アルゴリズム 19－1
（1991. 1. 29）

共有二分決定グラフ処理の幅優先ベクトルアルゴリズム

越智裕之　石浦菜岐佐　高木直史　矢島脩三
京都大学工学部情報工学教室

共有二分決定グラフは、S. B. Akers および R. E. Bryant によって提案されたグラフによる論理関数の表現法であり、近年論理回路設計支援などの分野で注目されている。本稿では共有二分決定グラフの処理を高速化するべく、ベクトル計算機向きの処理アルゴリズムを提案する。これは、従来深さ優先で行なわれていた共有二分決定グラフの処理を幅優先の処理に置き換えてベクトル化しようというものである。さらに提案した手法をベクトル計算機 HITAC S-820/80 上で実現、評価した結果も示す。

A Breadth-First Vector Algorithm for Manipulating SBDD

Hiroyuki OCHI　Nagisa ISHIURA　Naofumi TAKAGI　Shuzo YAJIMA
Department of Information Science
Faculty of Engineering, Kyoto University
Kyoto 606, Japan

Shared Binary Decision Diagram (SBDD) is a graph representation of Boolean functions proposed by S. B. Akers and R. E. Bryant which is used in various applications of computer-aided design (CAD) of digital systems. We propose a high-speed algorithm of manipulating SBDD, which is suitable for vector super computers. The proposed algorithm is based on, so called, breadth-first manipulation to utilize the high performance of vector super computers, while the conventional algorithms are based on depth-first manipulation. This paper also shows some benchmark results on a vector super computers HITAC S-820/80.

1 Introduction

Boolean function manipulation is one of the most essential operations in various applications of computer-aided design (CAD) of digital systems. Because the efficiency of Boolean function manipulation is closely connected with the representation of Boolean functions, various representations of Boolean functions have been proposed. Shared Binary Decision Diagram (SBDD) is a graph representation of Boolean functions [1][2]. Because of its excellent properties which enable us efficient Boolean manipulation, SBDD is now widely used in various applications such as design verification [3][4], test generation [5], logic synthesis [6] and so on.

At present, SBDD manipulators are, in most cases, implemented on work stations [7][8]. According to the recent progress of the VLSI technology, it is required to manipulate larger and larger scale Boolean functions, which will exceed the computational power of work stations. In order to fulfill this requirement, the use of parallel machines or connection machines is studied [9]. In this paper, we propose an algorithm suitable for vector processors, a kind of super computer with pipelined processors. The proposed algorithm is based on, so called, breadth-first manipulation to utilize the high performance of vector processors, while the conventional algorithms for work stations are based on depth-first manipulation. We implemented and evaluated the proposed algorithm on a vector processor HITAC S-820/80 at the University of Tokyo. The vector acceleration ratio on S-820/80 was 3.8 to 14.3. Our manipulator on S-820/80 was faster than that of Minato et al. on Sun3/60 up to 70 times.

In the following section, basic explanation on SBDD and vector processor are described. In section 3, a new algorithm will be proposed. In section 4, experimental results will be shown. Section 5 provides some concluding remarks.

2 Preliminaries

2.1 SBDD

A *Shared Binary Decision Diagram (SBDD)* is a representation of Boolean functions using an acyclic directed graph. An example of SBDD is shown in Fig. 1(b). The graph represents four Boolean functions corresponding to four *root edges*. Edges other than root edges are labeled either '0' or '1'. They are called *'0' edges* and *'1' edges*, respectively. SBDD is defined as the graph obtained from the binary decision trees (Fig. 1(a)) by repeating the following transformations until they are not applicable.

(a) Share isomorphic sub-graphs.

(b) Delete every node both of whose '0' edge and '1' edge point to the same node.

SBDD's have excellent properties as follows;

(1) If the ordering of the variables is fixed for the whole graph, the graph is canonical, i.e. there is no two root edges of a graph which point to different nodes and yet represent the same Boolean function [1][2].

(2) The size of the graph is feasible for many of the practical Boolean functions [10].

(3) The manipulations for various operations on Boolean functions represented by an SBDD can be done in time proportional to the number of the nodes of the graph [2].

(4) The equivalence of two Boolean functions can be tested by simply comparing the root edges corresponding to the functions.

Once the ordering of the variables is fixed, an integer number called *level* is assigned to all the variables corresponding to the ordering (i.e. the smaller number for the variables nearer to the leaf nodes). We indicate the level by the subscript of the variable such as X_{35}.

We denote the sub-function of a Boolean function f obtained by substituting 0 (1) to variable X as $f(X = 0)$ ($f(X = 1)$), or simply f_0 (f_1) if X is obvious from context. Note that if f depends on variable X (i.e. $f_0 \neq f_1$) and X is the variable of the root node (i.e. the node pointed by the root edge) of f, f_0 and f_1 are represented by the graph with the root edges pointing to the nodes pointed by the '0' edge and the '1' edge, respectively, of the root node of f. On the other hand, if f is independent of X (i.e. $f_0 = f_1 = f$), f_0 and f_1 are represented by the graph with the same root edge as f's itself.

In order to reduce the number of the nodes and/or the time for manipulation of SBDD, various *attributed edges* are proposed, such as output inverters, input inverters, variable shifters, and so on [7]. Among them, *output inverter* is effective to realize high-speed SBDD manipulation, which is the aim of this paper. Output inverter is the attribute indicating to complement the function of the subgraph pointed by the edge (Fig. 2). Using this attribute, we can reduce the size of SBDD's to a half in the best case and can execute *NOT* operation without traversing the graph. Abuse of output inverters break the important property of

SBDD's giving unique representations of Boolean functions. The following limitations are placed in order to keep this property;

(A) Output inverters must not be used in '0' edges, i.e. output inverters are used only in '1' edges or in the root edges.

(B) The leaf node (constant) must be unique, i.e. only 0 can be used, for example, as the leaf node.

2.2 A Conventional Algorithm for Manipulating SBDD

Principal tasks of Boolean function manipulators are

(1) comparison of two Boolean functions,

(2) the unary operation for a Boolean function, i.e. *NOT*,

(3) binary operations for Boolean functions, such as *AND*, *OR*, *EXOR*, and

(4) substitution of 0 or 1 for a variable of a Boolean function.

If the Boolean functions are represented by an SBDD, (1) can be achieved only by comparing two root edges of the given functions, and (2) is also easily realizable if output inverters are employed. (4) can be realized by the method analogous to (3). The rest of this paper is, therefore, devoted to consider the high-speed algorithm for (3).

For example, let us consider the conventional recursive algorithm [7][8] for generating the graph representing Boolean function $h = AND(f, g)$ where f and g are Boolean functions represented by an SBDD. Here we denote the levels of the root nodes of f and g as L_f and L_g, and let $L_h = max(L_f, L_g)$. Recall that the root edges for f_0, f_1, g_0 and g_1 can be immediately obtained.

[Conventional Algorithm for AND]
Examine the given root edges for f and g and execute either of the followings;

(case1) If $f = 0$ or $g = 0$, then return 0.

(case2) If $f = 1$ ($g = 1$), then return the root edge of g (f).

(case3) If $f = g$, then return the root edge for f.

(case4) If $f = NOT(g)$, then return 0.

(case5) Otherwise, compute the root edges for $h(X_{L_h} = 0) = AND(f(X_{L_h} = 0), g(X_{L_h} = 0))$ and $h(X_{L_h} = 1) = AND(f(X_{L_h} = 1), g(X_{L_h} = 1))$ recursively. Then examine the root edges for $h(X_{L_h} = 0)$ and $h(X_{L_h} = 1)$ and execute either of the followings;

(case5.1) If $h(X_{L_h} = 0) = h(X_{L_h} = 1)$, then return the root edge for $h(X_{L_h} = 0)$.

(case5.2) Otherwise, generate a new root node for h whose level is L_h and whose '0' edge and '1' edge point to the root node of $h(X_{L_h} = 0)$ and $h(X_{L_h} = 1)$, respectively.

Before generating a new node in case 5.2 of the above algorithm, one must check whether there exists a node whose level is L_h and '0' edge and '1' edge point to the root node of h_0 and h_1, respectively. If such a node exists, this old node must be used as the root node of h instead of generating a new node. This task is essential to keep SBDD canonical. For this purpose, a hash table called a *node table* is introduced which manages all nodes of the graph. The keys of the node table are the level, '0' edge and '1' edge of a node.

Another hash table called an *operation result table* is introduced to avoid repetitions of the same operations. The keys of the operation result table are a Boolean operator and given two root edges. Every time when case 5 of the above algorithm is completed, the result is registered to this table. One can save the time for executing case 5 if the result is found in this table before executing case 5. This table is essential, rather than merely effective, especially when there are many reconvergences in the subgraphs of the given functions. The simple example is shown in Fig. 3. Let us consider the case of computing $AND(f, g)$. According to the above algorithm, one must obtain $AND(f(X_{35} = 0), g)$ and $AND(f(X_{35} = 1), g)$. To obtain $AND(f(X_{35} = 0), g)$, both $AND(f(X_{35} = 0, X_{33} = 0), g)$ and $AND(f(X_{35} = 0, X_{33} = 1), g)$ are required, while to obtain $AND(f(X_{35} = 1), g)$, both $AND(f(X_{35} = 1, X_{33} = 0), g)$ and $AND(f(X_{35} = 1, X_{33} = 1), g)$ are required. Because $f(X_{35} = 0, X_{33} = 0)$ is equal to $f(X_{35} = 1, X_{33} = 0)$, we can reuse the result of $AND(f(X_{35} = 0, X_{33} = 0), g)$ as the result for $AND(f(X_{35} = 1, X_{33} = 0), g)$ if the operation result table is introduced.

2.3 Vector Processor

A vector processor is a highly pipelined super computer which is primarily used for large scale scientific and engineering computation. It has, in addition to a conventional processing unit (a *scalar unit*), several function pipelines (a *vector unit*) to yield more than GFLOPS (Giga FLoating Operations Per Second) of computation power. In order to support large scale computation, it has a large

main memory unit (usually a hundred mega bytes or more) and powerful load/store pipelines.

In addition, vector processors have many advanced features in order to make it versatile enough to be used in a wide range of applications. For example, the HITAC S-820/80 at the University of Tokyo on which we have developed our SBDD manipulator provides the following vector operations.

(1) Element-wise Vector Operations
The HITAC S-820/80 can handle integer and logical data as well as floating-point data by function pipelines. For example, integer arithmetic operations, bit-wise (32 bits per word) logical operations and logical shift operations are vectorizable.

(2) Conditional Vector Operations
Above operations can be masked by conditions, i.e. operations work only on elements which satisfy a specified condition. For example, the following program can be vectorized by this function.

```
      DO 10 I=1,N
        IF (IM(I).EQ.0) IA(I)=IB(I)+IC(I)
   10 CONTINUE
```

(3) List Vector Access
The HITAC S-820/80 provides indirect memory access (referred to as *list vector access*) as well as contiguous vector access.

```
      DO 20 I=1,N
        IA(I)=IB(IL(I))
   20 CONTINUE
```

For example, this operation can be used for vectorizing the access of grandbrother nodes by brother nodes. Another application of the list vector access is table look-ups.

(4) Compress operations
Compress operation, which constructs new vector IA from vector IB by collecting elements which satisfy a specified condition, can be vectorized. An example program for compress operation is as follows.

```
      K=0
      DO 30 I=1,N
        IF (IM(I).EQ.0) THEN
          K=K+1
          IA(K)=IB(I)
        ENDIF
   30 CONTINUE
```

Discussions in the following section is common to all vector processors which have above four features.

In order to utilize vector processor efficiently, we must tune up the coding schemes and/or modify the basic algorithms so that our programs are suitable for vector processing. The features of the programs required for efficient vector processing are

(1) *high vectorization ratio*, i.e. almost all the operations in the program should be processed by a vector unit.

(2) *long vector length*, i.e. sufficiently many elements should be processed at a time.

3 A Vector Algorithm for manipulating SBDD

As mentioned in the preceding section, the conventional algorithm for manipulating SBDD's is based on a recursive procedure (or depth-first operation), which is not suitable for vector processing. In this section, we propose a breadth-first algorithm for manipulating SBDD's.

The proposed algorithm consists of two parts; an *expansion phase* and a *reduction phase*. In the expansion phase, new nodes sufficient to represent the resultant function are generated in a breadth-first manner from the root node toward leaf nodes. In the reduction phase, the nodes generated in the expansion phase are checked and the redundant nodes and the equivalent nodes are removed in a breadth-first manner from nodes nearby leaf nodes toward the root node. The nodes generated in the expansion phase are called *temporary nodes*, while the nodes which already exists are called *permanent nodes*.

3.1 Expansion Phase

The input for the expansion phase is a triple (op, f, g), where op is a Boolean operator to be executed, and f and g are the root edges for operand Boolean functions. We refer to this triple as a *requirement*. The requirement (op, f, g) requires to compute the root edge for the resultant function of $op(f, g)$. During processing a requirement, new requirements will be generated for computing the operations between subfunctions or subsubfunctions ... of the operand functions. Actually a requirement corresponds to a procedure call in the depth-first algorithm. We introduce a queue called a *requirement queue* to manage these requirements, which makes our procedure breadth-

first. (The procedure would be depth-first if we use a stack instead of the queue.)

For given requirement (op, f, g), a new root node is not always generated. We should not generate a new node if a node representing the result of $op(f,g)$ already exists. For example, if the result of $op(f,g)$ is found trivially (the case 1-4 in the algorithm in section 2.2), or found by looking up the operation result table, we do not generate a new node. In these cases, the judgement can be done immediately from f and g. However, in general, there are cases where one can not tell the existence of the node of the same function as $op(f,g)$ until we construct the whole graph for the subfunctions of $op(f,g)$. In our breadth-first algorithm, we once generate a temporary node in such cases. Whether the temporary node is actually essential or not is examined in the reduction phase.

Following is the procedure of the expansion phase. Initially, the requirement queue is empty, and there is no temporary node.

[Expansion Phase of the Proposed Algorithm]
Put the given requirement (op, f, g) to the requirement queue and repeat the following operations for every requirement in the queue until the queue becomes empty.

(1) If the root node representing the result of $op(f,g)$ is trivially found, then return the edge pointing to the node.

(2) If the root node representing the result of $op(f,g)$ is found in the operation result table, then return the edge found in the table.

(3) Otherwise, generate a new temporary node and return the edge pointing to the temporary node. At the same time, register the edge pointing to the temporary node to the operation result table as the result for $op(f,g)$ and put new requirements (op, $f0$, $g0$) and (op, $f1$, $g1$) to the requirement queue, whose result will be '0' edge and '1' edge, respectively, of this temporary node.

Note that the temporary nodes must be registered to the operation result table in the expansion phase in order to avoid repetitions of the same operations (recall the example of Fig. 3). On the other hand, the registrations to the node table are done in the reduction phase because the node table is not referred in the expansion phase.

Also note that the total number of requirements processed in the above procedure is exactly the same as the number of procedure calls in the depth-first algorithm in section 2.2 and thus there is no serious increase on the computation cost. The only drawback of our algorithm is the increase of the storage required for temporary nodes.

This procedure is suitable for vector processing because of

(1) long vector length. All requirements existing in the queue can be processed at a time.

(2) high vectorization ratio. All of repeated operations are vectorized.

Manipulations according to the requirement queue are easily vectorizable by referring to the queue as a list vector. Above three cases can be exclusively executed using conditional vector operations. New requirements are put to the queue using compress operations. Registrations of the operation result table is also vectorizable.

3.2 Reduction Phase

After the expansion phase is finished, there may be the following temporary nodes;

(1) *Redundant node*: A temporary node both of whose '0' edge and '1' edge points to the same node.

(2) *Equivalent node*: A temporary node whose level, '0' edge and '1' edge are the same as one of the permanent nodes.

The main tasks of the reduction phase are to find redundant nodes and equivalent nodes and to remove them. In our algorithm, above tasks are done in a breadth-first manner from the nodes nearby the leaf nodes toward the root node. In addition, temporary nodes which are neither the redundant nodes nor the equivalent nodes are registered to the node table.

In practice, the removal of the redundant nodes and the equivalent nodes must be done at the end of the reduction phase because there are edges pointing to these nodes. In our algorithm, the redundant nodes and the equivalent nodes are marked as *slave nodes*. Every slave node has a pointer to its *master node* which takes the place of the slave node. When a slave node is pointed to by '0' edges or '1' edges of other nodes, the edges are modified to point to the master node.

The reduction phase is formalized as follows;

[Reduction Phase of the Proposed Algorithm]
Repeat the following operations while there are temporary nodes. For every temporary node both of whose '0' edge and '1' edge are not temporary

nodes (i.e. permanent nodes or leaf nodes), test as follows;

(1) If its '0' edge and '1' edge are the same, mark the node as a slave node whose master node is the node pointed to by its '0' edge.

(2) If there is an equivalent node registered in the node table, mark the temporary node as a slave node whose master node is the node registered in the node table.

(3) Otherwise, register the node to the node table, and change its attribute to permanent from temporary.

This procedure is also suitable for vector processing because all temporary nodes whose '0' edges and '1' edges are not temporary nodes can be processed at a time, and almost all operations are vectorizable.

3.3 Breadth-First manipulation of Output Inverters

As mentioned in section 2.1, various attributed edges are proposed and the output inverters are efficient to reduce the time for manipulation. In the conventional recursive algorithm, the attributes of the edge pointing to a new node can be easily determined using the result of the recursive steps for its sub-functions. In this section, we present a method to compute the output inverters of the edges without using the result of its sub-functions. This enables us to compute the output inverters in every step of the breadth-first operation in the expansion phase.

We denote the state of the output inverter of edge e as $oi(e)$, whose value is 'true' if the output inverter is attached and 'false' otherwise. The procedures to be added to the expansion phase are as follows;

(a) Attach an output inverter to the root edge of $op(f, g)$ iff $op(oi$(the root edge of f),oi(the root edge of g)) is 'true'.

(b) For the requirement (op, f_0, g_0), attach an output inverter to the root edge f_0 (g_0) iff oi(the root edge of f (g)) is 'true'.

(c) For the requirement (op, f_1, g_1), attach an output inverter to the root edge f_1 (g_1) iff oi(the root edge of f (g)) is different from oi('1' edge of the root node of f (g)).

(d) Never attach an output inverter to the '0' edge corresponding to the requirement (op, f_0, g_0).

(e) Attach an output inverter to the '1' edge corresponding to the requirement (op, f_1, g_1) iff $op(oi$(the root edge of f_1), oi(the root edge of g_1)) is different from $op(oi$(the root edge of f_0),oi(the root edge of g_0)).

One can prove the correctness of this method using the fact that oi(the root edge of f) is 'true' iff the value of f is 1 when 0 is substituted to all the variables.

4 Implementation and Evaluation

We implemented the SBDD manipulator based on the algorithm proposed in the preceding section on the vector processor HITAC S-820/80 at the University of Tokyo. The program is coded in FORTRAN77.

In Table 1, benchmark results on S-820/80 are shown. This table shows the required CPU time for constructing the graph representing the Boolean functions of all the nets from a circuit description. The benchmark circuits are chosen from ones in ISCAS'85 [11]. For the ordering of the variables, the dynamic weight assignment method [7] is employed (the computation time for the ordering is not contained in the Table 1). *Vector execution time* (V) is the required CPU time using all the features of S-820/80, while *scalar execution time* (S) is the required CPU time using only the conventional scalar processing unit of the S-820/80. The same source program is used for both scalar executions and vector executions. The *vector acceleration ratio* (S/V) shows how our program is suited for the vector processor. From Table 1, we can see that 3.8 to 14.3 acceleration ratio is gained. Especially, the circuits with large number of nodes and small number of nets, such as c432, c499, c1355 and c3540, are highly accelerated. Compared with the results on the work station Sun3/60 by Minato et al. [7], our results are up to 70 times faster. For example, 21.5 sec. and 51.4 sec. were required for c499 and c1355 respectively in [7], while only 0.305 sec. and 0.750 sec. respectively, are required in Table 1.

The required storage is 7 words (28 bytes) for a permanent node (including the space for the node table and the operation result table) and the additional required temporary storage for a temporary node is 5 words (20 bytes). Since we can use up to 128 mega byte main memory on HITAC S-820/80 at the University of Tokyo, we can manage an SBDD of more than 2 million temporary nodes or more than 4 million permanent nodes.

5 Conclusion

We have proposed a high-speed algorithm for manipulating an SBDD on vector processors, and shown benchmark results of the proposed algorithm on the vector processor HITAC S-820/80 at the University of Tokyo. The vector acceleration ratio on S-820/80 was 3.8 to 14.3. Our manipulator on S-820/80 was faster than that of Minato et al. on Sun3/60 up to 70 times. The developed SBDD manipulator is expected to be used for various applications of CAD systems such as design verification, test generation, logic synthesis and so on.

The future works on the vectorized SBDD manipulator are as follows;

(1) The reduction of the required storage. This includes the implementation of attributed edges other than output inverters.

(2) The improvement of the parallelism of the processing. If we put two or more requirements to the requirement queue at the first step of the expansion phase, two or more operations are processed at a time with longer vector length, which may produce more vector acceleration ratio.

Acknowledgment

The authors would like to express their sincere appreciation to Mr. Shin-ichi Minato for his valuable suggestions and advices. The authors would like to thank all the members of Yajima Laboratory at the Kyoto University for their valuable discussions and comments.

References

[1] S. B. Akers: "Binary Decision Diagrams", IEEE Trans. Comput., vol. C-27, no. 6, pp. 509-516, (June 1978).

[2] R. E. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. Comput., vol. C-35, no. 8, pp. 677-691, (Aug. 1985).

[3] M. Fujita, H. Fujisawa and N. Kawato: "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams", Proc. IEEE ICCAD-88, pp. 2-5, (Nov. 1988).

[4] N. Ishiura, Y. Deguchi and S. Yajima: "Coded Time-Symbolic Simulation Using Shared Binary Decision Diagram", Proc. 27th ACM/IEEE DAC, pp. 130-135, (June 1990).

[5] K. Cho and R. E. Bryant: "Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation", Proc. 26th ACM/IEEE DAC, pp. 418-423, (June 1989).

[6] H. Sato, Y. Yasue, Y. Matsunaga and M. Fujita: "Boolean Resubstitution with Permissible Functions and Binary Decision Diagrams", Proc. 27th ACM/IEEE DAC, pp. 284-289, (June 1990).

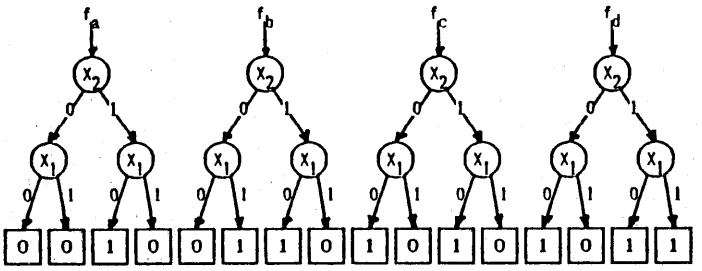
[7] S. Minato, N. Ishiura and S. Yajima: "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", Proc. 27th ACM/IEEE DAC, pp. 52-57, (June 1990).

[8] K. S. Brace, R. L. Rudell and R. E. Bryant: "Efficient Implementation of a BDD Package", Proc. 27th ACM/IEEE DAC, pp. 40-45, (June 1990).

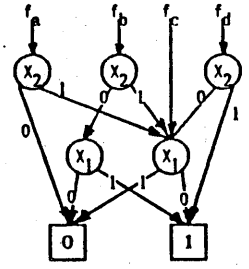
[9] S. Kimura, E. M. Clarke: "A Parallel Algorithm for Constructing Binary Decision Diagrams", Proc. IEEE ICCD'90, (Sep. 1990).

[10] N. Ishiura, T. Tohdo and S. Yajima: "A Class of Logic Functions Expressible by a Polynomial-Size Binary Decision Diagram", Proc. 39th Annual Convention IPS Japan, pp. 1808-1809, (Oct. 1989).

[11] F. Brglez and H. Fujiwara: "A Neutral Netlist of 10 Combinational Circuits", Special Session on ATPG and Fault Simulation, Proc. 1985 IEEE International Symposium on Circuit and Systems, Kyoto, Japan, (June 1985).



(a) Binary Decision Trees



f_a=AND(X_2,NOT(X_1))
f_b=EXOR(X_2,X_1)
f_c=NOT(X_1)
f_d=OR(X_2,NOT(X_1))

(b) A Shared Binary Decision Diagram

Fig.1 Binary Decision Tree and SBDD

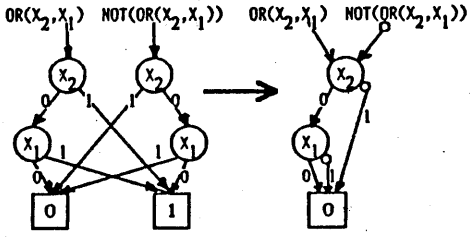


Fig.2 Output Inverters

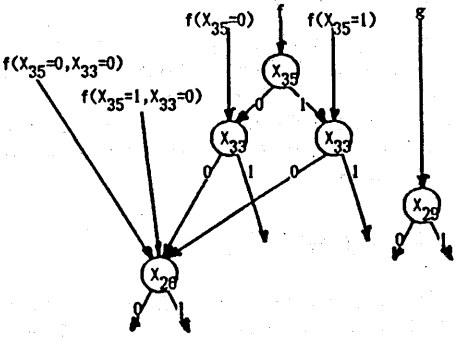


Fig.3 Example on the Effect of the Operation Result Table

Table 1. Experimental results.

Circuit	Circuit size			Number of nodes	CPU time [msec]		Acceleration Ratio
	In.	Out.	Nets.		Scalar (S)	Vector (V)	(S/V)
c432	36	7	203	104,066	8,269	734	11.3
c499	41	32	275	65,671	2,767	305	9.1
c880	60	26	464	31,378	1,919	351	5.5
c1355	41	32	619	208,324	6,210	750	8.3
c1908	33	25	938	60,850	2,894	567	5.1
c3540	50	22	1741	1,029,210	67,841	4,747	14.3
c5315	178	123	2608	48,353	5,843	1,531	3.8