An Algorithm for Bi-Decomposition of Logic Functions

Alan Mishchenko
Portland State University
B.O. Box 751,
Portland, OR 97207, USA
1-503-725-2780
alanmi@ee.pdx.edu

Bernd Steinbach
Freiberg Univ. of Mining and Techn.
Bernhard-von-Cotta-Str. 1
D-09596 Freiberg, Germany
03731-39-2568
steinb@informatik.tu-freiberg.de

Marek Perkowski
Portland State University
B.O. Box 751,
Portland, OR 97207, USA
1-503-725-5411
mperkows@ee.pdx.edu

ABSTRACT

We propose a new BDD-based method for decomposition of multi-output incompletely specified logic functions into netlists of two-input logic gates. The algorithm uses the internal don't-cares during the 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 terms of area and delay of the resulting circuits with comparable CPU time.

1. INTRODUCTION

Decomposition of Boolean functions consists in breaking large logic blocks into smaller ones while keeping the network functionality unchanged. Decomposition plays an important role in logic synthesis. Research in this area started in the 1950s [1,2]. Recently, there has been a revival of interest in disjoint decomposition (called also disjunctive decomposition) [3,4,5]. Decomposition methods are classified as follows:

1) Each block of the resulting network has a single binary output, or may have multiple binary outputs (Ashenhurst and Curtis decomposition, respectively).
2) Supports of the blocks may overlap, or never overlap (disjoint decomposition).
3) Each block has two or less inputs (bi-decomposition), or the number of inputs may be larger than two.
4) Decomposition is performed as technology mapping for FPGAs, as a technology-independent transformation of logic circuits, or as a specialized mapping technique.
5) The decomposed structure is derived by splitting the larger blocks into the smaller ones, or the decomposition structure is assembled by iteratively adding small components until the network is equivalent to the initial specification.
6) BDDs are used in the decomposition algorithm, or not. If yes, BDDs are used to represent functions and store intermediate results, or BDDs are used as the essential data structure directing the decomposition process.
7) Decomposition shares blocks across outputs (logic cones) or decomposes each output (logic cones) independently.
8) The algorithm allows for don't-cares 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:

1) Each decomposed block has a single binary output.
2) Blocks may have overlapping supports.
3) Each resulting block has two or less inputs.
4) It is a technology-independent decomposition.
5) Larger components are split into smaller ones.
6) BDDs are used to store functions.
7) Blocks are shared across outputs and internal logic cones.
8) Incompletely specified functions are allowed; the more don't-cares, the more efficient is the algorithm.

According to the above classification, the closest matches to our algorithm are its previous versions [6,7,8,9] and the recent approach [10,11]. As evidenced by our experiments, the present version of the algorithm outperforms its previous versions and compares favorably to [10,11]. A more detailed analysis of the differences of these approaches is given in Section 8 of the paper.

The paper is organized as follows: Section 2 introduces the notations and the decomposition models. Section 3 gives necessary and sufficient conditions for AND-, OR-, and EXOR-bi-decomposition. Section 4 gives the formulas for deriving the decomposed functions. Section 5 presents the variable grouping strategy. Section 6 presents hashing techniques. Section 7 discusses the 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 variable 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 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 *compatible* with the ISF $F = (Q, \overline{R})$, iff $Q \le f \le \overline{R}$.

This paper discusses *bi-decomposition* [12] (also known as *grouping* [6]), or decomposition of ISFs into netlists of two-input logic gates. One-step bi-decomposition is schematically represented in Fig. 1. Block C is an 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.

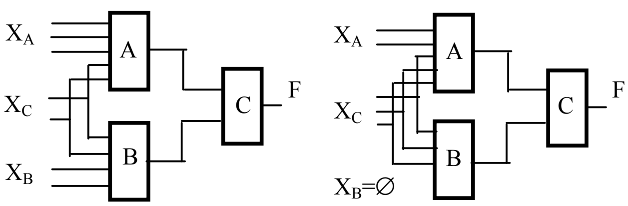

Figure 1. Schematic representation of the two types of bi-decompositoin: strong (left) and weak (right).

Notice that before decomposition function F(X) in Fig. 1 (right) has five inputs. The weak bi-decomposition creates five-input component A and three-input component B. The advantage of this decomposition consists in increasing the number of don't-cares of component A. As a result, the function of block A previously non-bi-decomposable in the strong sense, after the weak bi-decomposition may have a strong bi-decomposition.

$\exists_{ab}F$ $\xLeftarrow{\text{OR}}$ F(a,b,c,d) $\xRightarrow{\text{AND}}$ $\forall_{ab}F$

cd	$\exists_{ab}F$
00	1
01	1
11	0
10	1

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

Cd	$\forall_{ab}F$
00	0
01	0
11	0
10	1

Figure 2. Karnaugh map illustration of quantifications.

In this paper, all Boolean functions and their supports are represented using binary decision diagrams (BDDs) [13]. It is assumed that the reader is familiar with basic principles of BDDs. Two BDD operators, *existential* and *universal* quantification, are used extensively in the formulas. Quantification of a CSF w.r.t. a variable x is defined as follows: $\exists_x f = f_0 + f_1$ (existential) and $\forall_x f = f_0 \,\&\, f_1$, (universal). Symbols "+" and "&" stand for Boolean OR and AND, while f_0 and f_1 are the cofactors of f: $f_0 = f|_{x=0}$, $f_1 = f|_{x=1}$ [13]. Quantification over a set of variables is defined as an iterative quantification over each variable in the set.

For illustration, 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.

3. CHECKING BI-DECOMPOSABILITY

3.1 Bi-decomposition with an OR-gate

Consider the 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:

$$F = \text{OR}(\, a \oplus b,\; c\overline{d}\,)$$

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

Figure 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 these 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.

Property: F(X) is OR-bi-decomposable with variable sets (X_A,X_B,X_C), $X_C = \varnothing$, iff in the Karnaugh map there is no cell containing 1 such that 0's appear in *both* the row and the column to which this cell belongs.

If X_C is not empty, $2^{|X_C|}$ Karnaugh maps corresponding to different assignments of variables X_C are considered, but the condition of bi-decomposability remains essentially the same: if Property holds for all cells of all the Karnaugh maps, the function is OR-bi-decomposable. Therefore in the theorems below, there is no restriction on X_C, meaning that it can be either empty or non-empty. We refer the reader to [6,7] for a detailed discussion.

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 up all the 1's contained in the rows (columns).

Theorem 1: F(X) = { Q(X), R(X) } is OR-bi- decomposable with variable sets (X_A, X_B) 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 above formula. 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 an 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 including 1 and n variables respectively.

Theorem 2: F(X) = { Q(X), R(X) } is EXOR-bi-decomposable with variable sets (X_A, X_B), such that $|X_A| = 1$ and $|X_B| = n$, 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 EXOR-bi-decomposibility with arbitrary non-overlapping sets X_A and X_B is performed by a specialized algorithm (Fig. 4). 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 an 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.

```
procedure CheckExorBiDecomp( bdd Q, bdd R, bdd XA, bdd XB)
{  bdd QA = 0, RA = 0, QB = 0, RB = 0;
   bdd qA = 0, rA = 0, qB = 0, rB = 0;
   while ( Q != 0 )  {
       bdd Cube = SelectOneCube( Q );  qA = qA + ∃BCube;
       while ( qA + rA != 0 )    {
      qB = ∃xA ( Q & rA  + R & qA );  rB  = ∃xA ( Q & qA  + R & rA );
      if ( qB & rB != 0 )  return (0, 0, 0, 0);
      Q  = Q - ( qA + rA );  R  = R - ( qA + rA );
      QA = QA + qA;          RA = RA + rA;
      qA = ∃xB ( Q & rB  + R & qB ); rA  = ∃xB ( Q & qB  + R & rB );
      if ( qA & rA != 0 )   return (0, 0, 0, 0);
      Q  = Q - ( qB + rB );  R  = R - ( qB + rB );
      QB = QB + qB;          RB = RB + rB;
   } }
   if ( R != 0 ) {  RA = RA + ∃xB R;  RB = RB + ∃xA R;  }
   return (QA, RA, QB, RB);
}
```

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

Internally called procedure SelectOneCube() returns a randomly selected cube. The Boolean function of the cube is quantified and projected in the directions of X_A and X_B to find one part of the EXOR-decomposable component, which is next added to the on-sets and off-set of the components A and B and subtracted from the initial on-set and off-set. If the on-set and off-set of components A and B are incompatible, CheckExorBiDecomp() returns zeros, otherwise the process is repeated as long as the given on-set is not empty. If at the end the off-set contains some minterms, they are added to off-sets of components A and B. For a detailed discussion of this algorithm, see [9].

4. DERIVING DECOMPOSED FUNCTIONS

This section presents formulas for deriving ISFs implementing components A and B (see Fig. 1). The case of an EXOR-bi-decomposition has been addressed in the previous section, because the decomposed functions in EXOR-bi-decomposition are derived as a result of the bi-decomposability check.

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

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

Theorem 4: Let F(X) = { Q(X), R(X) } be OR-bi-decomposable with variable sets (X_A, X_B) and a CSF 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 ISFs representing components A and B are summarized in Table 1. Note that removing the existential quantifier w.r.t. X_B in the formulas for component A in the case of strong OR-decomposition leads to the corresponding formulas for weak OR-decomposition, because in the case of weak decomposition the variable set X_B is empty. Symbol "–" stands for Boolean SHARP ($A-B = A \,\&\, \overline{B}$).

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

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

5. VARIABLE GROUPING

An important task during the bi-decomposition is 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. First, X_A and X_B are initialized with a single variable. Next, 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 ∈ S   {
    XA = {x};
    for all y ∈ S – {x}   {
       XB = {y};
       if ( CheckDecomposability( Q, R, XA, XB ) )
           return ( XA, XB );
    } }
    return (∅, ∅);
}
```

Figure 5. Algorithm to find the initial 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 them, or two empty sets, if the function is not bi-decomposable in the strong sense with *any* variable grouping. The procedure CheckDecomposability() performs an OR-, AND-, or EXOR-bi-decomposability check, as discussed in Section 3, depending on what initial grouping is sought.

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, the procedure considers the remaining variables one by one and tries to add them to 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 is to keep the variable set sizes close.

Notice that the above greedy way of building X_A and X_B does not guarantee that they are the largest possible (meaning that the support sizes of components A and B are minimum). In practice, however, it gives a reasonably good trade-off between the size and the quality of the resulting variable sets 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 reduced the netlist area by less than 3% on average but the CPU time increased by 100%.

```
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 ∈ S – (X_A ∪ X_B)
        if ( |X_A| ≤ |X_B| )
                // try adding the new variable z first to X_A, next to X_B
                if ( CheckDecomposability( Q, R, X_A ∪ {z}, X_B ) )
                        X_A = X_A ∪ {z};
                else if ( CheckDecomposability( Q, R, X_A, X_B ∪ {z} ) )
                        X_B = X_B ∪ {z};
        else … // similarly if ( |X_A| > |X_B| )
   return ( X_A, X_B );
}
```

Figure 6. Procedure to find the variable grouping.

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 algorithm in Fig. 6, 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 BLOCKS

The functions considered as input to the decomposition algorithm are incompletely specified (the on-set/off-set pairs) while the decomposed functions are completely specified. To enable efficient reuse of components across the netlist we developed an original caching technique, which allows checking that among the set of CSFs there exists a function F such that F (or it complement) belongs to the interval $(Q, \overline{R})$.

Theorem 6: Let an ISF F be given by on-set Q and off-set R. A CSF f is compatible with an 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 compatibility with the given ISF can be performed efficiently if the completely specified functions are sorted by support. This can be done by introducing a lossless hash table that hashes supports (represented as BDDs) into pointer to linked lists of CSFs (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 to determine whether one of them (or its complement) belongs to the given interval.

This technique turned out to be efficient in practice by achieving up to 30% component reuse. The gain in area and CPU time is even more substantial, especially when a gate is reused on an early stage of the decomposition process, because in this case there is no need to generate the fanin cone of the given gate.

7. BI-DECOMPOSITION ALGORITHM

This section presents the upper-level procedure performing 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| ≤ 2 ) {
      (F_A, F_B, gate) = FindGate( Q, R );
      F = AddGateToDecompositionTree( F_A, F_B, gate );
      AddFunctionToCache( F );
      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) == ( ∅, ∅ ) )
        (X_A^BEST, X_B^BEST, gate) = GroupVariablesWeak( Q, R, S );
   (Q_A,R_A) = DeriveComponentA( Q,R, X_A^BEST, X_B^BEST, gate);
   F_A = BiDecompose( Q_A,R_A );
   (Q_B,R_B)=DeriveComponentB(Q,R,F_A,X_A^BEST,X_B^BEST, gate);
   F_B = BiDecompose( Q_B,R_B );
   F = AddGateToDecompositionTree( F_A, F_B, gate );
   AddFunctionToCache( F );
   return F;
}
```

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

The arguments, Qi and Ri, are the initial on-set and off-set of the ISF to be decomposed. The return value is a CSF in the range (Qi, $\overline{\mathrm{Ri}}$) representing the resulting network of gates. At the beginning the support of the ISF is minimized by a simple greedy algorithm and a new ISF $(Q, \overline{R})$ is created. In practice, support minimization occurs in less than 1% of recursive calls for typical MCNC benchmarks. Next, a cache look up is performed. If it is successful, it means that the CSF in the given interval has already been implemented and can be returned right away. If it is the terminal case, an appropriate two-input gate is added to the decomposition tree and to the cache before returning the gate's CSF. Otherwise, the procedure calls three functions GroupVariables() to find sets X_A and X_B leading to a strong bi-decomposition with OR, AND, and EXOR gates.

Procedure FindBestVariableGrouping() considers the variable sets and determines the best one taking into account that X_A and X_B should be well-balanced. 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 variable grouping to perform weak AND/OR-bi-decomposition, which always exists.

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. Calling BiDecompose() recursively for component A returns the CSF f_A representing this component by a netlist of gates. The CSF f_A together with X_A and X_B are used to compute the ISF of the component B. Procedure BiDecompose() is again called recursively for component B. The CSF f of the netlist implementing the initial ISF is found using

CSFs f_A and f_B and the decomposition gate. Finally, the CSF f is inserted into the cache and returned.

8. EXPERIMENTAL RESULTS

The algorithm has been implemented in the program BI-DECOMP written in platform-independent C++ using the BDD package CUDD [14]. The program has been tested on a 300Mhz Pentium II PC with 64Mb RAM under Microsoft Windows 98. The correctness of the resulting networks has been checked using a BDD-based verifier.

To demonstrate the optimization potential of bi-decomposition for both delay and area, we carried out two series of experiments. In the first one (Table 2), the bi-decomposition of MCNC benchmarks into two-input NAND/NOR/EXOR/NEXOR gates performed by BI-DECOMP is compared with similar results produced by SIS[15]. For SIS, the benchmarks have been preprocessed using *script.rugged* (with and without *speed_up*) followed by a delay-oriented mapping into a subset of *mcnc.genlib* containing the above listed two-input logic gates.

Columns "gates" ("exors") and "levels" give the number of (EXORs) gates and logic levels. Columns "area1" and "delay1" show results after running *script.rugged* and mapping. Columns "area2" and "delay2" show results if *script.rugged* is followed by *speed_up*. SIS used at the most one or two EXOR gates per circuit. Column "time" gives the CPU time in seconds needed for BI-DECOMP to perform the decomposition and write the resulting BLIF file. The runtime for SIS was dominated by *script.rugged* and was one minute on average.

From Table 1 we see that BI-DECOMP produces larger area compared to SIS. This is because BI-DECOMP builds BDDs for the primary outputs and applies the bi-decomposition algorithm to them without any pre- or post-processing. However, the delay of the networks produced by BI-DECOMP is better because of the way the algorithm derives supports of the components resulting in a well-balanced bi-decomposition.

The second series of experiments (Table 3) shows that BI-DECOMP produces good results in terms of area for EXOR-intensive circuits. BI-DECOMP is compared with SIS[15] (*script.rugged* followed by area-oriented mapping into *mcnc.genlib*) and BDS, a BDD-based logic synthesis system [10]. (The results for SIS and BDS are taken from [11].) The number of gates after decomposition is shown in column "gates". BDS outputs a netlist of 2/3/4-input (N)ANDs and (N)ORs, and 2-input (N)EXORs. To make a fair comparison, we translated the two-input-gate output of BI-DECOMP into a set of gates comparable to the output of BDS. (This is why the gate count of BI-DECOMP for *9sym* in Table 3 differs from that of Table 2). The column "area" shows the results of *script.rugged* followed by area-oriented mapping into *mcnc.genlib*.

One of the reasons why BI-DECOMP outperforms BDS in terms of gates on some test cases in that BDS does not make the most of the strong bi-decomposability of Boolean functions. The BDS algorithm does not guarantee that weak bi-decomposition is applied only when there is no strong bi-decomposition. Meanwhile, it is the strong bi-decomposition, with both X_A and X_B not empty (Fig. 1), that leads to the fast reduction in the component size (smaller area) and creation of well-balanced netlists (shorter delay). Another difference between BDS and BI-DECOMP is in the use of don't-cares. BDS uses them locally, to optimize the size of the BDD representation of one component. BI-DECOMP uses don't-cares locally *and* passes don't-cares (both external and locally generated) to the next decomposition steps by allowing the generic bi-decomposition procedure to work on incompletely specified functions.

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 that can be 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-balanced*, which significantly reduces the delay of the resulting circuit.
- The resulting netlists are 100% testable for single stuck-at faults [8]. Test pattern generation can be integrated into the decomposition algorithm with little if any increase in the complexity and runtime.

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 [16].

10. ACKNOWLEDGEMENTS

The research was sponsored by the NSF grant for the U.S./German collaborative research in functional decomposition for datamining (grant #315/PPP/gű-ab). The first author has been partially supported by a research grant from Intel Corporation.

11. REFERENCES

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

[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 1997*, pp. 78-82.

[4] S. Minato, G. De Micheli. "Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms". *Proc. of ICCAD 1998*, pp. 111-117.

[5] T.Sasao, M.Matsuura. "DECOMPOS: An Integrated System for Functional Decomposition". *Proc. of IWLS 1998*, pp. 471-477.

[6] D.Bochmann, F.Dresig, B.Steinbach, "A new decomposition method for multilevel circuit design". *Proc. of Euro-DAC* 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. Hartenstein, 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* 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 IWLS* 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] F. Somenzi. BDD package “CUDD v. 2.3.0.” http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.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.

[16] B. Steinbach, M. A. Perkowski, Ch. Lang. “Bi-Decomposition of Multi-Valued Functions for Circuit Design and Data Mining Applications.” *Proc. of ISMVL 1999*, pp. 50 – 58. http://www.informatik.tu-freiberg.de/prof2/publikationen/ismvl99_final.ps.

Table 2. Comparison of delay-oriented decomposition results with SIS.

Benchmark			SIS						BI-DECOMP					
			script.rugged + mapping				+ *speed_up*							
name	ins	outs	gates	levels	area1	delay1	area2	delay2	Gates	exors	levels	area	delay	time,c
9sym	9	1	235	18	487	30.0	486	28.5	83	26	11	226	17.1	0.39
alu4.pla	14	8	211	26	430	37.4	490	27.8	279	31	12	619	20.4	4.07
cps	24	109	1188	21	2428	54.5	2629	40.5	1733	130	13	3679	22.8	8.20
duke2	22	29	467	27	961	53.4	1280	39.0	684	70	12	1503	20.0	4.83
e64	65	65	253	125	506	169.0	798	32.3	1558	0	7	2999	11.2	3.46
misex3	14	14	717	30	1467	51.0	1528	42.2	1017	166	15	2414	27.8	6.87
pdc	16	40	415	20	861	31.0	955	29.8	328	30	8	712	14.5	1.48
spla	16	46	658	21	1350	31.2	1369	29.9	786	67	14	1691	23.3	2.36
vg2	25	8	107	14	214	20.4	234	13.3	259	39	11	601	18.3	7.91
Average			472	33.5	967	53.1	1085	31.4	747	62	11.4	1605	19.5	4.40
Ratio			100%	100%	89%	169%	100%	100%	158%	8%	34%	148%	62%	

Table 3. Comparison of area-oriented decomposition results for EXOR-intensive circuits with SIS and BDS.

Benchmark			SIS		BDS				BI-DECOMP			
name	ins	outs	gates	area	gates	exors	area	time, c	gates	exors	area	time, c
5xp1	7	10	81	195	67	16	172	0.4	62	21	160	0.27
9sym	9	1	152	396	42	4	109	1.0	50	27	155	0.33
alu2	10	6	217	524	230	53	632	2.8	198	61	519	1.60
alu4.blif	14	8	409	996	582	124	1655	15.9	508	126	1264	5.83
cordic	23	2	34	94	47	16	126	0.5	33	15	121	10.32
f51m	8	8	58	139	56	11	174	0.3	40	15	110	0.16
rd53	5	3	22	47	25	6	72	0.2	21	8	67	0.06
rd73	7	3	106	258	45	8	133	0.8	39	19	122	0.22
rd84	8	4	192	468	62	12	189	1.4	54	25	166	0.43
t481	16	1	407	1023	15	5	45	0.3	17	6	65	1.43
z4ml	7	4	20	59	20	6	53	0.1	30	6	61	0.11
Average			154.5	435	108.2	23.7	305	2.15	95.6	29.9	256	1.89
Ratio			100%	100%	70%	100%	70%	100%	62%	126%	59%	88%