設計自動化 55－8
(1990.12.12)

ＡＮＤ－ＥＸＯＲ論理式の最小化について

ダニエル・ブランド *

笹尾 勤 **

* ＩＢＭ ワトソン研究所

** 九州工業大学情報工学部

あらまし　最初に，繰り返し改善法を用いて積項数最小のＡＮＤ－ＥＸＯＲ二段論理式（ＥＳＯＰ）を導出できるための書換え規則の条件を考察している．最小解を求めるためには積項数が増加するような書換え規則が必須である．次に，ＥＳＯＰの変換を示している．一つの論理関数ｆ１のＥＳＯＰＦ１にある種の変換を施す事により，別のＥＳＯＰ　Ｆ２を得る．Ｆ２が表現する論理関数をｆ２とするとき，ｆ１の最小ＥＳＯＰ（ＭＥＳＯＰ）の積項数と，ｆ２のＭＥＳＯＰの積項数は等しい．さらに，ｆ１のＭＥＳＯＰに同じ変換を施すことにより，ｆ２のＭＥＳＯＰが得られる．この変換は，一つの同値関係を生成する．一つの同値類中には多数の論理関数が存在するが，この事実より，同じ同値類に属する関数ならばどの関数を簡単化してもよいことになる．最後にＥＳＯＰ簡単化のための非決定性アルゴリズムを示している．このアルゴリズム用いていくつかのベンチマーク関数を簡単化したところ，計算時間はかかったが既存のどのアルゴリズムよりもよい解が生成できた．

ON THE MINIMIZATION OF AND-EXOR EXPRESSIONS

Daniel Brand *

Tsutomu Sasao **

* IBM T.J. Watson Research Center, Yorktown Heights, N.Y. 10598, U.S.A.

** Department of Computer Science and Electronic Engineering
Kyushu Institute of Technology, Iizuka 820, Japan

Abstract: This paper considers conditions for generating minimal exclusive-of sum-of-products expressions (MESOPs) using rewrite rules. The relevance of these conditions in practice is evaluated. To obtain MESOPs, rewrite rules increasing the number of products are necessary. Transformations of expressions preserving the number of products in MESOPs are also presented. A non-deterministic algorithm using rewrite rules are the transformations is presented. Experimental results using various benchmark functions show that the algorithm produces better solutions than other heuristic methods.

1. Introduction

Minimization of AND-OR expressions always played an important role in logic synthesis. The problem has been relatively well understood and there are many good minimization algorithms[QUI 55, HON 74, MUR 79, BRA 84, SAS 84]. The problem of minimizing AND-EXOR expressions, on the other hand, has not been solved to the same degree. Such expressions are of interest because circuits based on EXOR elements are sometimes more advantageous from area, speed and testability point of view[MUL 54, RED 72, FUJ86, SAS 90a]. Error detecting circuitry is a typical example, where EXORs are heavily used.

Therefore a lot of work has been done on the minimization of AND-EXOR expressions [EVE 67, MUK 70, PAP 79, SAL 79, ROB 82, BES 83, FLE 87, HEL 88, SAS 90b]. This paper analyzes the method of rewrite rules [EVE 67, ROB 82, FLE 87, SAS 90b], which has proved to be a successful heuristic approach generating relatively good solutions in reasonable time.

Before rewrite rules can be applied, a given function must first be converted to some (non-optimal) two-level AND-EXOR representation. There are several ways of doing that. For examples, a two-level AND-OR representation can be modified to consist of disjoint terms only, and thus can also be considered an AND-EXOR expression.

From now on we will assume that we are given a two-level AND-EXOR representation to be minimized. We will use the symbol $\oplus$ to denote EXOR, concatenation to denote AND, and $\bar{}$ to denote negation. Our measure of minimality will be the number of product terms in the EXOR sum. Some of our results also apply to other measures, as long as the number of product terms remains the most important.

Once we have an AND-EXOR representation, it can be simplified using rewrite rules. One of the earliest rule sets has been proposed by [EVE 67] and we paraphrase it below using slightly different notation:

$x \oplus \bar{x} \rightarrow 1$ (S1)
$x \oplus 1 \rightarrow \bar{x}$ (S2)
$xy \oplus \bar{y} \rightarrow 1 \oplus \bar{x}\,y$ (S3)
$xy \oplus \bar{x}\,\bar{y} \rightarrow \bar{x} \oplus y$ (S4)
$xy \oplus \bar{x}\,\bar{y} \rightarrow x \oplus \bar{y}$ (S5)

While the authors did not state so explicitly, it is safe to assume that their simplifier also contained the rule

$x \oplus x \rightarrow 0$ (S6)

In addition, the commutative, associative and distributive laws are needed in the application any rules. To illustrate the use of rewrite rules, consider the following chain of minimization steps:

$x\,y\,\bar{z} \oplus \bar{y}\,\bar{z} \oplus \bar{x}\,y\,z$
(apply (S3) to first and second term)
$\bar{z} \oplus \bar{x}\,y\,\bar{z} \oplus \bar{x}\,y\,z$
(apply (S1) to second and third term)
$\bar{z} \oplus \bar{x}\,y$

The authors of [EVE 67] made the following statement about their rules (S1) - (S6): "The procedure does not guarantee minimality; it contains branching points and our experience shows that all branches lead to fairly economical expressions, but it is unknown whether at least one chain leads to a minimal expression."

The question posed by [EVE 67] is the very subject of this paper. In particular we will answer it for the above set of rules. This paper considers these problems:

1) Given a set of rules, is there always a chain leading to a minimal expression?
2) If the answer to the first question is "no", what rules need to be added?
3) If the answer to the first question is "yes", but not every chain leads to a minimum, how to find a chain that does result in a minimal expression?
4) To what extent are these issues relevant in practice?

We do not argue that methods guaranteeing optimality are better than those without such a guarantee. But understanding the requirements for optimality can help in designing better methods.

The paper is organized as follows.
Section 2. considers the first and second question. Section 3. considers the third question. Sections 4. and 5. consider the fourth question.
Throughout the paper the subject is described informally; for a formal treatment the reader is referred to [BRA 90].

2. Conditions of Optimality

Definition: A set of rules is "convergent" iff for any expression E there exists a sequence of expressions $_0E = E, \ldots, E_n$, where E_{i+1} is obtained from E_i by one of the rules, and E_n is minimal ($n >= 0$).

By applying such rules in all possible combinations we would eventually converge to a minimal representation. Please note that our definition of convergence does

not imply termination. Even after finding a minimal expression, rules may continue to be applied because we do not assume a procedure determining whether an expression is minimal.

Most of the published rules for minimizing AND-EXOR expressions (e.g. rules (S1) - (S6))have one important property -- their right hand side has no more product terms than their left hand side.
This guarantees that no rule can increase the number of product terms and thus guarantees that the final expression has no more product terms than the original one. Unfortunately this also guarantees that the set of rules is non-convergent, because the minimum of some functions cannot be obtained without a temporarily increase in the number of product terms [BRA 90].

Thus an increase in the number of product terms is a necessary condition for convergence. We will now concern ourselves with sufficient conditions, namely, how to make rules convergent, and we already know that it must be done by term-increasing rules. Not every set of rules allowing an increase in the number of product terms is convergent; however, it is not difficult to design convergent rules.

Given any set of rules containing the two rules (S1) and (S6) (plus possibly others) one can make it convergent by adding the two rules

$$1 \rightarrow x \oplus \bar{x} \qquad (S1*)$$
$$0 \rightarrow x \oplus x \qquad (S6*)$$

To see why this is so, consider any two-level AND-EXOR expression. It can be converted into a minterm expansion by applying (S1*) to split terms and by applying (S6) to eliminate duplicate ones. Therefore (S1), (S1*), (S6), (S6*) are sufficient to derive any expression from any other, by possibly going through a minterm expansion.

We are not proposing this as practical. The rule (S6*) allows the generation of any pair of identical terms, independently of the given function. Such a procedure would not converge to a minimum very fast. Fortunately the rule (S6*) can be replaced by

$$\bar{x} \rightarrow x \oplus 1 \qquad (S2*)$$

without jeopardizing convergence [BRA 90].

For practicality we wish to have as few term increasing rules as possible. The rule (S2*) cannot be eliminated; i.e., the rules (S1), (S1*), (S6) are not convergent. However, it is possible that (S2*) could be replaced by other term non-increasing rules. This question is related to the question of minterm representation of functions:
While an increase in the number of product terms is necessary when starting from an arbitrary representation of a function, it may be unnecessary if starting from a minterm representation. We have not been able to answer this question.

3. Transformations of Expressions

Now suppose that we have a convergent set of rules and we want a chain leading to an optimal solution. Even if the rules are not convergent we may want to find a chain that simply leads to a good solution. It is not our goal here to design a simplifier that explores the search space, but rather to force a given simplifier to do so. This sections describes such a method.

We will use the following notation. Any time we write an expression

$$\bar{x}f_0 \oplus xf_1 \oplus f_2$$

mean to imply that $\bar{x}f_0$ represents all the terms containing $\bar{x}$, with $\bar{x}$ itself factored out,
xf_1 represents all the terms containing x with x itself factored out, and
f_2 represents all the terms containing neither x nor $\bar{x}$.

In order to avoid confusion regarding equality between expressions, we will write $f = g$ to mean that f and g are the same expressions, while $f \equiv g$ will mean that f and g may be different expressions, but representing the same function.

Definition:
For a given expression f and for a literal r (x or $\bar{x}$), the "r-transform" of f, written $r\odot f$ for short, is defined as follows.
Express f as $f = \bar{r}f_0 \oplus rf_1 \oplus f_2$.
Then $r\odot f = \bar{r}f_0 \oplus f_1 \oplus rf_2$.
In other word, the r-transform is obtained by deleting the literal r from any term containing it and adding it to every term containing neither r nor $\bar{r}$.

Example:
$\bar{x}\odot(x\,y \oplus \bar{x}\,\bar{z} \oplus \bar{y}) = x\,y \oplus \bar{z} \oplus \bar{x}\,\bar{y}$
$z\odot(x\,y \oplus \bar{x}\,\bar{z} \oplus z) = x\,y\,z \oplus \bar{x}\,\bar{z} \oplus 1$

If we define composition of two r-transforms in the usual way as

$$(r_1\ r_2)\odot f = r_1\odot(r_2\odot f)$$

then we can see from the definition that it is associative, and it is commutative for literals corresponding to different variables.

We will use the symbol 1 to denote the identity transform, i.e.,

$$1\odot f = f$$

Please note that each transform is its own inverse, i.e., r r = 1, and we will write

$$r^{-1} = r$$

For a sequence $R = r_1 \ldots r_n$ of transforms, if we define

$$R^{-1} = r_n^{-1}\ldots r_1^{-1}$$

then $RR^{-1} = 1$.

In case that all the transforms in the sequence are pairwise commutative then

$$R^{-1} = R.$$

Theorem:

$\bar{x}f_0 \oplus xf_1 \oplus f_2 \equiv \bar{x}g_0 \oplus xg_1 \oplus g_2$ (1)

iff

$$f_0 \oplus g_0 \equiv f_1 \oplus g_1 \equiv f_2 \oplus g_2$$

(Proof) The equation (1) is true iff it is true for both x=0 and x=1, i.e.,

$f_0 \oplus f_2 \equiv g_0 \oplus g_2$ and (2)

$f_1 \oplus f_2 \equiv g_1 \oplus g_2$ (3)

EXORing both sides of (2) with $g_0 \oplus f_2$ and eliminating identical terms

$f_0 \oplus g_0 \equiv f_2 \oplus g_2$ (4)

EXORing both sides of (3) with $g_1 \oplus f_2$ and eliminating identical terms

$f_1 \oplus g_1 \equiv f_2 \oplus g_2$ (5)

The expression (4) is actually equivalent to (2) because we can get (2) our of (4) by EXORing with $g_0 \oplus f_2$ again. Similarly (5) and (3) are equivalent.
This proves the theorem. (Q.E.D.)

Corollary: $f \equiv g$ iff $r\odot f \equiv r\odot g$
(Proof) Follows immediately by applying the theorem to both equalities. (Q.E.D.)

The theorem and its corollary allow the following simplification procedure M:

Given an expression f, chose a sequence R of r-transforms. Let g be a the result of simplifying $R\odot f$.

Then $R^{-1}\odot g$ is a simplification of f.

Moreover, $R^{-1}\odot g$ has the minimal number of terms iff g has.

The advantage of this procedure lies in the fact that $R\odot f$ has many properties different from f and therefore a heuristic simplifier may behave quite differently when simplifying $R\odot f$ rather than f.

From an expression f containing a variable x we can derive six distinct expressions:
$1\odot f$, $x\odot f$, $\bar{x}\odot f$, $\bar{x}\odot x\odot f$, $x\odot\bar{x}\odot f$, $x\odot\bar{x}\odot x\odot f$.
Only the first three are of interest here because the other three can be obtained from the first three by the interchange of x and $\bar{x}$, and hence have the same properties from minimization point of view.

When minimizing a function of n variables using the procedure M, R can be chosen in 3^n different ways. Namely, for each variable x we can chose to put into R the transform 1, x, or $\bar{x}$. (Please note that in this way we ensure that $R^{-1} = R$.)

Many of the 3^n functions have quite different properties, which will make the given simplifier generate different chains of rules. Therefore running only a few of the them through the simplifier is an effective way of exploring the search space.

Given a set of rules one can apply r-transforms to both sides of each rule generating new rules. For examples, applying the y-transform to (S3) we obtain

$$x \oplus \bar{y} \rightarrow y \oplus \bar{x} \qquad (S3*)$$

We call a set of rules "closed under r-transforms" iff every one of the possible r-transforms generates a rule already present in the original set. Since (S3*) is not present in the original set of rules, (S1) - (S6) form an example of rules that are not closed.

For a set of rules that is not closed the procedure M is particularly beneficial because it may generate solutions unobtainable by the the rules themselves.

4. Algorithm

Until now we have discussed methods of improving the asymptotic behavior of rewrite rules. In order to evaluate their impact on short term behavior we have implemented them in a way described in this section.

For our experiments we have used the simplifier EXMIN[SAS 90b], which is known to produce in general very good results, but

not always optimal. It is also known not to be convergent, and it is known to be closed under r-transforms. Our implementation did not change EXMIN, but rather treated it as a "black box" and invoked it repeatedly.

The implementation is based on the following procedure P(f), which takes an expression f as an argument and returns a modified expression g. It divides f into two, calls itself recursively on the two halves and then combines the results together. When called on the top-most level it is supposed to return a minimum. However, the recursive calls are not supposed to return a minimum, not even necessarily a good solution; they are supposed to return expressions which can be eventually combined into a minimum. We do not know how to generate such expressions; therefore the procedure is non-deterministic and guarantees only the possibility of generating a minimum. Repeated invocations of P with different random choices will eventually produce an optimum [BRA 90].

procedure P(f) returns g

1. Possibly let g := f and go to step 6
2. Chose a variable x
3. Let g_1:= P(x f), g_2:= P($\bar{x}$ f), g := $g_1 \oplus g_2$
4. Apply the rule (S1) to "some" pairs of terms, one from g_1, the other from g_2.
5. Apply the rule (S2*) to "some" terms of g
6. Possibly apply EXMIN to g
7. Return g

The procedure P contains words like "possibly" and "some" because it is non-deterministic. All the non-deterministic decisions are done randomly, but the probability distributions are biased towards values that we found experimentally to speed up convergence.
Now we will explain each statement in turn:

1. The procedure calls itself recursively in statement 3 and Statement 1 controls the depth of recursion. For example, we would go to step 6 if f consisted of minterms only. We also have to limit the depth of recursion for practicality. But to guarantee convergence, we allow larger and larger depth every once a while.

2. We chose a variable x randomly, but making sure that neither x f nor $\bar{x}$ f is empty. That is the only requirement for convergence. To speed up convergence we give preference to variables that do split some terms into two, but do not split too many terms.

3. The expression x f implies that each term of f is ANDed with x, which may result in the elimination of some terms and reduction of others. Here is where the rule (S1*) takes place.

4. Apply the rule (S1) only along the chosen variable x, but not whenever possible. To guarantee convergence we must disallow some possible merges because a merge might prevent something better later.

5. The rule (S2*) is applied to some terms of g along the variables x, possibly followed by more applications of (S2*), (S1*) and (S1) to newly generated terms. As mentioned in Section 2 we do not know whether this steps is necessary for convergence in the presence of step 6.

6. EXMIN returns an expression no larger than its input. Therefore there is no harm in using it at the highest level, where we would like P to generate a minimum. However, in the recursive calls EXMIN is not always executed.

The procedure P is used in the following loop to minimize a given expression e:

```
do forever
    f := e  /* e contains the best
               expression so far */
    R := random sequence of r-transforms
    f := R⊙f
    f := P(f)
    if f < e then e := R^-1⊙f
end
```

As will be reported in the next section we ran experiments with various restrictions on term increasing rules. Disallowing the rule (S2*) is achieved by never executing statement 5 of the procedure P.
Disallowing both term-increasing rules (S1*) and (S2*) is accomplished by always executing statement 1 of the procedure P; in addition, in this case we never update e in the last statement of the above loop so that EXMIN always works on a transform of the very original input.

5. Experimental Results

We have run experiments in an effort to answer the following questions:

1) While an increase in the number of terms during simplification is necessary to obtain the optimum, it may not be needed to obtain merely good solutions. It is possible that enlarging the search space by term increasing rules does not add many good solutions on top of those already present.

2) We have mentioned that we do not know whether the rule (S2*) is necessary for convergence. While we cannot answer that question experimentally we can see whether it tends to improve the results.

3) How much of an improvement can one obtain using the techniques described in this paper and at what cost?

We performed experiments on three classes of functions because different functions have different simplification behavior. The results are reported in Tables 1, 2, and 3.

For each function the number under SIZE gives the smallest number of terms found by any of the methods described in this paper. The next three columns indicate improvement achieved by many iterations of the procedure P in comparison with one run of EXMIN. In Tables 1 and 2 there is another set of three columns indicating improvement if the amount of CPU time was allowed to be only twice that of one run of EXMIN.

The columns labeled "2" refer to runs where both rules (S1*) and (S2*) are used.
The columns labeled "1" refer to runs where (S1*) is the only term-increasing rule.
The columns labeled "0" refer to runs where no term-increasing rules are used.

In order to answer question 1) posed at the beginning of this section, we compare the columns labeled "1" and "0" for many iterations of P. While a statistical analysis would be difficult, we can see to what extent are the results consistent with the hypothesis that any differences are due to random variations only. We see that for only one function does column "0" have better improvement than column "1", while column "1" has better improvement more than half of the time. This is not likely to be due to random variations only and indicates that a temporary increase in the number of terms does improve the chances of finding better solutions.

To answer question 2), we compare the columns labeled "2" and "1" for many iterations of P. Here the results are much less conclusive. While column "2" produced smaller expressions more often than column "1", most of the time they were equal. Based on this data we cannot reject the hypothesis that any variations are purely statistical.

An answer to question 3), can be seen directly in Tables 1, 2, 3. The data under DOUBLE CPU TIME is of particular interest, because doubling CPU time is the minimal increase that our methods require, and is probably the maximum we can afford in practice for any larger functions. We see that for small functions the column labeled "0" shows the best improvement after doubling the CPU time. This is because one iteration of this method requires only one run of EXMIN, while several runs are required if we split the function into two. This difference disappears for large functions because the running time of EXMIN increases more than linearly with problem size; thus running EXMIN several times on smaller expressions may actually speed up the process. And in fact, we see that for larger functions columns labeled "1" and "2" tend to give better improvement after doubling CPU time.

6. Conclusions

We have investigated conditions that make rewrite rules convergent, i.e., capable of generating optimal AND-EXOR expressions. We have given one necessary condition for convergence, namely, a temporary increase in the number of product terms.

We have also given a sufficient condition for convergence. Assuming that a given set of rules already contains the rules (S1) and (S6) or their equivalent, which is normally the case, the set of rules can be made convergent by adding (S1*) and (S2*).

We have also considered the possibility of replacing (S2*) by the term nonincreasing rules of EXMIN, but were not able to prove or disprove whether this would result in a convergent set of rules.

We have shown how a given expression can be transformed in various ways so as to explore the search space of a given simplifier.

We have performed experiments in order to answer the three questions stated at the beginning of Section 5. From the experiments we drew these conclusions:
1) Adding the rule (S1*) to EXMIN does give better results in comparison with a more thorough exploration of the search space.
2) Adding the rule (S2*), in addition to (S1*) does not improve the results significantly.
3) Suppose that we wish a better solution without excessive increase in CPU time. Then for smaller functions a more thorough exploration of the search space is the best, while for larger functions adding the rule (S1*) is more advantageous.

Acknowledgements

We thank Mr. Koda for the use of his programs as part of our implementation. This work was supported in part by Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan.

References

[BES 83] Ph. W. Besslich, "Efficient computer method for ExOR logic design", IEE Proc., Vol. 130, Part E, 1983, pp. 203-206.
[BRA 84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli, Logic Minimization algorithms for VLSI Synthesis, Kluwer, 1984.
[BRA 90] D. Brand and T. Sasao, forthcoming publication.
[EVE 67] S. Even, I. Kohavi, A. Paz, "On minimal modulo-2 sum of products for switching functions", IEEE Trans. on Electronic Computers, Vol. EC-16, October 1967, pp. 671-674.
[FLE 87] H. Fleisher, M. Tavel, J. Yeager, "A computer algorithm for minimizing Reed-Muller canonical forms", IEEE trans. on Computers, Vol. C-36, No. 2, February 1987, pp. 247-250.
[FUJ 86] H. Fujiwara, "Logic testing and design for testability", Computer Systems Series, The MIT Press, 1986.
[HEL 88] M. Helliwell, M. Perkowski, "A fast algorithm to minimize multi-output mixed polarity generalized Reed-Muller forms", Proc. of the 25-th Design Automation Conference, June 1988, pp. 427-432.
[HON 74] S.J. Hong, R.G. Cain, D.L. Ostapko, "MINI: A heuristic approach for logic minimization", IBM J. of Research and Development, Vol 18, September 1974, pp. 443-458.
[MUK 70] A. Mukhophadhyay, G. Schmitz, "Minimization of EXCLUSIVE-OR and LOGICAL EQUIVALENCE switching circuits", IEEE Transactions on Computers, Vol. C-19, No. 2, February 1970, pp. 132-140.
[MUL 54] D.E. Muller, "Application of Boolean algebra to switching circuit design and to error detection", IRE Trans. on Electron. Comp., Vol. EC-3, September 1954, pp. 6-12.
[MUR 79] S. Muroga, Logic Design and Switching Theory, John Wiley and Sons, 1979.
[QUI 55] W.V. Quine, "A way to simplify truth functions", American Mathematical Monthly, Vol. 62, November 1955, pp. 627-631.
[PAP 79] G. Papakonstantinou, "Minimization of modulo-2 sum of products", IEEE Trans. on Computers, Vol. C-28, February 1979, pp. 163-167.
[RED 72] S.M. Reddy, "Easily testable realization for logic functions", IEEE Trans. on Computers, Vol. C-21, November 1972, pp. 1183-1188.
[ROB 82] J.P. Robinson, C.L. Yeh, "A method for modulo-2 minimization", IEEE Trans. on Computers, Vol. C-31, August 1982, pp. 800-801.
[SAL 79] K.K. Saluja, E.H. Ong, "Minimization of Reed-Muller canonical expansion", IEEE Trans. on Computers, Vol. C-28, February 1979, pp. 535-537.
[SAS 84] T. Sasao, "Input variable assignment and output phase optimization of PLA's", IEEE Trans. on Computers, Vol. C-33, No. 10, October 1984, pp. 879-894.
[SAS 90a] T. Sasao and P. Besslich, "On the complexity of MOD-2 sum PLA's", IEEE Trans. on Computers, Vol. 32, No. 2, February 1990, pp. 262-266.
[SAS 90b] T. Sasao, "EXMIN: A simplification algorithm for Exclusive-OR-Sum-of-Products expressions for multiple-valued input two-valued output functions", ISMVL-90, May 1990, pp. 128-135
[SAS 90c] T. Sasao, "Exclusive-or sum-of-products expressions: Their properties and minimization algorithms", IEICE Technical Report, VLD 90 December 1990 (to be publishded).

TABLE 1 Functions E(n,k) defined in [SAS 90a]

FUNCTION	SIZE	30 ITERATIONS(%) 2	1	0	DOUBLE CPU TIME(%) 2	1	0
E(6,3)	12	14	14	14	0	0	14
E(6,4)	11	8	8	8	0	0	0
E(7,2)	15	0	0	0	0	0	0
E(7,3)	21	19	19	11	0	0	8
E(7,4)	22	4	4	4	0	0	4
E(7,5)	15	6	6	6	0	0	0
E(8,2)	20	5	5	5	0	0	5
E(8,3)	32	15	13	13	0	0	5
E(8,4)	32	32	32	11	20	17	6
E(8,5)	34	10	8	8	0	0	8
E(8,6)	20	0	0	0	0	0	0
E(9,2)	24	0	11	7	0	0	7
E(9,3)	48	12	12	14	0	0	0
E(9,4)	58	22	19	8	11	14	4
E(9,5)	60	26	22	14	18	21	7
E(9,6)	49	11	9	5	0	2	4
E(9,7)	24	11	11	7	0	0	0
E(10,2)	32	9	9	9	0	0	6
E(10,3)	70	11	10	5	0	0	5
E(10,4)	105	10	15	3	0	5	0
E(10,5)	91	41	41	10	32	27	10
E(10,6)	101	21	25	12	16	16	5
E(10,7)	70	0	4	1	0	0	0
E(10,8)	32	11	11	11	5	8	11

TABLE 2 Arithmetic functions.
Function names ending with "A", e.g. "ADR4A" indicate a representation using multi-valued variables.
For these functions the column labeled "0" is missing because the method described in section 3 does not apply.

FUNCTION	SIZE	50 ITERATIONS(%) 2	1	0	DOUBLE CPU TIME(%) 2	1	0
ADR4	31	6	6	6	3	3	0
ADR4A	11	8	8		0	0	
LOG8	96	2	7	4	0	0	2
LOG8A	94	6	4		0	0	
MLP4	61	8	6	5	1	0	1
MLP4A	52	17	26		0	0	
NRM4	73	3	3	3	0	0	1
NRM4A	58	3	6		0	0	
RDM8	31	0	0	0	0	0	0
RDM8A	27	4	4		4	0	
ROT8	35	5	3	0	0	0	0
ROT8A	28	22	14		0	5	
SQR8	114		6	3		0	0
SQR8A	112	0	0		0	0	
WGT8	54	18	11	0	0	0	0
WGT8A	25	0	0		0	0	

TABLE 3 Random functions.
R(n,k) represents an average of 10 random functions with n variables and k minterms.

FUNCTION	SIZE	40 ITERATIONS(%) 2	1	0
R(6,8)	6.1	0	0	0
R(6,16)	8.6	3	3	3
R(6,24)	9.6	11	8	6
R(6,32)	10.3	10	11	8
R(6,40)	10.6	12	12	7
R(6,48)	9.7	20	20	18
R(6,56)	7.0	17	17	13
R(7,16)	11.2	3	3	3
R(7,32)	15.6	7	7	5
R(7,48)	17.3	11	11	4
R(7,64)	18.7	15	15	8
R(7,80)	19.3	14	15	7
R(7,96)	16.7	27	26	18
R(7,112)	12.3	17	17	16
R(8,32)	20.6	6	5	5
R(8,64)	29.4	7	7	4
R(8,96)	34.3	11	9	6
R(8,128)	35.6	14	15	6
R(8,160)	35.4	17	16	7
R(8,192)	29.8	28	28	7
R(8,224)	21.2	34	34	32