDes'04. Zielona Gora: University of Zielona Gora, 2004, s. 161-166

[12] P. Kubalík, P. Fišer, H. Kubátová: Fault Tolerant System Design Method Based on Self-Checking Circuits, Proc. 12th International On-Line Testing Symposium 2006 (IOLTS'06), Lake of Como, Italy, July 10-12, 2006

[13] J. Hlavička, P. Fišer: BOOM - A Heuristic Boolean Minimizer, Computers and Informatics, Vol. 22, 2003, No. 1, pp. 19-51

[14] G. De Micheli: Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994

[15] D. K. Pradhan: Fault-Tolerant Computer System Design, Prentice-Hall, Inc., New Jersey, 1996

[16] Ch. Posthoff, B. Steinbach: Logic Functions and Equations – Binary Models for Computer Science, Springer, Berlin, Heidelberg, New York, 2004, pp. 1 – 392