Multiple-Valued Minimization to Optimize PLAs with Output EXOR Gates

Debatosh Debnath and Tsutomu Sasao
Department of Computer Science and Electronics
Kyushu Institute of Technology
Iizuka 820-8502, Japan

Abstract

This paper considers an optimization method of programmable logic arrays (PLAs), which have two-input EXOR gate at the outputs. The PLA realizes an EXOR of two sum-of-products expressions (EX-SOP) for multiple-valued input two-valued output functions. We present techniques to minimize EX-SOPs, which is an extension of Dubrova-Miller-Muzio's AOXMIN algorithm. We conjecture that, when n is sufficiently large, an EX-SOP for n-bit adder requires at most 2^n products while an ordinary sum-of-products expression (SOP) requires $6 \cdot 2^n - 4n - 5$ products. Experimental results for two- and four-valued benchmark functions show that the proposed method produces better EX-SOPs than existing methods.

Index Terms—Three-level network, logic minimization, adder, multiple-valued logic, programmable logic array.

1 Introduction

Programmable logic arrays (PLAs) with two-input EXOR gate at the outputs, also known as AND-OR-EXOR PLAs (Fig. 1), are a powerful architecture to realize many logic functions. The AND-OR-EXOR PLA realizes an EXOR of two sum-of-products expressions (EX-SOP). Minimization of the number of products in EX-SOPs is an important step in the optimization of AND-OR-EXOR PLAs, because the number of products is directly related to the cost of PLAs. EX-SOPs are promising because, for many practical logic functions, they often require many fewer products than sum-of-products expressions (SOPs) [4, 6, 13].

Minimization of EX-SOPs were considered in the past [8, 14] and a cut-and-try method was reported [9]. Design methods for adders by using AND-OR-EXOR PLAs with more than one-bit decoders were developed at IBM [15]. In the last few years significant progress in the minimization of EX-SOPs have been made [4, 6, 13]. Upper bounds on the number of products in EX-SOPs are reported [2, 3, 5]. AND-OR-EXOR network where the output EXOR gate have unlimited fan-in is considered [12].

In this paper, we present a heuristic method to minimize EX-SOPs, which is an extension of AOXMIN [6]. Unlike AOXMIN, we can minimize EX-SOPs for multiple-valued input two-valued output functions. EX-SOPs for functions with two- and four-valued inputs correspond to AND-OR-EXOR PLAs with one- and two-bit decoders, respectively (Fig. 1) [13]. We also present a method to further reduce the number of products in EX-SOPs by considering output phase optimization [11], where some components of the function are implemented in the complemented form.

A crucial step in AOXMIN is to partition the products of an SOP of the given function into two sets, which is done by a random method. We propose a partitioning method for adders. Our experimental result demonstrates that, for an n-bit adder with two-valued inputs and sufficiently large n, the proposed partitioning method is about 250 times faster to produce about two times better solution than the random partitioning method. For adders with two-bit decoders, proposed partitioning method is faster than random partitioning method in producing comparable solutions.

The remainder of the paper is organized as follows: Section 2 reviews terminologies. Section 3 considers output phase optimization techniques. Section 4 summarizes AOXMIN and describes its extensions. Section 5 presents design method for adders. Section 6 shows experimental results and conjectures that, when n is sufficiently large, an EX-SOP for an n-bit adder requires at most 2^n products. Section 7 presents conclusions.

2 Definitions and Terminologies

In this section, we review basic terminologies related to multiple-valued functions [10, 11].

Definition 2.1 *A* ***multiple-valued input two-valued output function****, or* ***function*** *in short, is a mapping*

$$f(X_1, X_2, \ldots, X_n) : \mathop{\times}_{i=1}^{i=n} P_i \to B,$$

where $P_i = \{0, 1, \ldots, p_i - 1\}$, $p_i \geq 2$, $B = \{0, 1\}$, and X_i is a ***multiple-valued variable*** *taking a value from P_i.*

Definition 2.2 *Let $S_i \subseteq P_i$. A* ***literal*** *$X_i^{S_i}$ represents 0 if $X_i \notin S_i$ and 1 if $X_i \in S_i$. A* ***product*** *$X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ is AND of literals. A* ***cube*** *is a convenient representation of a product for computer manipulation.*

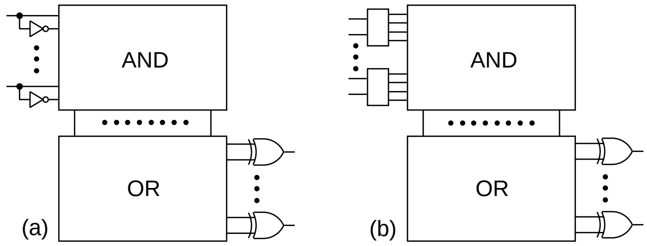

Figure 1: AND-OR-EXOR PLA with (a) one-bit and (b) two-bit decoders.

Definition 2.3 *A* ***sum-of-products expression (SOP)***

$$\bigvee_{(S_1,S_2,\ldots,S_n)} X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$$

is OR of products. An SOP is represented by a ***cover****, which is a set of cubes. An* ***EX-SOP***

$$\bigvee_{(S_1,S_2,\ldots,S_n)} X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n} \oplus \bigvee_{(S_1,S_2,\ldots,S_n)} X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n} \quad (2.1)$$

is the EXOR of two SOPs.

Definition 2.4 *Let* $f_i(X_1,X_2,\ldots,X_n)$ $(i = 0,1,\ldots,m-1)$ *be an n-input m-output function. Then, the two-valued output function* $F(X_1,X_2,\ldots,X_n,X_{n+1})$, *where* X_{n+1} *is an m-valued variable representing the outputs such that* $F(X_1,X_2,\ldots,X_n,i)$ = $f_i(X_1,X_2,\ldots,X_n)$, *is the* ***characteristic function*** *for the multiple-output function [11].*

Definition 2.5 *An* ***SOP for a multiple-output function*** *indicates an SOP for its characteristic function, and an* ***EX-SOP for a multiple-output function*** *indicates an EX-SOP for its characteristic function.*

Definition 2.6 *The* ***intersection*** *of the products* $c_1 = X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ *and* $c_2 = X_1^{T_1} X_2^{T_2} \cdots X_n^{T_n}$, *denoted by* $c_1 \cap c_2$, *is the product* $X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}$. *If* $S_i \cap T_i = \emptyset$ *for some i, then the intersection denotes a null cube.*

Definition 2.7 ***Disjoint sharp*** *of two covers F and G, denoted by* $F \;\#\; G$, *represents only those minterms of F which are not contained by G.*

Definition 2.8 ***ON-set****,* ***OFF-set****, and* ***DC-set*** *is the set of cubes for which the function value is 1, 0, and unspecified, respectively.*

In this paper, we often use the same symbol for a function and its cover; and unless otherwise specified, *adder* refers to *adder without carry input*, and adr$n$ represents an n-bit adder.

3 Output Phase Optimization

In many cases, we can realize a function f in either *positive phase* (f) or *negative phase* ($\bar{f}$). For m-output function, we can choose the output phases in 2^m ways. The choice of the output phases in the realization of a function influences on the number of products in its minimized expressions. To reduce the number of products by choosing the output phases is *output phase optimization* [11].

Definition 3.1 *Let* $(f_0,f_1,\ldots,f_{m-1})$ *be an m-output function. Then, the minimized SOP G for the characteristic function of* $(g_0,g_1,\ldots,g_{m-1})$, *where* $g_i \in \{\bar{f_i},f_i\}$ $(i = 0,1,\ldots,m-1)$ *such that the number of products in G is minimal, is the* ***output phase optimized SOP*** *for* $(f_0,f_1,\ldots,f_{m-1})$.

Similarly, we can define an *output phase optimized EX-SOP*.

We handle the output phase optimization of EX-SOPs by using the output phase optimization of SOPs. We use an output phase optimized SOP as the input of the EX-SOP minimizer. For a function with m outputs, an EX-SOP minimizer produces two SOPs each having m outputs. We optimize the output phases of the $2m$-output SOP to obtain an output phase optimized EX-SOP.

Let the output phase for the function f_i be $a_i \in \{0,1\}$, where $a_i = 0$ indicates f_i is in the positive phase and $a_i = 1$ indicates f_i is in the negative phase. Let the output phases of the two SOPs of the EX-SOP for f_i be b_{i0} and b_{i1}. Therefore, the output phase of the EX-SOP for f_i is $a_i \oplus b_{i0} \oplus b_{i1}$. When output phase optimization of the two m-output SOPs is impractical, we consider a_i as the output phase of the EX-SOP for f_i.

An output phase optimized EX-SOP can be realized in an AND-OR-EXOR PLA, where the polarity of the outputs are programmable.

4 Simplification Techniques

In this section, we review AOXMIN [6], which is a heuristic algorithm to simplify EX-SOPs. We then present an extension of AOXMIN.

4.1 An Overview of AOXMIN [6]

Basic steps of AOXMIN are as follows:

1. Obtain a minimized cover F for the given function f and compute a cover R for $\bar{f}$.
2. Group the cubes of F into clusters of cubes. Two cubes are in the same *cluster* if they intersect or they are connected through a chain of intersecting cubes. (In [6], a cluster of cubes are called an equivalence class.)
3. Randomly partition the cluster of cubes into two covers, F_A and F_B.
4. Obtain two EX-SOPs by using AOXMIN_SPECIFY_BOTH(F_A,F_B,R) and AOXMIN_SPECIFY_BOTH(F_B,F_A,R) (Fig. 2). AOXMIN_SPECIFY_BOTH returns two SOPs which form an EX-SOP. ESPRESSO(F_k,D_k,R_k) in Fig. 2 obtains a minimized cover for a function, where F_k, D_k, and R_k represents the ON-set, DC-set, and OFF-set, respectively.
5. Iterate steps 3 and 4 for some specified number of times, and take the best EX-SOP among all the EX-SOPs generated so far.

In addition, AOXMIN simplifies complement of the given function and uses some output phase optimization technique to obtain better solution.

procedure AOXMIN_SPECIFY_BOTH(F_A,F_B,R) {
  $F_a \leftarrow$ ESPRESSO(F_A,R,F_B);
  $R_{assigned} \leftarrow F_a \, \# \, F_A$;
  $F_{temp} \leftarrow F_B \cup R_{assigned}$;
  $R_{temp} \leftarrow F_A \cup (R \, \# \, R_{assigned})$;
  $F_b \leftarrow$ ESPRESSO(F_{temp},$\emptyset$,R_{temp});
  return (F_a,F_b);
}

Figure 2: Pseudocode AOXMIN_SPECIFY_BOTH.

4.2 An Extension of AOXMIN

The proposed heuristic method to simplify EX-SOPs, which is an extension of AOXMIN [6], have the following features:

- It can simplify EX-SOPs for functions with two- and four-valued variables, and can treat functions where different variables have different values. On the other hand, AOXMIN simplifies only two-valued functions.
- It uses heuristic algorithms to partition the cluster of cubes for adders. In this regard, AOXMIN uses only a random partitioning method.
- During iterative improvement, it concurrently minimizes both SOPs of the EX-SOP to reduce the total number of products by increasing shared products between two SOPs. On the other hand, AOXMIN uses simultaneous minimization of both SOPs only once as part of its simplification technique for multiple-output functions.
- For multiple-output functions, it performs concurrent simplification of all the outputs. However, AOXMIN simplifies each output separately throughout the algorithm. A modified AOXMIN considers simplification of all the outputs simultaneously [7].
- For the output phase optimization of EX-SOPs, it uses techniques for the output phase optimization of SOPs [11]. AOXMIN handles the output phase optimization problem in a different way.
- To find good solutions quickly, especially for adders, it selects from two different minimizers for SOPs. On the other hand, AOXMIN uses only Espresso [1].
- The method makes efficient use of the given don't care conditions during grouping the cover into cluster of cubes and also during every minimization of the SOPs of the EX-SOP. AOXMIN does not use don't care conditions during these two operations.

Theorem 4.1 *An arbitrary multiple-valued input two-valued output function can be represented by an EX-SOP of the form (2.1).*

The minimization of SOP for a multiple-output function corresponds to the minimization of SOP for its characteristic function [11]. Similarly, we can prove the following:

procedure MODIFIED_SPECIFY_BOTH(F_A,F_B,D,R) {
  $F_{AsharpD} \leftarrow F_A \, \# \, D$;
  $F_{BsharpD} \leftarrow F_B \, \# \, D$;

  $F_a \leftarrow$ SIMPLIFY_SINGLE($F_{AsharpD}$,$D \cup R$,$F_{BsharpD}$);
  $R_{assigned} \leftarrow F_a \cap R$;
  $R_{remained} \leftarrow R \, \# \, R_{assigned}$;

  $F_b \leftarrow F_{BsharpD} \cup R_{assigned}$;
  $R_a \leftarrow F_{BsharpD} \cup R_{remained}$;
  $R_b \leftarrow F_{AsharpD} \cup R_{remained}$;

  $F_{dbl} \leftarrow$ MAKE_DOUBLE_OUT_COVER(F_a,F_b);
  $R_{dbl} \leftarrow$ MAKE_DOUBLE_OUT_COVER(R_a,R_b);
  $D_{dbl} \leftarrow$ MAKE_DOUBLE_OUT_COVER(D,D);

  $F_{EX\text{-}SOP} \leftarrow$ SIMPLIFY_DOUBLE(F_{dbl},D_{dbl},R_{dbl});
  return $F_{EX\text{-}SOP}$;
}

Figure 3: Pseudocode MODIFIED_SPECIFY_BOTH.

/* F = ON-set, D = DC-set, R = OFF-set */
procedure SIMPLIFY_LOCAL(F,D,R) {
  $F \leftarrow$ REDUCE(F,D);
  $F \leftarrow$ EXPAND(F,R);
  $F \leftarrow$ IRREDUNDANT(F,D);
  return F;
}

Figure 4: Pseudocode SIMPLIFY_LOCAL.

Theorem 4.2 *The minimization of EX-SOP for a multiple-output function corresponds to the minimization of EX-SOP for its characteristic function.*

Now, the definition of the cluster of cubes can be extended as follows:

Definition 4.1 *Let F and D be the covers for the ON-set and DC-set, respectively, of the characteristic function for a multiple-output function. Then, two cubes $c_i, c_j \in F$ are in the same* ***cluster*** *if*

(a) $G(i,j) \neq \emptyset$, *or*

(b) $G(i,i+1) \neq \emptyset, G(i+1,i+2) \neq \emptyset, \ldots, G(j-1,j) \neq \emptyset$,

where $G(p,q)$ *denotes* $(c_p \cap c_q) \, \# \, D$.

Section 4.1 shows that during every iteration AOXMIN calls AOXMIN_SPECIFY_BOTH twice. We replaced these calls by MODIFIED_SPECIFY_BOTH(F_A,F_B,D,R) and MODIFIED_SPECIFY_BOTH(F_B,F_A,D,R) (Fig. 3). MAKE_DOUBLE_OUT_COVER(F_k,G_k) in Fig. 3 receives n-input m-output covers F_k and G_k, and returns an n-input $2m$-output cover such that covers corresponding to outputs $0,1,\ldots,m-1$ and $m,m+1,\ldots,2m-1$ represent F_k and G_k, respectively.

In Fig. 3, both SIMPLIFY_SINGLE(F_k,D_k,R_k) and SIMPLIFY_DOUBLE(F_k,D_k,R_k) obtain a minimized cover for a function, where F_k, D_k, and R_k represents the ON-set,

adr3:	$1_5, 2_2, 3_2, 5_2$
adr4:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2$
adr5:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2$
adr6:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2$
adr7:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2, 63_2, 95_2$
adr8:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2, 63_2, 95_2, 127_2, 191_2$
adr9:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2, 63_2, 95_2, 127_2, 191_2, 255_2, 383_2$
adr10:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2, 63_2, 95_2, 127_2, 191_2, 255_2, 383_2, 511_2, 767_2$
adr11:	$1_5, 2_2, 3_2, 5_2, 7_2, 11_2, 15_2, 23_2, 31_2, 47_2, 63_2, 95_2, 127_2, 191_2, 255_2, 383_2, 511_2, 767_2, 1023_2, 1535_2$

Figure 5: Distribution of the clusters of output phase optimized SOPs for adders with two-valued inputs.

DC-set, and OFF-set, respectively. SIMPLIFY_SINGLE and SIMPLIFY_DOUBLE can be either SIMPLIFY_LOCAL (Fig. 4) or Espresso-MV [10]. SIMPLIFY_LOCAL uses a single pass of REDUCE, EXPAND, and IRREDUNDANT operations to obtain a simplified SOP [10]. It reduces the number of cubes by locally changing the shape of the cubes. Espresso-MV iterates these operations as long as the solution improves. Sections 5 and 6 explain how the choice of the two-level minimizers affect the quality of the solution and execution time.

5 Design of Adders

In this section, we propose partitioning methods of the cluster of cubes for adders with one- and two-bit decoders, and discuss about the choice of the two-level minimizers. Note that EX-SOPs for functions with two- and four-valued inputs correspond to AND-OR-EXOR PLAs with one- and two-bit decoders, respectively (Fig. 1).

During minimization of adders, we use SIMPLIFY_LOCAL for SIMPLIFY_SINGLE and Espresso-MV for SIMPLIFY_DOUBLE in Fig. 3. We observe that if Espresso-MV is used for SIMPLIFY_SINGLE then the resulting awkward shape of $R_{assigned}$ in Fig. 3 prevent us from obtaining a good solution in the next minimization by using SIMPLIFY_DOUBLE.

5.1 Adders with One-Bit Decoders

We found that output phase optimized SOP for n-bit ($3 \leq n \leq 11$) adder with two-valued inputs have $4n - 1$ cluster of cubes. Fig. 5 shows the distribution of these clusters, where an entry c_k represents k clusters each having c cubes. It is interesting that the number of cubes in the clusters have a regular structure. To partition the cluster of cubes into two covers F_A and F_B, we use the following method:

1. Sort the clusters in descending order of the number of cubes in it.
2. Starting from the beginning of the sorted list of the clusters, alternatively add a pair of clusters to F_A and a pair of clusters to F_B.
3. Add the remaining cluster to F_B.

Example 5.1 *For three-bit adder with two-valued inputs, the number of cubes in the clusters which form F_A and F_B are 5, 5, 2, 2, 1, 1, and 3, 3, 1, 1, 1, respectively.*

adr4:	$1_4, 2_2, 3_2$
adr5:	$1_4, 2_2, 3_2, 4_2$
adr6:	$1_4, 2_2, 3_2, 4_2, 5_2$
adr7:	$1_4, 2_2, 3_2, 4_2, 5_2, 6_2$
adr8:	$1_4, 2_2, 3_2, 4_2, 5_2, 6_2, 7_2$
adr9:	$1_4, 2_2, 3_2, 4_2, 5_2, 6_2, 7_2, 8_2$
adr10:	$1_4, 2_2, 3_2, 4_2, 5_2, 6_2, 7_2, 8_2, 9_2$
adr11:	$1_4, 2_2, 3_2, 4_2, 5_2, 6_2, 7_2, 8_2, 9_2, 10_2$

Figure 6: Distribution of the clusters of output phase optimized SOPs for adders with two-bit decoders.

The above partitioning method is devised by considering outputs. Adders have pairs of clusters, where each pair belongs to a particular set of outputs. Roughly, the strategy is to put the clusters from such pair into two different partitions. A similar method is also devised for adders with four-valued inputs.

5.2 Adders with Two-Bit Decoders

We obtained functions with four-valued inputs from their two-valued counterparts by pairing two variables using Espresso-MV [10]. Fig. 6 shows the distribution of the clusters of output phase optimized SOPs for adders with two-bit decoders, where an entry c_k represents k clusters each having c cubes. It shows that the output phase optimized SOP for n-bit ($4 \leq n \leq 11$) adder with two-bit decoders have $2n$ clusters. Note that the number of cubes in the clusters for adders with two-bit decoders also have a regular structure. We use the following method to partition the clusters into two covers F_A and F_B:

1. Sort the clusters in descending order of the number of cubes in it.
2. Starting from the beginning of the sorted list of the clusters, at first add a pair of clusters to F_A, then alternatively add a cluster to F_A and F_B.

6 Experimental Results

We implemented the proposed heuristic method to simplify EX-SOPs, which is an extension of AOXMIN [6], in C by using Espresso-MV [10] routines. On an HP C160 workstation with 256 megabytes memory resources, we conducted

Table 1: Experimental result for adders with two-valued inputs.

Data	In	Out	SOP	OPO SOP	Proposed Partition EX-SOP	Proposed Partition Time	Proposed Partition OPO EX-SOP	Random Partition 20 Iterations EX-SOP	Random Partition 20 Iterations Time	Random Partition 50 Iterations EX-SOP	Random Partition 50 Iterations Time
adr3	6	4	31	25	12	0.02	11	17	1.24	13	2.95
adr4	8	5	75	61	21	0.07	18	32	4.48	32	13.46
adr5	10	6	167	137	37	0.35	36	50	26.04	50	61.98
adr6	12	7	355	293	67	1.45	66	146	114.38	133	332.95
adr7	14	8	735	609	122	4.56	120	128	422.94	128	1033.24
adr8	16	9	1499	1245	233	19.58	MEM	423	1634.73	380	3972.45
adr9	18	10	3031	2521	454	66.42	MEM	840	6469.34	840	16492.60
adr10	20	11	6099	5077	967	312.17	MEM	2168	28767.08	1898	74434.15
adr11	22	12	12239	10193	1993	1596.42	MEM	4136	169809.86	3677	425304.35

OPO: Output phase optimized. MEM: Espresso-MV memory over.

Table 2: Experimental result for adders with two-bit decoders.

Data	SOP	OPO SOP	Proposed Partition EX-SOP	Proposed Partition Time	Proposed Partition OPO EX-SOP	Random Partition 20 Iterations EX-SOP	Random Partition 20 Iterations Time	Random Partition 50 Iterations EX-SOP	Random Partition 50 Iterations Time
adr4	17	14	13	0.09	12	13	1.64	13	4.10
adr5	26	22	18	0.34	18	18	5.81	18	15.64
adr6	37	32	25	0.81	25	25	18.39	25	51.90
adr7	50	44	33	1.94	33	33	58.91	33	136.55
adr8	65	58	42	6.62	42	43	181.41	43	429.76
adr9	82	74	52	26.11	52	51	655.76	51	1523.54
adr10	101	92	63	74.46	63	65	2433.61	61	5911.23
adr11	122	112	75	353.35	75	75	7918.99	75	23507.19

OPO: Output phase optimized.

experiments by using adders with two- and four-valued inputs and other benchmark functions with four-valued inputs. We obtained functions with four-valued inputs from their two-valued counterparts by pairing two variables using Espresso-MV. For all the experiments, we prepared minimized SOPs and output phase optimized SOPs by using Espresso-MV with default options.[†]

Tables 1 and 2 summarize the experimental data for adders with one- and two-bit decoders, respectively. To minimize EX-SOPs for adders we used: a) output phase optimized SOPs as the input for the EX-SOP minimizer; b) two different techniques to partition the cluster of cubes: partitioning method for adders from Section 5 and random partitioning method from AOXMIN [6]; and c) SIMPLIFY_LOCAL for SIMPLIFY_SINGLE and Espresso-MV for SIMPLIFY_DOUBLE in Fig. 3.

Table 1 shows that, for an n-bit adder with sufficiently large n, the proposed partitioning method is about 250 times faster to produce about two times better solution than the random partitioning method. Note that an SOP and an output phase optimized SOP for n-bit adder with two-valued inputs require $6 \cdot 2^n - 4n - 5$ and $5 \cdot 2^n - 4n - 3$ products, respectively [11]. However, from Table 1, we have the following:

[†]For all the tables in this section, the columns with heading 'SOP', 'OPO SOP', 'EX-SOP', and 'OPO EX-SOP' indicate the number of products in the corresponding expression, where 'OPO' is an abbreviation for 'output phase optimized'. Also, the columns with heading 'Time' indicate the CPU seconds spent by the extended version of AOXMIN to simplify EX-SOP and it does not include the time to prepare minimized SOP or output phase optimized SOP.

Conjecture 6.1 *When n is sufficiently large, an output phase optimized EX-SOP for n-bit adder with two-valued inputs requires at most* 2^n *products.*

The above shows that an output phase optimized EX-SOP requires about one sixth of the products in an SOP. This result would be useful to design adders.

Table 2 shows that the proposed partitioning method produced good solutions quickly. However, in most cases, these solutions can be obtained by random partitioning method by a reasonable increase in the computation time. The experimental data also reveals that the minimization time for EX-SOPs with four-valued inputs is much smaller than that for the corresponding EX-SOPs with two-valued inputs, because the former requires many fewer products than the later. Note that an EX-SOP for an n-bit adder with two-bit decoders requires at most $(n^2+n+2)/2$ products [13].

Table 3 presents experimental results for benchmark circuits with four-valued inputs. We used both SOPs and output phase optimized SOPs as the input for the EX-SOP minimizer; and Espresso-MV for both SIMPLIFY_SINGLE and SIMPLIFY_DOUBLE in Fig. 3.

We used adr6 to see how the choice of the two-level minimizers in Fig. 3 affect the quality of the solution and execution time. By using random partitions and 1000 iterations, we found that when Espresso-MV is used for both SIMPLIFY_SINGLE and SIMPLIFY_DOUBLE the algorithm requires 6253.79 seconds and produces a solution with 122 products; however, when we use SIMPLIFY_LOCAL for

Table 3: Experimental result for EX-SOPs with four-valued inputs.

	Input is SOP					Input is OPO SOP				
		20 Iterations		50 Iterations			20 Iterations		50 Iterations	
Data	SOP	EX-SOP	Time	EX-SOP	Time	OPO SOP	EX-SOP	Time	EX-SOP	Time
5xp1	46	35	19.31	35	51.15	42	29	14.89	29	39.56
addm4	109	97	208.03	95	519.51	101	89	168.09	89	404.82
f51m	51	37	17.86	35	47.13	50	40	18.16	37	46.20
life	26	20	4.72	20	11.86	26	20	4.74	20	12.43
rd84	54	35	22.81	35	57.72	37	31	37.30	31	92.09
rdm8	51	37	17.85	35	44.19	50	40	18.17	37	60.20
sqr8	157	152	274.39	147	674.02	148	139	212.63	139	577.24

OPO: Output phase optimized.

SIMPLIFY_SINGLE and Espresso-MV for SIMPLIFY_DOUBLE, the algorithm produces a solution with 81 products and requires 5956.69 seconds. We found similar tendencies for other adders too. However, it is our experience that, for many other benchmark functions, Espresso-MV for both SIMPLIFY_SINGLE and SIMPLIFY_DOUBLE is often a good choice.

7 Conclusions and Comments

EX-SOPs are promising because they often require many fewer products than SOPs. We demonstrated that, when n is sufficiently large, an n-bit adder with two-valued inputs requires at most 2^n products in an output phase optimized EX-SOP, while an output phase optimized SOP requires $5 \cdot 2^n - 4n - 3$ products. We presented partitioning method, which is very effective to optimize EX-SOPs for adders. Our experimental result shows that random partitioning method is unsuitable to design adders when n is large, because it requires excessive amount of CPU time to obtain a moderate design. For adders with two-bit decoders, proposed partitioning method is faster than random partitioning method in producing comparable solutions. We found that the choice of the two-level minimizers in AOXMIN-like-algorithm have a great influence on the number of products in EX-SOPs and a powerful minimizer is not always a good choice.

We obtained functions with four-valued inputs from their two-valued counterparts by pairing two variables using Espresso-MV code [10], which reduces the number of products in SOPs [11]. A different pairing algorithm targeting EX-SOPs may lead to better solutions. Currently, we are studying minimization of EX-SOPs for adders with carry inputs.

Acknowledgement

This work was supported in part by a Postdoctoral Fellowship of the Japan Society for the Promotion of Science and in part by a Grant-in-Aid for the Scientific Research of the Ministry of Education, Science, Culture, and Sports of Japan. We thank Prof. D. M. Miller and Dr. E. V. Dubrova for their valuable discussions on AOXMIN.

References

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

[2] D. Debnath and T. Sasao, “An optimization of AND-OR-EXOR three-level networks,” *Proc. Asia and South Pacific Design Automation Conference*, pp. 545–550, Jan. 1997.

[3] D. Debnath and T. Sasao, “Exclusive-OR of two sum-of-products expressions: Simplification and an upper bound on the number of products,” *Proc. 3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Oxford, U.K., pp. 45–60, Sept. 1997.

[4] D. Debnath and T. Sasao, “A heuristic algorithm to design AND-OR-EXOR three-level networks,” *Proc. Asia and South Pacific Design Automation Conference*, pp. 69–74, Feb. 1998.

[5] E. V. Dubrova, D. M. Miller, and J. C. Muzio, “Upper bounds on the number of products in AND-OR-XOR expansion of logic functions,” *Electronics Letters*, Vol. 31, No. 7, pp. 541–542, Mar. 1995.

[6] E. V. Dubrova, D. M. Miller, and J. C. Muzio, “AOXMIN: A three-level heuristic AND-OR-XOR minimizer for Boolean functions,” *Proc. 3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Oxford, U.K., pp. 209–218, Sept. 1997.

[7] E. V. Dubrova and D. M. Miller, “Some experimental result of modified AOXMIN,” Personal communication, May 1998.

[8] H. Fleisher, J. Giraldi, D. B. Martin, R. L. Phoenix, and M. A. Tavel, “Simulated annealing as a tool for logic optimization in a CAD environment,” *Proc. International Conference on Computer-Aided Design*, pp. 203–205, Nov. 1985.

[9] D. Pellerin and M. Holley, *Practical Design Using Programmable Logic*, Prentice Hall, 1991.

[10] R. L. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-6, No. 5, pp. 727–750, Sept. 1987.

[11] T. Sasao, “Input variable assignment and output phase optimization of PLA’s,” *IEEE Trans. Comput.*, Vol. C-33, No. 10, pp. 879–894, Oct. 1984.

[12] T. Sasao, “Logic synthesis with EXOR gates,” in T. Sasao, ed., *Logic Synthesis and Optimization*, Kluwer Academic Publishers, 1993.

[13] T. Sasao, “A design method for AND-OR-EXOR three-level networks,” *Proc. International Workshop on Logic Synthesis*, Lake Tahoe, California, pp. 8:11–8:20, May 1995.

[14] K. Shu, H. Yasuura, and S. Yajima, “Optimization of PLDs with output parity gates,” *National Convention, Information Processing Society of Japan*, Mar. 1985 (in Japanese).

[15] A. Weinberger, “High-speed programmable logic array adders,” *IBM Journal of Research and Development*, Vol. 23, No. 2, pp. 163–178, Mar. 1979.