An Algorithm for Bi-Decomposition of Logic Functions

Bernd Steinbach
Freiberg University of Mining and Technology
Institute of Computer Science
D-09596 Freiberg, Germany
steinb@informatik.tu-freiberg.de

Alan Mishchenko, Marek Perkowski
Portland State University
Department of Electrical and Computer Engineering
Portland, OR 97207, USA
[alanmi,mperkows]@ee.pdx.edu

Abstract

We propose a new BDD-based method for bi-decomposition of multi-output incompletely specified logic functions into a netlist of two-input logic gates. Our algorithm makes use of the internal don't cares generated in the process of decomposition to produce compact well-balanced netlists with short delay. The resulting netlists are provably non-redundant and facilitate test pattern generation. Experimental results over MCNC benchmarks show that our approach outperforms SIS and other BDD-based decomposition methods in the quality of final results with comparable CPU time.

1 Introduction

Decomposition of boolean functions consists in breaking down large logic blocks into smaller ones while keeping the network functionality unchanged. Decomposition plays an important role in logic synthesis. Research in this area has started in 1950 [1,2]. Almost every year new techniques and improvements of the old ones are proposed. The recent years have been marked by the reviving of interest to disjoint decompositions of logic functions [3,4,5].

Decomposition methods can be classified according to the following criteria:

- Each block of the resulting network has single binary output, or may have multiple binary outputs (Ashenhurst and Curtis decomposition, respectively).
- Blocks have non-overlapping, or overlapping supports (the former is known as *disjoint decomposition*)[1].
- Each block has two or less inputs, or the number of inputs may be larger than two (the former is known as *bi-decomposition*).
- Decomposition is performed as technology mapping for FPGAs (in this case, the blocks have the upper limit on the number of inputs, typically, five), as a technology-independent transformation of logic circuits, or as a specialized mapping technique.
- The decomposed structure is derived by breaking down the large initial block, or the decomposition structure is assembled by iteratively adding small components until the network is funtionally equivalent to the initial specification.
- BDDs are used in the decomposition algorithm, or not.
  - If yes, BDDs are used to represent function and store intermediate results, or BDDs are used as the essential data structure directing the decomposition process itself.
- Decomposion shares blocks between multiple-outputs (if specified) or decomposes each output independently.
- Decomposition algorithm allows for incompletely specified functions, or not.

This classification can be extended using other criteria such as methods for variable partitioning, methods for deriving the decomposed functions, cost functions used to evaluate the results of decomposition, etc.

In terms of the above classification, the algorithm proposed in this paper is characterized as follows:

- Each decomposed block has single binary output
- Blocks may have overlapping supports
- Each block has two or less inputs (bi-decomposition)
- It is a technology-independent transformation
- Larger components are broken down into smaller ones
- BDDs are used in decomposing to store functions
- The decomposed blocks are shared between outputs
- Incompletely specified functions are welcome

According to the above classification, there are other approaches similar to ours in a number of ways. None of them, however, has exactly the same list of characteristics. The closest match are previous versions of the our algorithm [6,7,8,9] and the recent BDD-based approach [10,11]. As evidenced by our experiments, the algorithm presented in this paper significantly outperforms its previous versions in both its speed and the quality of final

[1] Some authors [3,4] use the term *disjunctive decomposition* instead of *disjoint decomposition*. In our opinion, this is misleading. The term *disjunctive* is suggestive of "disjunction" in the sense of the boolean OR. Therefore *disjunctive decomposition* may be mistaken for decomposition using OR-gates.

results. A more detailed analysis of the differences of our approach from [10,11] is given in Section 8 of the paper.

The rest of the paper is organized as follows: Section 2 introduces the notations and decomposition models used in the paper. Section 3 introduces necessary and sufficient conditions for bi-decomposition of the given type (AND, OR, EXOR) to exist. Section 4 gives the formulas for deriving the decomposed functions. Section 5 discusses the variable grouping strategy. Section 6 presents efficient techniques to simplify the resulting netlist by using the decomposed functions more than once. Section 7 gives the pseudo-code and discusses the general outline of decomposition algorithm. Section 8 presents experimental results. Section 9 summarizes the paper.

2 Preliminaries

Let f: $B^n \rightarrow B$, $B \in \{0,1\}$, be a completely specified boolean function (CSF). The varibale set X, on which f depends, in called *support* of f. Support size is denoted |X|. Let F: $B^n \rightarrow \{0,1,-\}$ be an incompletely specified boolean function (ISF) given by its on-set (Q) and its off-set (R). It is easy to convert the on-set/off-set representation of an ISF into the interval specifying the set of permissible CSFs: $(Q, \overline{R})$. A CSF f is said to be compatible with (or belong to) the ISF $F = (Q, \overline{R})$, iff $Q \leq f \leq \overline{R}$.

This paper considers decomposition of ISFs into netlists of two-input gates, or bi-decomposition [12]. One-step bi-decomposition is schematically represented in Fig. 1. Symbol $\otimes$ stands for AND, OR, or EXOR gate, while components A and B are arbitrary ISFs. The support X of the initial function is divided into three parts: variables X_A that feed only into block A, variables X_B that feed only into block B, and the common variables X_C. By definition, sets X_A, X_B, and X_C are disjoint. If X_A or X_B is empty, the resulting bi-decomposition is called *weak*. Otherwise it is a *strong* (or non-weak) bi-decomposition. In this paper, we consider both types of bi-decomposition and use the term "bi-decomposition" to denote strong bi-decomposition.

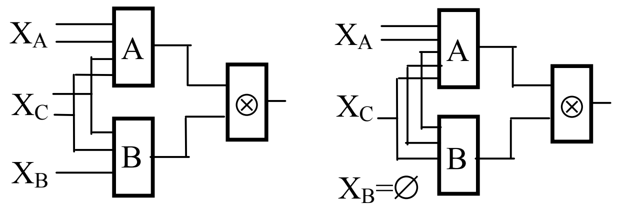

Fig. 1. Schematic representation of two types of bi-decompositoin: strong (left) and weak (right)

Notice that the function in Fig. 1 (right) has one component with five inputs before weak bi-decomposition and two components after the decomposition, one of them with five inputs. Weak bi-decomposition may seem to give no advantage, only increase the complexity of the network. The advantage, however, consists in increasing the number of don't-cares of component A. Before the decomposition the function may be not decomposable in the strong sense, while after the weak bi-decomposition both created blocks may have strong bi-decomposition.

This paper assumes the reader's familiarity with Binary Decision Diagrams (BDDs) [13]. The functions and their supports are represented by BDDs. Two BDD operators, known as *existential* and *universal* quantification, are used extensively in the formulas of this paper. Quantifications of a CSF w.r.t. a variable x is defined as follows: $\exists_x f = f_{\bar{x}} + f_x$ (existential) and $\forall_x f = f_{\bar{x}} \& f_x$, (universal). Symbols "+" and "&" stand for boolean OR and AND, while $f_{\bar{x}}$ and f_x are the cofactors of f: $f_{\bar{x}} = f|_{x=0}$, $f_x = f|_{x=1}$ [13]. Quantification over a set of variables in a straight-forward generalization of the above definition.

Speaking informally, if a CSF is represented by its Karnaugh map (Fig.2), existential (universal) quantification w.r.t. the column-encoding variables is a function, whose Karnaugh map is the sum (product) of columns.

$\exists_{ab}F$

cd	
00	1
01	1
11	0
10	1

OR $\Leftarrow$

F(a,b,c,d)

cd\ab	00	01	11	10
00	1	0	1	0
01	0	0	1	1
11	0	0	0	0
10	1	1	1	1

AND $\Rightarrow$

$\forall_{ab}F$

cd	
00	0
01	0
11	0
10	1

Fig. 2. Karnaugh map illustration of quantifications.

3 Checking decomposability

3.1 Bi-decomposition with OR-gate

Consider a four-input CSF in Fig. 3 (left). This function is bi-decomposable using OR-gate with $X_A = \{c,d\}$ and $X_B = \{a,b\}$. The result of bi-decomposition is:

$$F = OR(\, a \oplus b,\ c\bar{d}\,)$$

cd\ab	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	1	1	1	1

cd\ab	00	01	11	10
00	-	1	0	1
01	0	1	-	-
11	0	-	0	1
10	1	-	1	1

Fig. 3. Examples of OR bi-decomposition.

From the Karnaugh map in Fig. 3 (left) it follows that, for the function to be OR-bi-decomposable, it should have all 1's grouped in a subset of columns and a subset of rows in such a way that none of the columns and rows contain 0's. The requirement does not change for functions with don't-cares, as witnessed by an ISF in Fig. 3 (right), which is OR-bi-decomposable using the same formula.

The above observation motivates the following statement:

Property: F(X) is *not* OR-bi-decomposable with variable sets (X_a,X_b,X_c) iff in the Karnaugh map there exists a cell containing 1 such that 0's appear in *both* the row and the column to which this cell belongs.

For the lack of space, the proof of this and the following statements is left to the reader.

Bi-decomposability of ISFs can be checked by applying the existential quantification to Q and R, representing the on-set and the off-set, because the existential quantification over variables representing columns (rows) consists in adding all the 1's contained in the rows (columns).

Theorem 1: Function F(X) = { Q(X), R(X) } is OR-bi-decomposable with variable sets (X_A, X_B, X_C) iff

$$Q \,\&\, \exists x_A R \,\&\, \exists x_B R = 0.$$

Due to the duality of AND and OR operations, the formula for checking AND-bi-decomposition can be derived by replacing on-set (Q) by off-set (R) in the formulas. For this reason, the rest of the paper considers only OR and EXOR bi-decomposition.

Formulas for checking OR-bi-decomposability of strong and weak types are summarized in Table 1.

3.2 Bi-decomposition with EXOR-gate

Because checking for EXOR-bi-decomposition is rather complicated, the following theorem is formulated for a simpler case of X_A and X_B consisting of one variable only.

Theorem 2: Function F(X) = { Q(X), R(X) } is EXOR-bi-decomposable with variable sets (X_A, X_B, X_C), such that $|X_A| = |X_B| = 1$ iff

$$Q_D \,\&\, \exists x_B R_D = 0,$$

where Q_D and R_D are the on-set and off-set of the derivation of F w.r.t. the variable in X_A:

$$Q_D = \exists x_A Q \,\&\, \exists x_A R, \quad R_D = \forall x_A Q + \forall x_A R.$$

Checking for EXOR-bi-decomposition with arbitrary (non-overlapping) variable sets X_A and X_B is performed using a specialized algorithm in Fig. 4. The procedure CheckExorBiDecomp() takes four arguments. The first two are the on-set (Q) and the off-set (R) of an ISF. The next two are variable sets X_A and X_B. If the EXOR-bi-decomposition exists, the procedure returns the on-sets and off-sets of ISFs implementing components A and B, otherwise it returns zero BDDs. A detailed discussion of this algorithm can be found in [9].

```
procedure CheckExorBiDecomp( bdd Q, bdd R, bdd XA, bdd XB)
{   bdd $Q_A$ = 0, $R_A$ = 0, $Q_B$ = 0, $R_B$ = 0;
    bdd $q_A$ = 0, $r_A$ = 0, $q_B$ = 0, $r_B$ = 0;
    while ( Q != 0 )  {
        bdd Cube = SelectOneCube( Q );
        $q_A = q_A + \exists_B$Cube;
        while ( $q_A + r_A$ != 0 )   {
            $q_B = \exists_A ( Q \& r_A + R \& q_A )$;  $r_B = \exists_A ( Q \& q_A + R \& r_A )$;
            if ( $q_A \& r_A$ != 0 )  return (0, 0, 0, 0);
            Q  = Q - ( $q_A + r_A$ );  R  = R - ( $q_A + r_A$ );
            $Q_A = Q_A + q_A$;        $R_A = R_A + r_A$;
            $q_A = \exists_B ( Q \& r_B + R \& q_B )$;  $r_A = \exists_B ( Q \& q_B + R \& r_B )$;
            if ( $q_B \& r_B$ != 0 )  return (0, 0, 0, 0);
            Q  = Q - ( $q_B + r_B$ );  R  = R - ( $q_B + r_B$ );
            $Q_B = Q_B + q_B$;        $R_B = R_B + r_B$;
        }
    }
    if ( R != 0 ) { $R_A = R_A + \exists_B R$;  $R_B = R_B + \exists_A R$;  }
    return ($Q_A$, $R_A$, $Q_B$, $R_B$);
}
```

Fig. 4. Algorithm for checking the existence of EXOR-bi-decomposition with arbitrary sets X_A and X_B.

4 Deriving decomposed fu nctions

This section presents formulas for deriving ISFs implementing components A and B (see Fig. 1). The case of AND-bi-decomposition is not considered due to its duality with OR. The case of EXOR-bi-decomposition has been adressed in the previous section, because checking for the existence of EXOR-bi-decomposition derived the ISFs of the decomposed functions as a by-product.

Theorem 3: Let F(X) = { Q(X), R(X) } be OR-bi-decomposable with variable sets (X_A, X_B, X_C). The ISF F_A = { Q_A(X), R_A(X) } of the component A is:

$$Q_A = \exists x_B(R \,\&\, \exists x_A R),\ R_A = \exists x_B R.$$

Theorem 4: Let F(X) = { Q(X), R(X) } be OR-bi-decomposable with variable sets (X_A, X_B, X_C) and a CSFs f_A belonging to the ISF F_A is selected to represent component A. The ISF { Q_B(X), R_B(X) } representing component B is:

$$Q_B = \exists x_A(Q - f_A),\ R_B = \exists x_A R.$$

Formulas to derive the ISFs representing components A and B are summerized in Table 1. Symbol "-" in the formulas stands for boolean SHARP ($A\text{-}B = A \,\&\, \overline{B}$).

Type	Checking	Deriving A	Deriving B
OR	$Q\&(\exists x_B R)\&(\exists x_A R) = 0$	$Q_A = \exists x_B(R\&\exists x_A R)$ $R_A = \exists x_B R$	$Q_B = \exists x_A(Q - f_A)$ $R_B = \exists x_A R$
Weak OR	$Q - \exists x_A R \neq 0$	$Q_A = Q \,\&\, \exists x_A R$ $R_A = R$	$Q_B = \exists x_A(Q - f_A)$ $R_B = \exists x_A R$

Table 1. Checking OR-bi-decomposability and deriving ISFs for components A and B.

5 Variable Grouping

An important task arising during the bi-decomposition consists in finding variable sets X_A and X_B, for which the given type of bi-decomposition is feasible. This task is solved in two steps. On the first step, varible sets X_A and X_B are initialized to include one variable. On the second step, attempts are made to add new variables to the sets while preserving the set sizes as close to being equal as possible.

```
procedure FindInitialGrouping( bdd Q, bdd R, bdd S )
{   for all $x \in$ S   {
        $X_A$ = {x};
        for all $y \in$ S – {x}   {
            $X_B$ = {y};
            if ( CheckDecomposability( Q, R, $X_A$, $X_B$ ) )
                return ( $X_A$, $X_B$ );
        }
    }
    return (∅, ∅);
}
```

Fig. 5. Algorithm to find the initial grouping of variables in sets X_A and X_B.

Consider procedure FindInitialGrouping() implementing the first step of variable grouping (Fig. 5). It takes three arguments: the on-set Q, the off-set R, and the support S of an ISF. It returns two singleton sets, X_A and X_B, if the function is strongly bi-decomposable with these sets, or two empty sets, if the function is not bi-decomposable in the strong sense under any variable grouping. The internally called procedure CheckDecomposability() performs OR-, AND-, or EXOR-bi-decomposability check, as discussed in Section 3, depending on what initial grouping is sought.

```
procedure GroupVariables( bdd Q, bdd R, bdd S )
{  ( $X_A$, $X_B$ ) = FindInitialGrouping( Q, R, S );
   if ( ($X_A$, $X_B$) == (∅, ∅) )  return (∅, ∅);
   for all $z \in$ S – ($X_A \cup X_B$)
      if ( $|X_A| \le |X_B|$ )
         // try adding new variable z, first to $X_A$, next to $X_B$
         if ( CheckDecomposability( Q, R, $X_A \cup$ {z}, $X_B$ ) )
            $X_A = X_A \cup$ {z};
         else if ( CheckDecomposability( Q, R, $X_A$, $X_B \cup$ {z} ) )
            $X_B = X_B \cup$ {z};
      else … // similarly if ( $|X_A| > |X_B|$ )
   return ( $X_A$, $X_B$ );
}
```

Fig. 6. Procedure to find the variable grouping.

Procedure GroupVariables() (Fig. 6) implements the second step. The arguments and the return values are the same as in procedure FindInitialGrouping(). Having found a non-empty initial grouping, GroupVariables() considers the remaining support variables one by one and tries to add them to sets X_A and X_B. Depending on sizes of X_A and X_B, it tries to add the new variable to the smaller set first. The rationale for first considering the smaller set is to bring the variable set sizes as close to being equal as possible.

Notice that the above greedy way of building X_A and X_B does not quarantee that they will be the largest possible (meaning that the support sizes of components A and B will be minimum). In practice, however, it gives reasonably good good trade-off between the size and quality of the resulting variable sets, on the one hand, and the computation time needed to evaluate the quantified formulas on each step of variable grouping.

We tried a number of ways to increase the sizes of X_A and X_B, for example, by excluding one variable at a time while trying to add others, and accepting the change only if excluding one variable led to the addition of two or more. This strategy improved the netlist area to an insignificant degree (less than 3%) but the CPU time grew up to 100%.

The above algorithm for variable grouping has another important consequence related to the testability of the netlist resulting from the bi-decomposition. Here we only formulate this result and refer the reader to [8] for details.

Theorem 5: If function F(X) = { Q(X), R(X) } is OR-, AND-, or EXOR-bi-decomposable with variable sets (X_A, X_B), that has been selected using the algorihtm in Fig. XXX, and the ISFs were derived using the formulas of Theorems 3 and 4, then the resulting netlist does not have redundant internal signals, i.e. it is completely testable for all stuck-at-0 and stuck-at-1 faults assuming the single stuck-at fault model.

6 Reusing decomposed blo cks

While developing the approach to bi-decomposition, we experimented with a number of strategies to enable the efficient reuse of components across the netlist. One of the strategies we explored consisted in modifying the variable grouping algorithm to prefer groupings that create functions existing among the decomposed components. However, this approach tended to degrade the delay of the resulting circuits by creating variable subsets of disballanced sizes.

The solution to the problem of component reuse has been found in developing an original caching technique, which allows for fast search among the already existent components. The idea of this technique is suggested by the observation that functions prior to the decomposition are incompletely specified (given by on-set Q and off-set R) while the decomposed functions are completely specified. So the problem of component reuse can be reduced to the efficient way of checking that among the set of CSFs there exists function F such that F (or it complement) belongs to the interval (Q, $\overline{R}$).

Theorem 5: Let an ISF F be specified by the on-set Q and the off-set R. A CSF f belongs to the ISF F iff

$$Q \,\&\, \bar{f} = 0 \text{ and } R \,\&\, f = 0.$$

The complement of f belongs to the ISF F iff

$$R \,\&\, \bar{f} = 0 \text{ and } Q \,\&\, f = 0.$$

Checking many functions for belonging can be performed efficiently if the completely specified functions are grouped by their supports. This can be done by introducing a lossless hash table hashing supports (represented as BDDs) into pointer to linked lists of completely specified functions (also represented as BDDs). In this case, checking reduces to getting the pointer to the linked list of all functions with the given support and walking through the list of functions to determine whether one of them (or its complement) belongs to the given interval.

This technique turned out to be very efficient in practice by achieving up to 20% component reuse. The gain in area and CPU time is, in fact, more substatial, especially when a gate is reused on an early stage of the decomposition process, because in this case there is no need to generate any gates belonging to the fanin cone of the given gate.

7 Bi-decomposition algorithm

This section present the upper-level procedure that performs one step of recursive bi-decomposition (Fig. 7).

```
procedure BiDecompose( bdd Qi, bdd Ri )
{ bdd Q, R, S, $F_A$, $F_B$, F;
   ( Q, R ) = RemoveInessentialVariables( Qi, Ri );
   S = Find_Support( Q, R );
   if ( LookupCacheForACompatibleComponent(Q, R, S ) ) {
      F = GetCompatibleComponent( Q, R, S );
      return F; }
   if ( $|S| \le 2$ ) {
      ($F_A$, $F_B$, gate) = FindGate( Q, R );
      F = AddGateToDecompositionTree( $F_A$, $F_B$, gate );
      return F; }
   bdd $X_A^{OR}$, $X_B^{OR}$, $X_A^{AND}$, $X_B^{AND}$, $X_A^{EXOR}$, $X_B^{EXOR}$, $X_A^{BEST}$, $X_B^{BEST}$;
   ( $X_A^{OR}$, $X_B^{OR}$ ) = GroupVariablesOR( Q, R, S );
   ( $X_A^{AND}$, $X_B^{AND}$ ) = GroupVariablesAND( Q, R, S );
   ( $X_A^{EXOR}$, $X_B^{EXOR}$ ) = GroupVariablesEXOR( Q, R, S );
   ( $X_A^{BEST}$, $X_B^{BEST}$, gate ) = FindBestVariableGrouping(
                  ($X_A^{OR}$, $X_B^{OR}$), ($X_A^{AND}$, $X_B^{AND}$), ($X_A^{EXOR}$, $X_B^{EXOR}$) );
   if ( ($X_A^{BEST}$, $X_B^{BEST}$) == ($\varnothing$, $\varnothing$) )
      ($X_A^{BEST}$, $X_B^{BEST}$, gate) = GroupVariablesWeak( Q, R, S );
   ($Q_A$,$R_A$) = DeriveComponentAFunctions( Q,R, $X_A^{BEST}$, $X_B^{BEST}$ );
   $F_A$ = BiDecompose($Q_A$,$R_A$);
   ($Q_B$,$R_B$)=DeriveComponentBFunctions(Q,R,$F_A$,$X_A^{BEST}$,$X_B^{BEST}$);
   $F_B$ = BiDecompose($Q_B$,$R_B$);
   F = AddGateToDecompositionTree( $F_A$, $F_B$, gate );
   return F;
}
```

Fig. 7. The pseudo-code of bi-decomposition algorithm.

Procedure BiDecompose() is called with two arguments, Qi and Ri, the initial on-set and off-set of the ISF to be decomposed. It returns a CSF in the range (Qi, $\overline{\mathrm{Ri}}$) representing the network of gates implementing the ISF.

If the ISF has inessential variables, they are removed using a simple greedy algorithm and a new ISF (Q, $\overline{R}$) is created. In practice, inessential variables occur in less than 1% of recursive calls for typical MCNC benchmarks.

Next, support S of the ISF is determined and the cache is searched for a compatible component. If the component is found, there is no need to perform decomposition because the netlist implementing the function already exists and the CSF representing it can returned right away.

If there is no compatible component in the cache, the procedure starts decomposing the function. First, support S is checked for being less than two. If it is the terminal case, an approapriate two-input gate is added to the decomposition tree and the gate's CSF is returned. Otherwise, the procedure calls functions GroupVariables(), to find variable sets X_A and X_B, leading to strong bi-decomposition with OR, AND, and EXOR gates.

Procedure FindBestVariableGrouping() considers the variable sets and determines the best one. The cost function evaluating the variable sets takes into account two factors: how many variables are included into X_A and X_B (the more, the better), and whether X_A and X_B are well-ballanced (the closer their sizes are, the better). The procedure returns the best variable grouping and the indication what decomposition to perform (OR, AND, or EXOR).

If variable grouping with non-empty variable sets X_A and X_B is not available (this happens in 20-30% of recursive calls for typical MCNC benchmarks), procedure GroupVariablesWeak() finds the best grouping to perform weak AND- or OR-bi-decomposition, which always exists. The variable grouping in this case is reduced to determining X_A that introduces as many don't-cares into the ISF of component A as possible (recall that in the case of weak bi-decompositon, X_B is empty). After some experimentation, we found that the best results are achived when X_A includes only one variable. The reasons is that, in this case, X_A and X_B have close sizes, which leads to well-ballanced netlilsts, which in turn reduces the delay.

Given the variable sets and the type of decomposition, the on-set and off-set Q_A and R_A of the ISF of component A are derived using the formulas of Section 4. Next procudure BiDecompose() is called recursively for component A, returning the completely specified function f_A representing this component by a netlist of gates. The CSF f_A together with variable sets X_A and X_B are used to compute the on-set and off-set Q_B and R_B of the ISF of component B. Procedure BiDecompose() is called again for component B.

Finally, CSFs f_A and f_B derived in the process of decomposition and the information about the decomposed gate is used to find the CSF f, which represent the netlist implementing the initial ISF. At the end of BiDecompose(), the CSF f is returned to the calling procedure.

8 Experimental results

The algorithm has been implemented in the program BI-DECOMP written in platform-independent C++ using the BDD package "BuDDy", Ver. 1.9 [14]. The program has been tested on a Pentium 266Mhz PC under Microsoft Windows 98. Table 2 shows the experimental results for running BI-DECOMP on MCNC benchmarks. *16Sym8* is a 16-variable totally symmetric function with polarity "00001111000011110". The results of area-oriented mapping into the two-input gate library produced by SIS[15] is given for comparison.

Columns "Ins" and "Outs" give the number of inputs and outputs in the benchmark function. Columns "Gates" and "Exors" give the number of all two-input gates and EXOR gates in the resulting netlist. Column "Cascades" gives the number of logic levels. Columns "Area" and "Delay" show the area and delay of the resulting circuit computed assuming the typical area/delay ratio for two-input gates. For example, the ratio of area and delay of EXOR and NOR is assumed to be 5/2 and 2.1/1.0 respectively. Finally, column "Time" gives the CPU time in seconds needed to perform the bi-decomposition and write the results into a BLIF file (the input file reading time is not included). The correctness of the resulting networks has been tested using a BDD-based verifier.

Table 2 shows that in almost all cases BI-DECOMP outperforms SIS in both area and delay. Both programs used the PLA input files. In case of SIS, after reading input files and before applying mapping, the function was optimized using commands "resub –a; simplify –m" (as suggested in the SIS manual). We deciced not to use the curcuits preoptimized with *script.rugged*, because even though it gave significant advantage in the number of gates, in most cases it led to the increase in the delay compared to the circuits generated using mapping only.

The large area and delay of two-input-gate circuits generated by SIS can be explained by the fact that SIS used mostly NOR/NAND gates and all but ignored other two-input gates, even through they were listed in the library (see column "Exors" in the SIS section of Table 2).

Benchmark			SIS						BI-DECOMP					
name	ins	outs	gates	exors	area	casc	delay	time,c	gates	exors	area	casc	delay	time,c
9sym	9	1	255	5	526	15	21.0	3.1	**65**	**27**	**230**	**11**	**17.1**	**0.17**
alu4	14	8	1305	4	2687	21	37.6	29.0	**288**	**26**	**785**	**13**	**18.2**	**3.35**
cps	24	109	2354	12	4782	25	42.2	49.0	**1608**	**152**	**4382**	**12**	**18.4**	**7.74**
duke2	22	29	706	0	**1466**	17	23.2	4.9	**608**	**70**	1695	**11**	**17.1**	**3.19**
e64	65	65	2141	0	4301	8	13.0	8.8	**1443**	**0**	**3607**	**7**	**8.4**	**4.51**
misex3	14	14	1524	11	3134	17	33.0	27.8	**897**	**186**	**2707**	**15**	**22.1**	**5.88**
pdc	16	40	1320	0	2653	17	26.6	27.4	**866**	**85**	**2375**	**14**	**20.7**	**3.02**
spla	16	46	1201	44	2541	16	27.4	53.2	**811**	**81**	**2227**	**15**	**21.9**	**2.52**
vg2	25	8	**90**	0	**195**	**11**	**14.2**	**1.00**	216	38	635	**11**	14.5	3.90
16sym8	16	1	958	0	2123	37	56.0	488.6	**299**	**113**	**1030**	**21**	**33.1**	**11.87**

Table 2. Comparison of decomposition results with SIS [15].

Table 3 shows the results of comparison of BI-DECOMP with a BDD-based bi-decomposition package, BDS [10,11]. Notice that measurements in [11] are made in terms of the standard library *mcnc.genlib*, which includes three-input and four-input gates, while measurements of BI-DECOMP are in terms of two-input gates. BDS performs numerious preprocessing steps in order to achieve better partitioning before bi-decomposition, while our approach applies the above algorithms to the input functions without any preprocessing. Therefore it is not quite clear why BI-DECOMP outperforms BDS (e.g. benchmark *alu4.pla*).

Bench mark	BDDlopt[11]			BI-DECOMP		
	gates	exors	time, c	gates	exors	time, c
5xp1	**67**	4/16	0.4	70	17	0.11
9sym	**42**	0/4	1.0	65	27	0.17
alu2	230	13/53	2.8	**221**	52	0.82
alu4	582	23/124	15.9	**288**	26	3.35
Cordic	47	6/16	0.5	**44**	15	18.80
rd84	62	6/12	1.4	**55**	18	0.17
t481	**15**	5/5	0.3	17	5	0.49

Table 3. Comparison of decomposition results with [11].

We conjecture that BDS does not make the most of the strong bi-decomposability of the functions. Using the terminology introduced above, BDS applies only weak bi-decomposition (when one of the decomposed functions can potentially depend on all input variables). Meanwhile, in our experience, it is the strong bi-decomposition with both XA and XB (see Fig. 1) different from the empty set that leads to the fast reduction in the size of component supports (resulting in smaller area) and facilitates creating well-ballanced netlists (resulting in shorter delay).

9 Conclusions

We presented a new approach to decomposition of incompletely specified multi-output functions into netlists of two-input AND/OR/EXOR gates. The decomposition is based on boolean formulas with quantifiers which can be efficiently evaluated using a standard BDD package.

Our algorithm can be characterized as follows:

- The generated netlists are *compact* because it uses the EXOR gates for EXOR-intensive circuits, exploits external and internal don't cares, and achieves significant degree of component reuse by applying an original caching technique.
- The netlists are *well-ballanced* (i.e., the subnetworks for both inputs of a logic gate are typically similar in the number of gates), which significantly reduces the delay of the resulting circuit.

- Netlists created by our algorithm are 100% testable. A test pattern generation technique can be integrated into the decomposition algorithm with little if any increase in the complexity and running time.

The future work includes extending the algorithm to work with arbitrary standard cell libraries, integration of ATPG into the process of decomposition, and generalization of the algorithm for multi-valued logic with potential applications in datamining.

References

[1] R. L. Ashenhurst. “The decomposition of switching functions”. *Ann. Computation Lab*, Harvard University, Vol. 29, pp. 74-116, 1959.

[2] A. Curtis. *New approach to the design of switching circuits.* Van Nostrand, Princeton, NJ, 1962.

[3] V. Bertacco, M. Damiani. "The Disjunctive Decomposition of Logic Functions". *Proc. of ICCAD '97*, pp. 78-82.

[4] S. Minato, G. De Micheli. "Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms". *Proc. of ICCAD' 98*, pp. 111-117.

[5] T.Sasao, M.Matsuura. “DECOMPOS: An Integrated System for Functional Decomposition”. *Proc. of IWLS ’98*, pp. 471-477.

[6] D.Bochmann, F.Dresig, B.Steinbach, “A new decomposition method for multilevel circuit design”. *Proc. of Euro-DAC*, Amsterdam, 1991, pp. 374 – 377.

[7] B. Steinbach, F. Schumann, M. Stockert. “Functional Decomposition of Speed Optimized Circuits”. In *Power and Timing Modelling…*, D. Auvergne, R. Harternskin, eds., Springer-Verlag, 1993, pp. 65-77.

[8] B. Steinbach, M. Stockert. “Design of Fully Testable Circuits by Functional Decomposition and Implicit Test Pattern Generation”. *Proc. of VLSI Test Symposium*, New Jersey, 1994, pp. 22-27.

[9] B. Steinbach, A. Wereszczynski, “Synthesis of Multi-Level Circuits Using EXOR-Gates”. *Proc. of "IFIP WG 10.5 - Workshop on Applications of the Reed-Muller Expansion in Circuit Design"*, Japan, 1995, pp. 161 - 168

[10] C. Yang, M. Ciesielski, V. Singhal. "BDS: A BDD-based Logic Optimization System". *Proc. of DAC 2000*, pp. 92-97.

[11] C.Yang, M.Ciesielski. “BDD-Based Logic Optimization System”. *Tech. Report CSE-00-1*, February 2000.

[12] T. Sasao, J. Butler, “On bi-decomposition of logic functions”, *Proc. of IWSL*, Lake Tahoe, California, May 1997.

[13] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Comp., Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[14] J. Lind-Nielsen. BDD Package BuDDy, v.1.9, August 2000, http://www.itu.dk/research/buddy/index.html

[15] E. Sentovich, et al. “SIS: A System for Sequential Circuit Synthesis”, Tech. Rep. UCB/ERI, M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.