BDD-Based Logic Decomposition: Theory

Yung-Te Lai, Massoud Pedram,
& Sarma B. K. Vrudhula (Sastry)

CENG Technical Report 92-17

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

BDD-Based Logic Decomposition: Theory

Yung-Te Lai, Massoud Pedram

Dept. of EE-Systems
Univ. of So. California
LA, CA 90089

Sarma B.K. Vrudhula
(a.k.a. Sarma Sastry)
Dept. of ECE
Univ. of Arizona
Tuscon, AZ 85721

October 23, 1992

Contents

List of Figures

List of Tables

Abstract

This report presents the theory for (disjunctive and non-disjunctive) function decomposition using the BDD representation of Boolean functions. Incompletely specified as well as multi-output Boolean functions are addressed as part of the general theory. A novel algorithm (based on an EVBDD representation) for generating the set of all bound variables that make the function decomposable is also presented. We compared our BDD-based decomposition procedure with existing implementations of the Roth-Karp procedure and obtained significant speed-ups.

1 Introduction

We address the problem of decomposing combinational logic. The motivation is to separate a large circuit into smaller pieces which can be optimized more effectively or can be mapped into Look-Up Tables of an FPGA device.

The problem of Boolean decomposition of a function $f(x_0, \ldots, x_{n-1})$ is to find a representation of f which consists of a composition of simpler Boolean functions. In the late 1950's, Ashenhurst [1] and Curtis [3] determined the necessary and sufficient conditions for the existence of a given decomposition. Their methods are based on the Karnaugh map representation and hence are practical for functions of very few variables. In contrast, the decomposition method of Roth and Karp[10] uses covers of the onset and offset of the function and thus is more practical and efficient.

Shen and McKellar [11] proposed an algorithm for obtaining all simple disjunctive decompositions of a Boolean function. They construct a *decomposition graph* based on a necessary condition for the existence of simple disjunctive decompositions. Only the k-complete subgraphs of the decomposition graph need to be checked for decomposability. The construction of the decomposition graph is based on a *mod 2 map* rather than a Karnaugh map. The mod 2 map uses the Reed-Muller canonical form of a Boolean function. Even though a small number of bound sets are to be examined, their method still requires $O(2^n)$ computations.

Many problems involving Boolean functions, including decomposition, require efficient representations. Reduced, ordered Binary Decision Diagrams [2] (BDD) provide a compact and canonical representation of Boolean functions. The various aspects of Boolean decompositions that are addressed in this paper are based on BDDs. In addition, we will also have occasion to use a new data structure, called Edge-Valued Binary Decision Diagrams (EVBDDs) [7]. EVBDDs provide a representation of Boolean functions over the integer domain and have been shown to be useful for verification of arithmetic functions.

The paper is organized as follows. In Section 2, we give basic definitions, terminology and notations used thereafter. Our BDD-based disjunctive and non-disjunctive decomposition algorithms are presented in Sections 3 and 4. Incompletely specified logic functions and multi-output function decomposition are described in Sections 5 and 6. Section 7 is devoted to the decomposability issue, i.e., finding a bound set of variables that will reduce the variable support of the function. Application of the BDD-based decomposition algorithms to the synthesis of XC4000 FPGA device is discussed in Section 8. Experimental results and conclusions are given in Sections 9 and 10.

2 Definitions and Notations

Definition 2.1 A function $f(x_0, \ldots, x_{n-1})$ is said to be *decomposable* if f can be transformed to a form $f'(g_0, \ldots, g_{m-1})$ in which every function g_i has fewer than n variables and m is smaller than n. A decomposition is *disjunctive* if $f(x_0, \ldots, x_{n-1}) = f'(g_0(A_0), \ldots, g_{m-1}(A_{m-1}))$ where $\{A_0, \ldots, A_{m-1}\}$ is a partition of the set of input variables $\{x_0, \ldots, x_{n-1}\}$. A decomposition is *non-disjunctive* if there exist i and j such that $A_i \cap A_j \neq \phi$. A *simple* decomposition contains only a subfunction of the form $f(x_0, \ldots, x_{n-1}) = f'(g(A_0), A_1)$ where $A_0, A_1 \subseteq \{x_0, \ldots, x_{n-1}\}$ and $A_0 \cup A_1 = \{x_0, \ldots, x_{n-1}\}$. Again, if $A_0 \cap A_1 = \phi$ then it is a simple disjunctive decomposition; otherwise it is a simple non-disjunctive decomposition. The sets A_0 and A_1 are referred to as the *bound set* and the *free set* respectively. A ***bound (free) variable*** is a variable in the bound (free) set.

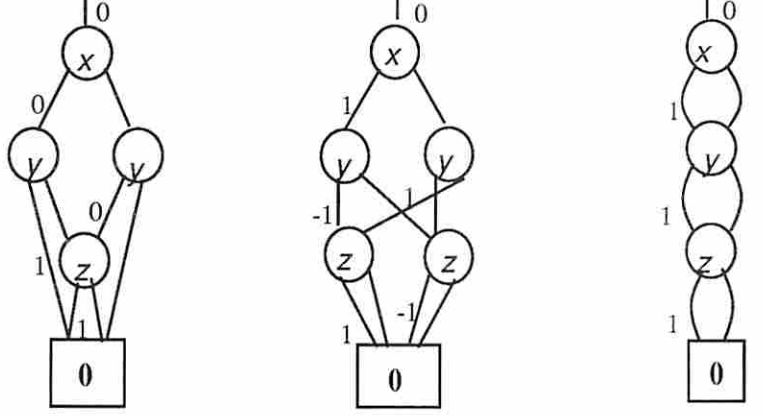
Definition 2.2 A BDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A terminal node $\mathbf{v}$ is either $\mathbf{0}$ or $\mathbf{1}$. A BDD is ordered if there exist an index function $index(x) \in \{0, \ldots, n-1\}$ such that for every nonterminal node $\mathbf{v}$, either $child_l(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_l(\mathbf{v})))$, and either $child_r(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_r(\mathbf{v})))$. A BDD is reduced if there is no nonterminal node $\mathbf{v}$ such that $child_l(\mathbf{v}) = child_r(\mathbf{v})$, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$. The function denoted by $\langle x, \mathbf{v}_l, \mathbf{v}_r\rangle$ is $xf_l + \bar{x}f_r$ where f_l and f_r are the functions denoted by $\mathbf{v}_l$ and $\mathbf{v}_r$, respectively. The functions denoted by $\mathbf{0}$ and $\mathbf{1}$ are the constant function 0 and 1, respectively.

In this paper, we consider only reduced, ordered BDD and use the following notation.

1. The left edge of a node represent 1 or the true edge and the right edge represents 0 or the false edge.
2. $\mathbf{v}$ represents both a BDD node and the BDD rooted by node $\mathbf{v}$.
3. $index(\mathbf{v})$: the index of the variable associated with node $\mathbf{v}$. If $\mathbf{v}$ is a terminal node, then $index(\mathbf{v}) = n$.
4. $\pi(i) = x$ if $index(x) = i$, that is, π is the variable ordering of BDD.
5. $new\_bdd(x, \mathbf{l}, \mathbf{r})$ returns a BDD node $\mathbf{v}$ such that $variable(\mathbf{v}) = x$, $child_l(\mathbf{v}) = \mathbf{l}$ and $child_r(\mathbf{v}) = \mathbf{r}$.
6. $eval(\mathbf{v}, \langle b_0, \ldots, b_{i-1}\rangle) = \mathbf{v}'$ where $\mathbf{v}$ and $\mathbf{v}'$ represent the functions $f(x_0, \ldots, x_{n-1})$ and $f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. When i is known, we also use $eval(\mathbf{v}, j)$ for $eval(\mathbf{v}, \langle b_0, \ldots, b_{i-1}\rangle)$ where $j = 2^{i-1}b_0 + \ldots + 2^0 b_{i-1}$.
7. $descendant(\mathbf{v})$ the descendant nodes of $\mathbf{v}$ including itself.
8. $replace\_node(\mathbf{v}, \mathbf{v}_o, \mathbf{v}_n)$ replaces node $\mathbf{v}_o$ in BDD $\mathbf{v}$ by node $\mathbf{v}_n$.

Definition 2.3 An EVBDD is a tuple $\langle c, \mathbf{f}\rangle$ where c is a constant value and $\mathbf{f}$ is a directed acyclic graph consisting of two types of nodes. There is a single terminal node $\mathbf{0}$. A nonterminal node $\mathbf{v}$ is represented by a 4-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v}), value\rangle$ such that

1. either $child_l(\mathbf{v}) = \mathbf{0}$ or $index(\mathbf{v}) < index(child_l(\mathbf{v}))$,
2. either $child_r(\mathbf{v}) = \mathbf{0}$ or $index(\mathbf{v}) < index(child_r(\mathbf{v}))$,
3. $child_l(\mathbf{v}) = child_r(\mathbf{v})$ and $value = 0$ can not exist at the same time, and



(a) carry (b) sum (c) x+y+z

Figure 1: EVBDD representation of full adder

4. there are no two nonterminal nodes **u** and **v** such that **u** = **v**.

Definition 2.4 An EVBDD $\langle c, \mathbf{f} \rangle$ denotes the arithmetic function $c + f$ where f is the function denoted by **f**. **0** denotes the constant function 0, and $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$ denotes the arithmetic function $x(v + l) + (1 - x)r$. The operator $eval_{ev}$ is defined as the following.

1. $eval_{ev}(\langle c, \mathbf{0} \rangle, \langle b_0, \ldots, b_{n-1} \rangle) = 0$,

2. $eval_{ev}(\langle c, \langle x_i, \mathbf{l}, \mathbf{r}, v \rangle \rangle, \langle b_0, \ldots, b_{n-1} \rangle) =$
   $eval_{ev}(\langle c + v, \mathbf{l} \rangle, \langle b_0, \ldots, b_{n-1} \rangle)$ if $b_i = 1$,
   $eval_{ev}(\langle c, \mathbf{r} \rangle, \langle b_0, \ldots, b_{n-1} \rangle)$ if $b_i = 0$.

An EVBDD can represent both Boolean and arithmetic functions. For example, the *carry* and *sum* functions of a full adder represented by EVBDDs are shown in Figure 1 (a) and (b) where $carry = xy + yz + zx - 2xyz$ and $sum = x + y + z - 2xy - 2yz - 2xz + 4xyz$ are represented by arithmetic functions. The arithmetic function of $2 * carry + sum$ is shown in Figure 1 (c).

Note that in a BDD the function value is determined by the terminal node at the end of a path. In an EVBDD the function value is computed by summing the edge values (right edge values are 0) along the path. For example, the function value of $x = 1$, $y = 0$ and $z = 1$ on *carry* and *sum* are $0 + 0 + 0 + 1 = 1$ and $0 + 1 + 0 - 1 = 0$.

3 Disjunctive Decomposition

We start with a brief review of the Ashenhurst-Curtis and Roth-Karp methods for detecting simple disjunctive decompositions. We then show how to use the BDD representation to perform the same operation. All the functions considered in this section are fully specified. Incompletely specified functions will be discussed in Section 6.

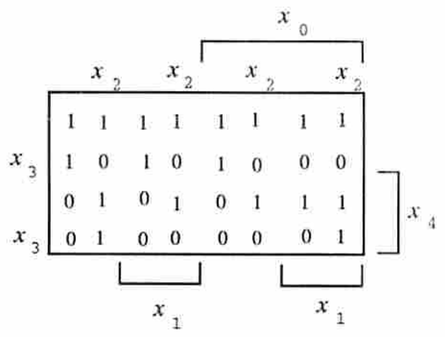


Figure 2: A decomposition chart

The Ashenhurst-Curtis method is based on an arrangement of the Karnaugh map where the rows correspond to the variables in the free set and the columns correspond to the variables in the bound set. The arrangement is referred to as a *decomposition chart.* A simple disjunctive decomposition exists if and only if the decomposition chart has at most two distinct columns.

Example 3.1 Let $f = x_0x_1x_2x_3 + x_0x_1x_2\bar{x}_3\bar{x}_4 + x_0x_1\bar{x}_2\bar{x}_4 + x_0\bar{x}_1x_2\bar{x}_4 + x_0\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_0x_1x_2\bar{x}_4 + \bar{x}_0x_1\bar{x}_2\bar{x}_3 + \bar{x}_0\bar{x}_1x_2x_3 + \bar{x}_0\bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$, the decomposition chart of $f(x_0, x_1, x_2, x_3, x_4)$ with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_3, x_4\}$ is shown in Figure 2. Since there are three distinct columns, namely 1100, 1011 and 1010, there is no simple disjunctive decomposition of f with respect to the given bound and free sets.

The Roth-Karp algorithm is based on the computation of compatible classes. Let f be a Boolean function with a bound set A_0 and a free set A_1, with $\mid A_0 \mid= i$ and $\mid A_1 \mid= n-i$. Let $B_b = B^i$ and $B_f = B^{n-i}$ where $B = \{0,1\}$. Two variables x_1 and x_2, $x_1 \in B_b$ and $x_2 \in B_b$ are said to be *compatible* if, for all $y \in B_f, f(x_1, y) = f(x_2, y)$; otherwise, they are said to be incompatible. Roth and Karp show that a function has simple disjunctive decomposition with respect to given bound and free sets if and only if B_b can be partitioned into $k \leq 2$ classes consisting of mutually compatible elements. When a function is completely specified, compatibility is an equivalence relation, and k is simply the number of equivalence classes.

The relation between a distinct column of a decomposition chart and a compatible class is bijective. The basic difference between these two methods is in the use of different function representations. One uses the Karnaugh map to represent a function while the other uses covers of the onset and offset for the function.

Our method for detecting simple disjunctive decomposition is based on the concept of a *cut_set* in BDD representation of Boolean functions. With respect to a given bound set $\{x_0, \ldots, x_{i-1}\}$ and free set $\{x_i, \ldots, x_{n-1}\}$ for a function f, we first construct a BDD $\mathbf{v}$ with the variable ordering $\{x_0, \ldots, x_{i-1}\} < \{x_i, \ldots, x_{n-1}\}$, then we compute the set $cut\_set(\mathbf{v}, i-1)$. If the cardinality of $cut\_set(\mathbf{v}, i-1)$ is less than or equal to 2 then f is simple disjunctive decomposable, otherwise it is not.

Definition 3.1 Given a BDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$,
$cut\_set(\mathbf{v}, level) = \{eval(\mathbf{v}, i) \mid 0 \leq i < 2^{level+1}\}$
where $0 \leq level < n$.

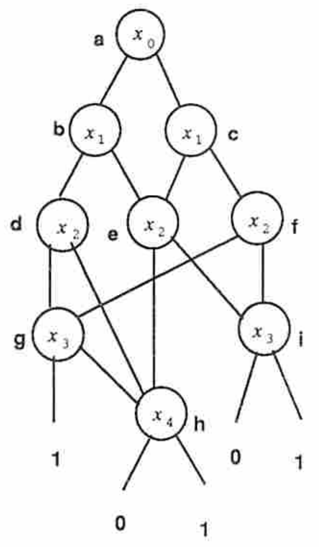


Figure 3: A BDD example

An alternative (and equivalent) definition is as follows. $cut\_set(\mathbf{v}, level)$ represents a set of BDD nodes $\mathbf{u}$ such that $\mathbf{u} \in descendant(\mathbf{v})$, $index(\mathbf{u}) > level$, and $\exists \mathbf{w} \in descendant(\mathbf{v})$ such that $index(\mathbf{w}) \leq level$ and either $child_l(\mathbf{w}) = \mathbf{u}$ or $child_r(\mathbf{w}) = \mathbf{u}$. The last constraint ensures that only the topmost $\mathbf{u}$'s with indices greater than $level$ are included in the set. Note that $level$ is an absolute value between 0 and $n-1$, and not a relative value with respect to the index of $\mathbf{v}$.

Example 3.2 The BDD representation of the function f in Example 3.1 is shown in Figure 3 which has $cut\_set(\mathbf{a}, 2) = \{\mathbf{g}, \mathbf{h}, \mathbf{i}\}$ and $cut\_set(\mathbf{d}, 2) = \{\mathbf{g}, \mathbf{h}\}$.

If the BDD is constructed as described above, then a node in the *cut_set* corresponds to a distinct column of the corresponding decomposition chart and a compatible class of B_b. For example, nodes **g**, **h** and **i** in Figure 3 correspond to columns 1011, 1010 and 1100 in Figure 2, respectively. Note that changing the variable ordering within the groups $\{x_0, \ldots, x_{i-1}\}$ and $\{x_i, \ldots, x_{n-1}\}$ will not change the size of cut_set.

In the following, we present the procedure $cut\_set(\mathbf{v}, level)$ to computes the *cut_set* as described above. When the variable ordering of a BDD does not correspond to a given bound set, we use *rotate* to move the bound variables to the top. The operation $rotate(\mathbf{v}, x)$ returns a new BDD with the new variable ordering π' such that $\pi'(0) = x$, $\pi'(i+1) = \pi(i)$ for $0 \leq i < index(x)$, $\pi'(i) = \pi(i)$ for $i > index(x)$.

```
$cut\_set(\mathbf{v}, level)$
{
  if $(index(\mathbf{v}) > level)$ return({ v });
  else return( $cut\_set(child_l(\mathbf{v}), level) \cup cut\_set(child_r(\mathbf{v}), level)$ );
}
```

```
$rotate(\mathbf{v}, x)$
{
    if ($index(\mathbf{v}) < index(x)$) {
      $\mathbf{l} = rotate\_2(\mathbf{v},\ x,\ 'l')$;
      $\mathbf{r} = rotate\_2(\mathbf{v},\ x,\ 'r')$;
      return($new\_bdd(x, \mathbf{l}, \mathbf{r})$);
    }
    else return($\mathbf{v}$);
}

$rotate\_2(\mathbf{v}, x, flag)$
{
      if ($index(\mathbf{v}) < index(x)$) {
           $\mathbf{l} = rotate\_2(child_l(\mathbf{v}), x, flag)$;
           $\mathbf{r} = rotate\_2(child_r(\mathbf{v}), x, flag)$;
           return($new\_bdd(variable(\mathbf{v}), \mathbf{l}, \mathbf{r})$);
      }
      else if ($index(\mathbf{v}) == index(x)$) {
           if ($flag ==\ 'l'$) return($child_l(\mathbf{v})$);
           else return($child_r(\mathbf{v})$);
      }
      else return($\mathbf{v}$)
}
```

The computation of the *cut_set* in place will be discussed in Section 10.

We show how to perform the disjunctive decomposition

$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{m-1}(x_0, \ldots, x_{l-1}), x_l, \ldots, x_{n-1})$
directly on a BDD. Given a function f and the bound set $\{x_0, \ldots, x_{l-1}\}$, we first construct a BDD $\mathbf{v}$ with variable ordering $x_0 < \ldots < x_{n-1}$ and compute the $cut\_set(\mathbf{v}, l-1)$ (Figure 4 (a)). We then encode each node $\mathbf{u}$ in the *cut_set* by an m-bit integer where $2^{m-1} \leq k = | cut\_set(\mathbf{v}, l-1) | < 2^m$.

To construct a BDD $\mathbf{v}_{f'}$ corresponding to function f', we replace the top part of $\mathbf{v}_f$ (above the cutset) by a new top on variables $y_0, \ldots, y_{m-1}$ such that $eval(\mathbf{v}_{f'}, j) = \mathbf{u}_j$ for $0 \leq j < k-1$, $eval(\mathbf{v}_{f'}, j) = \mathbf{u}_{k-1}$ for $k-1 \leq j < 2^m$ (Figure 4 (b)). Note that this is an arbitrary input encoding which is not unique. Different encodings will result in different f' functions.

To construct a BDD $\mathbf{v}_{g_p}, 0 \leq p < m$, corresponding to function $g_p(x_0, \ldots, x_{l-1})$, we replace every node $\mathbf{u}_j \in cut\_set$ in $\mathbf{v}$ by terminal node whose value is b_{j_p} (p^{th} bit from the right of integer j, Figure 4 (c)). For example, $\mathbf{u}_1$ is replaced by terminal node **1** in the construction of $\mathbf{v}_{g_{m-1}}$ and by terminal node **0** in the construction of other $\mathbf{v}_{g_p}$'s.

The above two operations are implemented through algorithms *decomp_f* and *decomp_g* which are called by algorithm *decompose*.

Lemma 3.1 Given a BDD $\mathbf{v}_f$ with variable ordering $x_0, \ldots, x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$ and $cut\_set(\mathbf{v}_f, l) = \{\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}\}$, $2^{m-1} < k \leq 2^m$, the algorithm *decompose* returns $m+1$

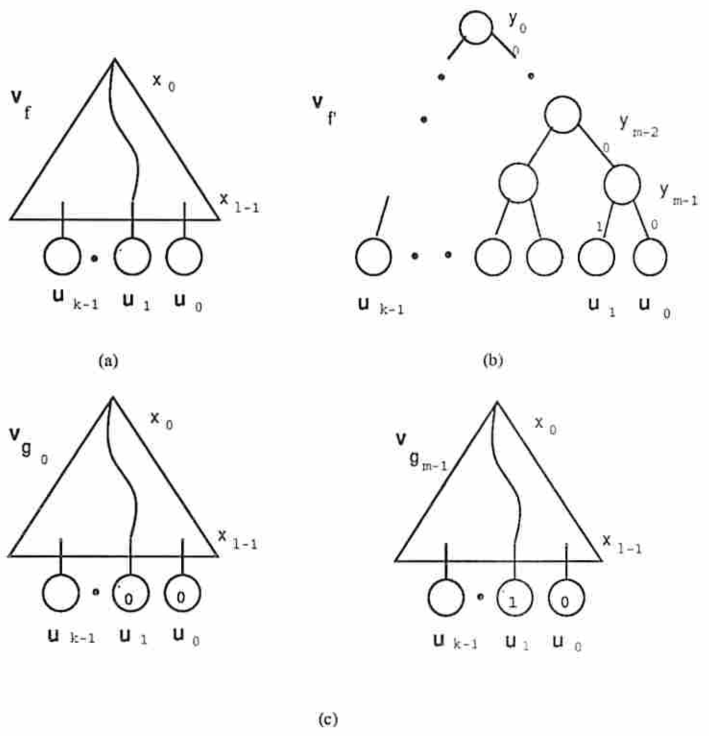


Figure 4: Disjunctive decomposition

BDDs $\mathbf{v}_{f'}, \mathbf{v}_{g_0}, \ldots, \mathbf{v}_{g_{m-1}}$ such that

$$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{m-1}(x_0, \ldots, x_{l-1}), x_l, \ldots, x_{n-1})$$

where $f', g_0, \ldots, g_{m-1}$ are the functions denoted by $\mathbf{v}_{f'}, \mathbf{v}_{g_0}, \ldots, \mathbf{v}_{g_{m-1}}$, respectively.

Proof: Consider the behavior of an input pattern $\langle b_0, \ldots, b_{l-1}\rangle$ on $\mathbf{v}_f, \mathbf{v}_{f'}$ and $\mathbf{v}_{g_i}$'s. Suppose $\mathbf{u}_j$ is the node we reach in $\mathbf{v}_f$ using the input pattern, that is, $eval(\mathbf{v}_f, \langle b_0, \ldots, b_{l-1}\rangle) = \mathbf{u}_j$, where $\mathbf{u}_j = f(b_0, \ldots, b_{l-1}, x_l, \ldots, x_{n-1})$. Since $\mathbf{u}_j$ has been replaced by the i^{th} bit of j in $\mathbf{v}_{g_i}$, $eval(\mathbf{v}_{g_i}, \langle b_0, \ldots, b_{l-1}\rangle) = b_{j_i}$, that is, $g_i(b_0, \ldots, b_{l-1}) = b_{j_i}$. Because of the way we construct $\mathbf{v}_{f'}$, $eval(\mathbf{v}_{f'}, \langle b_{j_0}, \ldots, b_{j_{m-1}}\rangle) = \mathbf{u}_j$, that is,
$f'(b_{j_0}, \ldots, b_{j_{m-1}}, x_l, \ldots, x_{n-1}) = \mathbf{u}_j = f(b_0, \ldots, b_{l-1}, x_l, \ldots, x_{n-1})$.
Thus, $f'(g_0(b_0, \ldots, b_{l-1}), \ldots, g_{m-1}(b_0, \ldots, b_{l-1}), x_l, \ldots, x_{n-1}) = f(b_0, \ldots, b_{l-1}, x_l, \ldots, x_{n-1})$.

□

```
$decompose(\mathbf{v}_f, cutset)$        /* $cutset = \{\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}\}, 2^{m-1} < k \leq 2^m$ */
{
    $\mathbf{v}_{f'} = decomp\_f(cutset, \{y_{m-1}, \ldots, y_0\})$;
    for (i = 0; i < m; i++)
        $\mathbf{v}_{g_i} = decomp\_g(\mathbf{v}_f, cutset, i)$;
    return $\mathbf{v}_{f'}$ and $\mathbf{v}_{g_i}$'s;
}
```

```
$decomp\_f(cutset, ids)$
  $cutset$: an array of BDD nodes with $2^{m-1} < length(cutset) \leq 2^m$
  return: a BDD $\mathbf{v}_{f'}$ with variables $y_0, \ldots, y_{m-1}, x_l, \ldots, x_{n-1}$
{
  $ptr\_id = 0$;
  while ($length(cutset) > 1$) {   /* do until $cutset$ becomes a single (root) node */
    $id = ids[ptr\_id{+}{+}]$;
    $i = 0$;        /* pointer to $cutset$ */
    $ptr = 0$;        /* pointer to $newset$ */
    while ($i < length(cutset)$) {
      if ($i == length(cutset) - 1$) {   /* the last element of $cutset$ */
        $newset[ptr{+}{+}] = cutset[i]$;   /* just move it to $newset$ */
        $i{+}{+}$;        /* $i$ is increased by 1 */
      }
      else {   /* create a new node with new variable $y_{id}$ */
        $newset[ptr{+}{+}] = new\_bdd(id, cutset[i+1], cutset[i])$;
        $i{+}{=}2$;     /* $i$ is increased by 2 */
      }
    }
    for ($j = 0; j < ptr; j{+}{+}$)        /* move nodes from $newset$ to $cutset$ */
      $cutset[j] = newset[j]$;
  }
  return($cutset[0]$);        /* $cutset[0]$ contains $\mathbf{v}_{f'}$ */
}
```

```
$decomp\_g(\mathbf{v}, cutset, i)$
  $\mathbf{v}$: a BDD
  $cutset$: an array of BDD nodes with $2^{m-1} < length(cutset) \leq 2^m$
  $i$: an integer between 0 and $m - 1$
  return: $\mathbf{v}_{g_i}$
{
  for ($j = 0; j < length(cutset); j{+}{+}$) {
    $\mathbf{v}_j = cutset[j]$;
    if ($bit(i, j) == 0$)     /* i$^{th}$ bit from right of $j$ */
        $\mathbf{v} = replace\_node(\mathbf{v}, \mathbf{v}_j, \mathbf{0})$;   /* replace $\mathbf{v}_j$ in $\mathbf{v}$ by $\mathbf{0}$ */
    else $\mathbf{v} = replace\_node(\mathbf{v}, \mathbf{v}_j, \mathbf{1})$;   /* replace $\mathbf{v}_j$ in $\mathbf{v}$ by $\mathbf{1}$ */
  }
  return $\mathbf{v}$;       /* as $\mathbf{v}_{g_i}$ */
}
```

Example 3.3 As an example of how *decompose* works, consider the BDD shown in Figure 3. Let the bound set be $\{x_0, x_1, x_2\}$. Therefore $l = 3$ and $cut\_set(\mathbf{v}_f, 2) = \{\mathbf{g}, \mathbf{h}, \mathbf{i}\} = \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$, and $m = 2$. We can thus construct a decomposition of the form $f'(g_0(x_0, x_1, x_2),$

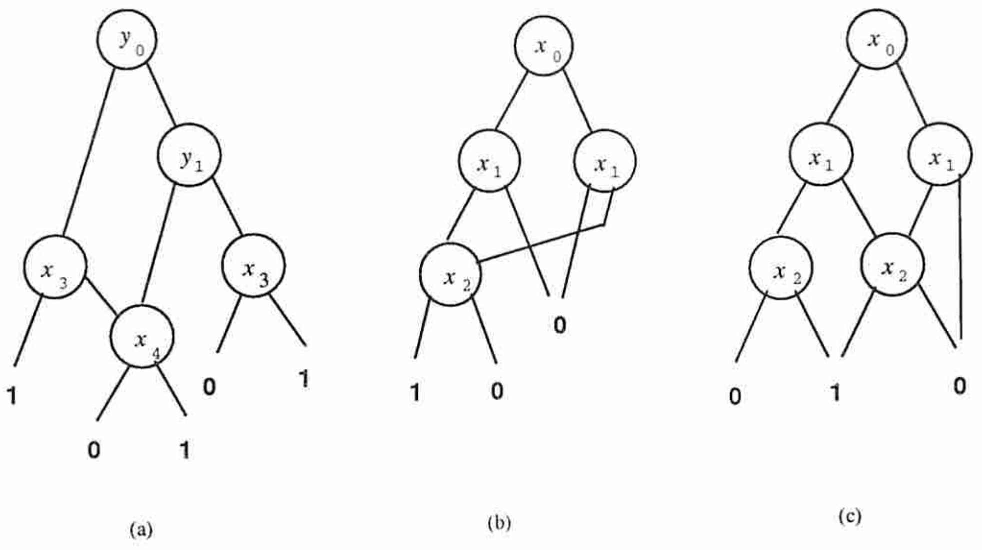


Figure 5: (a) $\mathbf{v}_{f'}$ (b) $\mathbf{v}_{g_0}(y_0)$ (c) $\mathbf{v}_{g_1}(y_1)$

$g_1(x_0, x_1, x_2), x_3, x_4)$. Consider the encoding of $\mathbf{v}_0 = 00, \mathbf{v}_1 = 01$, and $\mathbf{v}_2 = 10$. The BDD for $\mathbf{v}_{g_0}$ is obtained by replacing the nodes $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ with the terminal node 0, 0, 1, respectively. The BDD for $\mathbf{v}_{g_1}$ is obtained by replacing the nodes $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ with the terminal node 0,1,0, respectively (Figure 5).

Lemma 3.2 Given a BDD $\mathbf{v}$ with variable ordering $x_0, \ldots, x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, the decomposition $f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{j-1}(x_0, \ldots, x_{l-1}), x_l, \ldots, x_{n-1})$ exists if and only if there exists some $m \leq j$ such that $2^{m-1} < | \, cut\_set(\mathbf{v}, l-1) \, | \leq 2^m$.

Proof: **Necessity**: Since BDD is a canonical representation of Boolean functions, every node $\mathbf{u}$ in $cut\_set(\mathbf{v}, l-1)$ represents a distinct function. For $2^{m-1} < | \, cut\_set(\mathbf{v}, l-1) \, |$, the minimum j required to encode every element of $cut\_set(\mathbf{v}, l-1)$ is m.
Sufficiency: Follows from the *decompose* algorithm. □

4 Non-disjunctive Decomposition

Before describing how to perform non-disjunctive decomposition based on the BDD representation, we extend the concept of *cut_set* in the following definition.

Definition 4.1 Given a BDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$,
$cut\_set\_nd(\mathbf{v}, start, level, i) = \{eval(\mathbf{w}, i) \mid \mathbf{w} \in cut\_set(\mathbf{v}, start - 1)\}$,
where $0 \leq start \leq level < n, 0 \leq i < 2^{level - start + 1}$.

An alternative (and equivalent) definition is as follows. The set $cut\_set\_nd(\mathbf{v}, s, l, i)$ is the set of BDD nodes $\mathbf{u}$ such that

1. $\mathbf{u} \in descendant(\mathbf{v})$,

2. $index(\mathbf{u}) > l$,

3. $\mathbf{u}$ is in the path $\langle x_0, \ldots, x_{s-1}, b_s, \ldots, b_l, x_{l+1}, \ldots, x_{n-1} \rangle$, where $2^{l-s}b_s + \ldots + 2^0 b_l = i$,

4. $\exists \mathbf{w} \in descendant(\mathbf{v})$ such that $index(\mathbf{w}) \leq l$ and either $child_l(\mathbf{w}) = \mathbf{u}$ or $child_r(\mathbf{w}) = \mathbf{u}$.

It can be seen that $cut\_set(\mathbf{v}, level)$ is a special case of the above definition, that is, $cut\_set(\mathbf{v}, level) = cut\_set\_nd(\mathbf{v}, level, level, 0) \cup cut\_set\_nd(\mathbf{v}, level, level, 1)$.

Example 4.1 The BDD in Figure 3 has $cut\_set\_nd(\mathbf{a}, 1, 1, 0) = \{\mathbf{e}, \mathbf{f}\}$, $cut\_set\_nd(\mathbf{a}, 1, 1, 1) = \{\mathbf{d}, \mathbf{e}\}$, $cut\_set\_nd(\mathbf{a}, 1, 2, 0) = \{\mathbf{i}\}$, $cut\_set\_nd(\mathbf{a}, 1, 2, 1) = \{\mathbf{g}, \mathbf{h}\}$, $cut\_set\_nd(\mathbf{a}, 1, 2, 2) = \{\mathbf{h}, \mathbf{i}\}$, and $cut\_set\_nd(\mathbf{a}, 1, 2, 3) = \{\mathbf{g}, \mathbf{h}\}$.

```
$cut\_set\_nd(\mathbf{v}, s, l, i)$
{
  if $(index(\mathbf{v}) < s)$
    return($cut\_set\_nd(child_l(\mathbf{v}), s, l, i) \cup cut\_set\_nd(child_r(\mathbf{v}), s, l, i)$);
  else if $(s \leq index(\mathbf{v}) \&\& index(\mathbf{v}) \leq l)$ {
    $k = 2^{l-index(\mathbf{v})}$;
    if $(k \leq i)$
      return($cut\_set\_nd(child_l(\mathbf{v}), s, l, i-k)$);
    else return($cut\_set\_nd(child_r(\mathbf{v}), s, l, i)$);
  }
  else return($\{\mathbf{v}\}$);
}
```

Given a function f, the non-disjunctive decomposition
$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_s, \ldots, x_{l-1}), g_{m-1}(x_0, \ldots, x_s, \ldots, x_{l-1}), x_s, \ldots x_{l-1}, \ldots, x_{n-1})$ is carried out in the following way.

We first construct a BDD $\mathbf{v}_f$ with variable ordering $x_0 < \ldots < x_{n-1}$ and compute the $cut\_set\_nd(\mathbf{v}_f, s, l, i)$ for $0 \leq i < 2^{l-s+1}$ (Figure 6 (a)). Let the size of the largest $cut\_set\_nd(\mathbf{v}_f, s, l, i)$ ($cutset_i$ for short) be k. The construction of $\mathbf{v}_{f'}$ as shown in Figure 6 (b) (where $q = 2^{l-s+1} - 1$) is carried out in two steps. First, we construct $\mathbf{v}_j, 0 \leq j < k$, such that $eval(\mathbf{v}_j, i) = \mathbf{u}_{j,i}$ where $\mathbf{u}_{j,i}$ is the j^{th} element in $cutset_i$ (or the last element of $cutset_i$ if $j >| cutset_i |$.) Second, construct $\mathbf{v}_{f'}$ such that $eval(\mathbf{v}_{f'}, j) = \mathbf{v}_j$ for $0 \leq j < k-1$ and $eval(\mathbf{v}_{f'}, j) = \mathbf{v}_{k-1}$ for $k-1 \leq j < 2^m$.

To construct $\mathbf{v}_{g_p}, 0 \leq p < m-1$, we replace each node $\mathbf{u}_{j,i}$ (j^{th} node of $cutset_i$) from $\mathbf{v}_f$ by the terminal node whose value is b_{j_p} where b_{j_p} is the p^{th} bit from the right of integer j.

Note that, a node $\mathbf{u}$ may be the i^{th} element of $cutset_a$ and j^{th} element of $cutset_b$ which requires different encodings for $\mathbf{u}$. This will not cause a problem because we can first duplicate the node $\mathbf{u}$ and then assign each copy a different encoding. An alternative statement is that if paths p_1 and p_2 both end at node $\mathbf{u}$ and require different encodings b_1 and b_2, then we let p_1 end at terminal node $\mathbf{b}_1$ and p_2 end at terminal node $\mathbf{b}_2$.

The above algorithms are implemented through *nd_decompose*, *nd_decomp_f* and *nd_decomp_g*.

```
$nd\_decompose(\mathbf{v}_f, cutsets)$
$cutsets : \{cutset_i \mid 0 \leq i < 2^{level-start+1}\};$
{
  $\mathbf{v}_{f'} = nd\_decomp\_f(cutsets, \{x_{l-1}, \ldots, x_s\}, \{y_{m-1}, \ldots, y_0\});$
  for $(p = 0; p < m; p++)$
      $\mathbf{v}_{g_p} = nd\_decomp\_g(\mathbf{v}_f, cutsets, p);$
  return $\mathbf{v}_{f'}$ and $\mathbf{v}_{g_p}$'s;
}
```

```
$nd\_decomp\_f(cutsets, idx, idy)$
$cutsets : \{cutset_i \mid 0 \leq i < 2^{level-start+1}\}$ with $max\{\mid cutset_i \mid\} = k;$
$cutset_{j,i} : j^{th}$ element of $cutset_i$, or the last element of $cutset_i$ if $j > \mid cutset_i \mid;$
$idx : \{x_{l-1}, \ldots, x_s\};$
$idy : \{y_{m-1}, \ldots, y_0\};$
return: a BDD $\mathbf{v}_{f'}$ with variables $y_0, \ldots, y_{m-1}, x_s, \ldots, x_{l-1}, \ldots, x_{n-1};$
{
  for $(j = 0; j < k; j++)$
    $\mathbf{v}_j = decomp\_f(\{cutset_{j,i} \mid 0 \leq i < 2^{l-s+1}\}, idx);$
  $\mathbf{v}_{f'} = decomp\_f(\{\mathbf{v}_j \mid 0 \leq j < k\}, idy);$
  return $\mathbf{v}_{f'}$
}
```

```
$nd\_decomp\_g(\mathbf{v}_f, cutsets, p)$
return: $\mathbf{v}_{g_p};$
{
    for $(r = 0; r < 2^{l-1}; r++)$ {
        if $(eval(\mathbf{v}_f, r) == cutset_{j,i})$
        let $eval(\mathbf{v}_{g_p}, r) = bit(p, j)$
    }
    return $\mathbf{v}_{g_p};$
}
```

Lemma 4.1 Given a BDD $\mathbf{v}_f$ with variable ordering $x_0, \ldots, x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, and $k = max\{\mid cut\_set\_nd(\mathbf{v}_f, s, l, i) \mid, 0 \leq i < 2^{l-s+1}\}$, $2^{m-1} < k \leq 2^m$, the algorithm *nd_decompose* returns $m+1$ BDDs $\mathbf{v}_{f'}, \mathbf{v}_{g_0}, \ldots, \mathbf{v}_{g_{m-1}}$ such that
$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{m-1}(x_0, \ldots, x_{l-1}), x_{s-1}, \ldots, x_{l-1}, \ldots, x_{n-1})$
where $f', g_0, \ldots, g_{m-1}$ are the functions denoted by $\mathbf{v}_{f'}, \mathbf{v}_{g_0}, \ldots, \mathbf{v}_{g_{m-1}}$, respectively.

Proof: Consider the behavior of an input pattern $\langle b_0, \ldots, b_s, \ldots, x_{l-1}\rangle$ on $\mathbf{v}_f, \mathbf{v}_{f'}$ and $\mathbf{v}_{g_p}$'s. Let $eval(\mathbf{v}_f, \langle b_0, \ldots, b_s, \ldots, b_{l-1}\rangle) = \mathbf{u}_{j,i}$, where $i = 2^{l-s+1}b_s + \ldots + 2^0 b_{l-1}$. Since $\mathbf{u}_{j,i}$ has been replaced by b_{j_p} in $\mathbf{v}_{g_p}$, we have $eval(\mathbf{v}_{g_p}) = b_{j_p}$. Then from $eval(\mathbf{v}_{f'}, \langle b_{j_0}, \ldots, b_{j_{m-1}}\rangle) = \mathbf{v}_j$ and $eval(\mathbf{v}_j, i) = \mathbf{u}_{j,i} = eval(\mathbf{v}_j, \langle b_s, \ldots, b_{l-1}\rangle)$, we have
$eval(\mathbf{v}_{f'}, \langle b_{j_0}, \ldots, b_{j_{m-1}}, b_s, \ldots, b_{l-1}\rangle) = \mathbf{u}_{j,i} = eval(\mathbf{v}_f, \langle b_0, \ldots, b_s, \ldots, b_{l-1}\rangle)$. $\square$

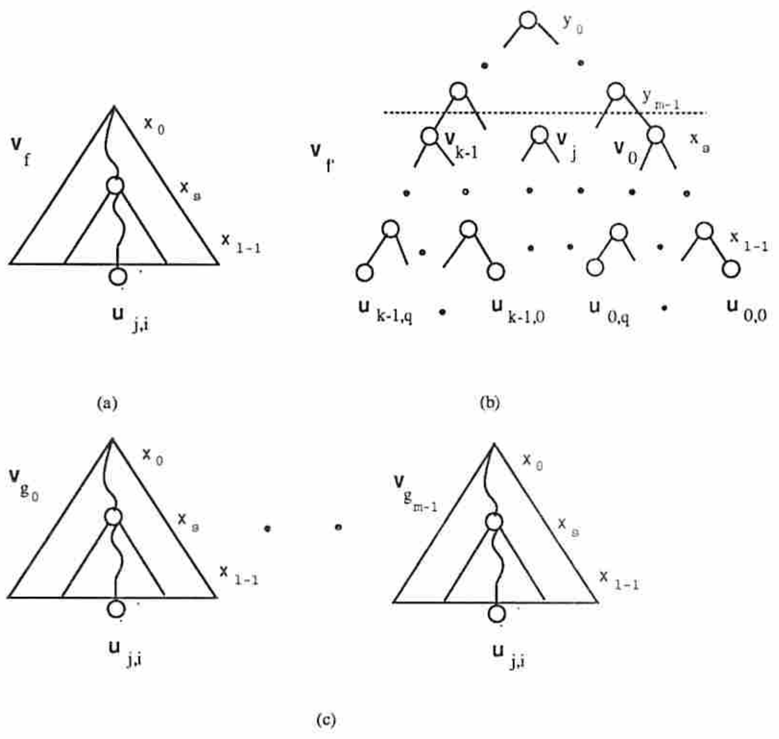


Figure 6: Non-disjunctive decomposition

Example 4.2 One possible non-disjunctive decomposition of the BDD in Figure 3 with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_2, x_3, x_4\}$ is shown in Figure 7. In this decomposition, we use the following coding: $\{\mathrm{i} = \mathbf{v}_{0,0}, \mathrm{h} = \mathbf{v}_{0,1}\} = cut\_set\_nd(\mathbf{v}_f, 2, 2, 0)$ and $\{\mathrm{g} = \mathbf{v}_{1,0}, \mathrm{h} = \mathbf{v}_{1,1}\} = cut\_set\_nd(\mathbf{v}_f, 2, 2, 1)$.

Lemma 4.2 Given a BDD $\mathbf{v}$ with variable ordering $x_0, \ldots, x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, the non-disjunctive decomposition
$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{j-1}(x_0, \ldots, x_{l-1}), x_s, \ldots, x_{l-1}, \ldots, x_{n-1})$
exists if and only if there exist $m \leq j$ such that $2^{m-1} < max\{\mid cut\_set\_nd(\mathbf{v}, s, l, i) \mid, 0 \leq i < 2^{l-s+1}\} \leq 2^m$.

Proof: Necessity: The maximum of $\mid cut\_set\_nd(\mathrm{v}, s, l, i) \mid$, $0 \leq i < 2^{l-s+1}$, is the minimum number of $\mathbf{v}_p$s, $0 \leq p < k$, required in algorithm *nd_decomp_f*. Thus, the minimum j required to encode each $\mathbf{v}_p$ is m if $k > 2^{m-1}$.
Sufficiency: Follows from the algorithm *nd_decompose*. □

5 Multiple Output Functions

Given a vector of Boolean functions $f_0, \ldots, f_{m-1}$ on n variables, we can not extend the concept of *cut_set* by taking the set union of the *cut_sets* for the individual functions. To see this, consider the two extreme cases $\bigcup_{i=0}^{m-1} cut\_set(\mathrm{v}_i, 0)$ (bound set size 1) and

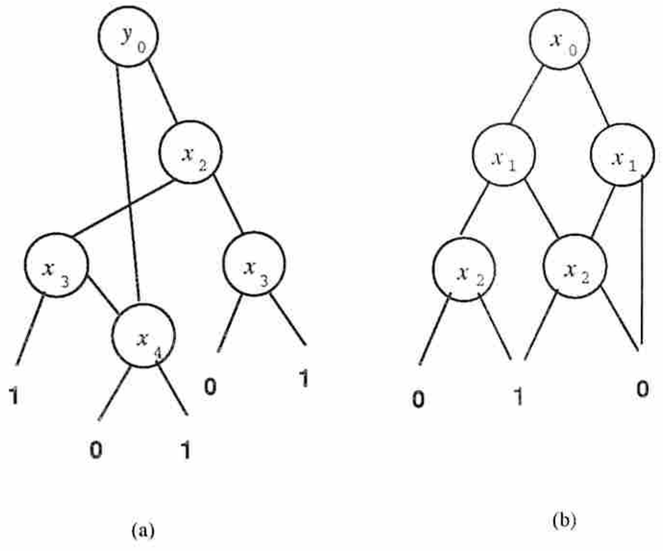


Figure 7: (a) $\mathbf{v}_{f'}$ (b) $\mathbf{v}_g$

$\bigcup_{i=0}^{m-1} cut\_set(\mathbf{v}_i, n-1)$ (bound set size n). The cardinality of the former might be greater than 2. This implies that more than two distinct functions must be implemented by a single variable which is not possible. On the other hand, the *cut_set* of the latter is always $\{0, 1\}$. This implies that m Boolean functions could be implemented by only one output line which is again not possible. This is because even when a BDD node is shared by i^{th} and j^{th} functions, it should be treated differently.

Definition 5.1 Given a vector of BDDs $\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1}\rangle$ representing Boolean functions $F = \langle f_0, \ldots, f_{m-1}\rangle$ on n variables. The *multiple-output cut_set* of F is the following set.
$cut\_set\_mo(\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1}\rangle, level) = \{\langle eval(\mathbf{v}_0, i), \ldots, eval(\mathbf{v}_{m-1}, i)\rangle \mid 0 \leq i < 2^{level-1}\}$.

From the above definition, we can determine the upper bound and lower bound of the cardinality of *cut_set_mo* based on the *cut_sets* of individual functions. Let $cutset_i = \mid cut\_set(\mathbf{v}_i, l-1) \mid$, that is, the size of *cut_set* for output function f_i. The maximum number of different vectors that can be generated is $\Pi_{i=0}^{m-1} cutset_i$. Thus, $\Pi_{i=0}^{m-1} cutset_i$ is an upper bound. It is easy to see that the cardinality of *cut_set_mo* can not exceed 2^l for bound set size l. Thus, 2^l is also an upper bound. The minimum number of vectors we can generate for *cut_set_mo* is the maximum of $cutset_i$ for $0 \leq i < m$. Thus, the cardinality of multiple-output *cut_set* is in the following range:
$max\{cutset_i\} \leq \mid cut\_set\_mo \mid \leq min\{\Pi_{i=0}^{m-1} cutset_i, 2^l\}$.

Example 5.1 Figure 8 shows a two-output Boolean function where $f_0 = x \oplus y \oplus z$ and $f_1 = x \oplus z$. Let the bound set be $\{x, y\}$, then the *cut_sets* of f_0 and f_1 are $\{\mathbf{a}, \mathbf{b}\}$ and $\{\mathbf{a}, \mathbf{b}\}$ while $cut\_set\_mo = \{\langle \mathbf{a}, \mathbf{a}\rangle, \langle \mathbf{b}, \mathbf{a}\rangle, \langle \mathbf{b}, \mathbf{b}\rangle, \langle \mathbf{a}, \mathbf{b}\rangle\}$.

The concept of *cut_set* is based on the canonical property of the BDDs. The cardinality of a *cut_set* represents the number of distinct functions that need to be implemented. EVBDDs are canonical representations of arithmetic functions, thus the cardinality of a

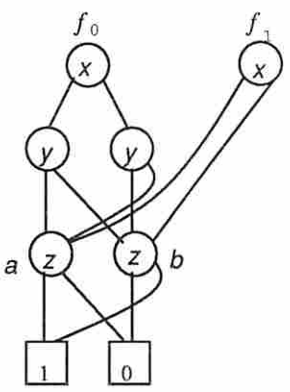


Figure 8: A two-output Boolean function

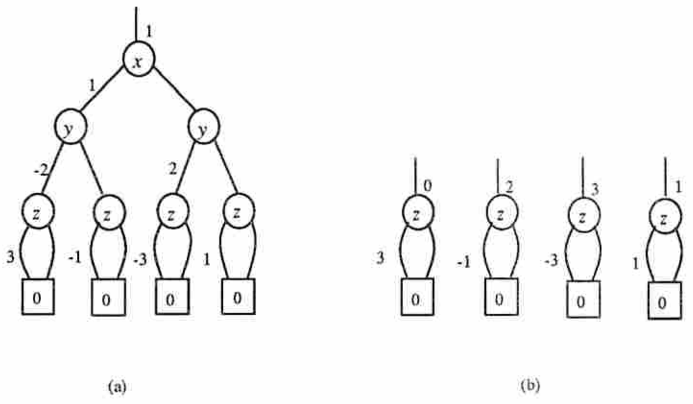


Figure 9: Example of *cut_set_ev*

cut_set in EVBDD also represents the number of distinct arithmetic functions that need to be implemented. The difference between these two representations is that each BDD node represents a Boolean function while each EVBDD node represents an arithmetic function (or a multiple output Boolean function with respect to some encoding). We therefore use EVBDDs to efficiently compute *cut_set_mo* as follows. F is encoded as $2^{m-1}f_{m-1} + \ldots + 2^0 f_0$ and then implemented by a single EVBDD. The *cut_set* of this single output EVBDD is equivalent to the *cut_set_mo* of the m BDDs.

Definition 5.2 Given an EVBDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$,

$$cut\_set\_ev(\mathbf{v}, level) = \{eval_{ev}(\mathbf{v}, i) \mid 0 \leq i < 2^{level+1}\}$$

where $0 \leq level < n$.

Example 5.2 The multiple output function in the previous example (Figure 8) represented in EVBDD is shown in Figure 9 (a). Figure 9 (b) is the $cut\_set\_ev(\mathbf{v}, 1)$.

Similarly, the *cut_sets* for non-disjunctive decomposition of multiple-output Boolean functions and arithmetic functions are defined as the follows.

Definition 5.3 Given a vector of BDDs $\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1} \rangle$ representing Boolean functions $F = \langle f_0, \ldots, f_{m-1} \rangle$ on n variables.

$cut\_set\_nd\_mo(\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1}\rangle, start, level, i) =$
$\{\langle eval(\mathbf{w}_0, i), \ldots, eval(\mathbf{w}_{m-1}, i)\rangle \mid \langle \mathbf{w}_0, \ldots, \mathbf{w}_{m-1}\rangle \in cut\_set\_mo(\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1}\rangle, start - 1)\}$,
where $0 \leq start \leq level < n, 0 \leq i < 2^{level-start+1}$.

Definition 5.4 Given an EVBDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$,
$cut\_set\_nd\_ev(\mathbf{v}, start, level, i) = \{eval_{ev}(\mathbf{w}, i) \mid \mathbf{w} \in cut\_set\_ev(\mathbf{v}, start - 1)\}$,
where $0 \leq start \leq level < n, 0 \leq i < 2^{level-start+1}$.

6 Incompletely Specified Functions

When functions are incompletely specified, the detection of decomposability becomes very complicated. For example, the compatibility in the Roth-Karp method is no longer an equivalence relation. The determination of k compatible classes then requires solving the minimum clique covering problem for the compatibility graph which is NP-hard [5].

A similar task has to be performed on BDDs. We first extend BDDs to include a third terminal node **dc** to represent the constant function *dc*. Next, we compute the *cut_set* as before. Since each node in *cut_set* may represent an incompletely specified function, we need to compute the compatibility between any two nodes in the *cut_set* so that a minimal k can be found. The determination of compatibility between two BDD nodes, or the compatibility between their corresponding functions, can be carried out by algorithm *is_compatible*. After this step, the construction of compatibility graph and the computation of minimum clique cover is the same as in the Roth-Karp algorithm.

```
$is\_compatible(\mathbf{f}, \mathbf{g})$        /* f and g are BDDs with dc */
{
  if (f == dc || g == dc || f == g)
    return 1;
  if (f == 0 && g == 1 || f == 1 && g == 0)
    return 0;
  if ($index(\mathbf{f}) \geq index(\mathbf{g})$) {
    $\mathbf{g}_l = child_l(\mathbf{g})$;
    $\mathbf{g}_r = child_r(\mathbf{g})$;
  }
  else { $\mathbf{g}_l = \mathbf{g}_r = \mathbf{g}$; }
  if ($index(\mathbf{f}) \leq index(\mathbf{g})$) {
    $\mathbf{f}_l = child_l(\mathbf{f})$;
    $\mathbf{f}_r = child_r(\mathbf{f})$;
  }
  else { $\mathbf{f}_l = \mathbf{f}_r = \mathbf{f}$; }
  return($is\_compatible(\mathbf{f}_l, \mathbf{g}_l)$ && $is\_compatible(\mathbf{f}_r, \mathbf{g}_r)$);
}
```

7 Decomposibility

In the previous sections, we showed how to compute *cut_set* by using *rotate*. In this section, we first show how to compute the *cut_sets* for all bound sets by using EVBDD representation. We then show how to compute the *cut_set* in place for a given bound set.

Our method is based on the encoding of columns of the decomposition chart. Decomposability is determined by the number of distinct columns (i.e., bit vectors corresponding to the columns.) Therefore, this is equivalent to computing the cardinality of a set of bit vectors. By encoding these bit vectors, we may transform the process to computing the cardinality of a set of integer vectors. In the same way that a bit vector corresponds to a BDD node, an integer vector corresponds to an EVBDD node. For example, the column in Figure 10 (a) corresponds to the EVBDD node (not reduced for readability) **a** in Figure 10 (e), while the column in Figure 10 (d) corresponds to node **d** in Figure 10 (g).

Now we show how to include (or exclude) a variable into (or from) a bound set. Consider the decomposition chart in Figure 10 (a) which has an empty bound set. Including x in the bound set corresponds to the decomposition chart shown in Figure 10 (b). To exclude x from the bound set, or to avoid x being included in the bound set, we first rearrange the minterms such that $\bar{x}u$ and xu become adjacent where $u = \{yz, y\bar{z}, \bar{y}z, \bar{y}\bar{z}\}$. This is shown in Figure 10 (c). We then encode the pairs of minterms $\bar{x}u$ and xu as $2xu + \bar{x}u$ which will result in the decomposition of Figure 10 (d). The effect of this operation is that variables y and z can be later included in the bound set but variable x is permanently excluded from the bound set. By repeated application of the two operations *include_in_set* and *exclude_from_set*, we can generate the decomposition charts for all bound sets.

The inclusion or exclusion of a variable in a bound set can be performed on EVBDDs without having to move the variable to the top. First, to include the top most variable in the bound set, we collect its left and right children to form a set. For example, to include x in the bound set, we form $\{child_l(\mathbf{a}), child_r(\mathbf{a})\} = \{\mathbf{b}, \mathbf{c}\}$. This is shown in Figures 10 (e) and 10 (f). To exclude the top most variable from the bound set we perform the EVBDD arithmetic operation $2 * child_l(\mathbf{a}) + child_r(\mathbf{a})$. This would be $2 * \mathbf{c} + \mathbf{b} = \mathbf{d}$ as shown in Figure 10 (g). Figures 11 (a) and 11 (b) show how the variable y is included and excluded from the bound set, respectively. Note that the encoding now is $4yu + \bar{y}u, u = \{z, \bar{z}\}$.

After the application of the above two operations, we determine the cardinality of *cut_set* for each bound set. The number of distinct columns in Figure 10 (b) is 2 and the corresponding *cut_set* in Figure 10 (f) is $| \{\mathbf{c}, \mathbf{b}\} | = 2$. The number of distinct columns in Figure 10 (d) is 1 and the size of the corresponding *cut_set* in Figure 10 (g) is $| \{\mathbf{d}\} | = 1$.

Algorithm *decomp_all* computes the cardinality of *cut_set* for every bound set. *decomp_all* should initially be called with $var\_set = \phi$, $node\_set = \{root\_node\}$, and $level = 0$. The routine returns the set $\{\langle b, k \rangle \mid b$ is a bound set, k is the cardinality of the *cut_set* of b $\}$. Subroutines *include_in_set* and *exclude_from_set* perform the variable inclusion and exclusion operations.

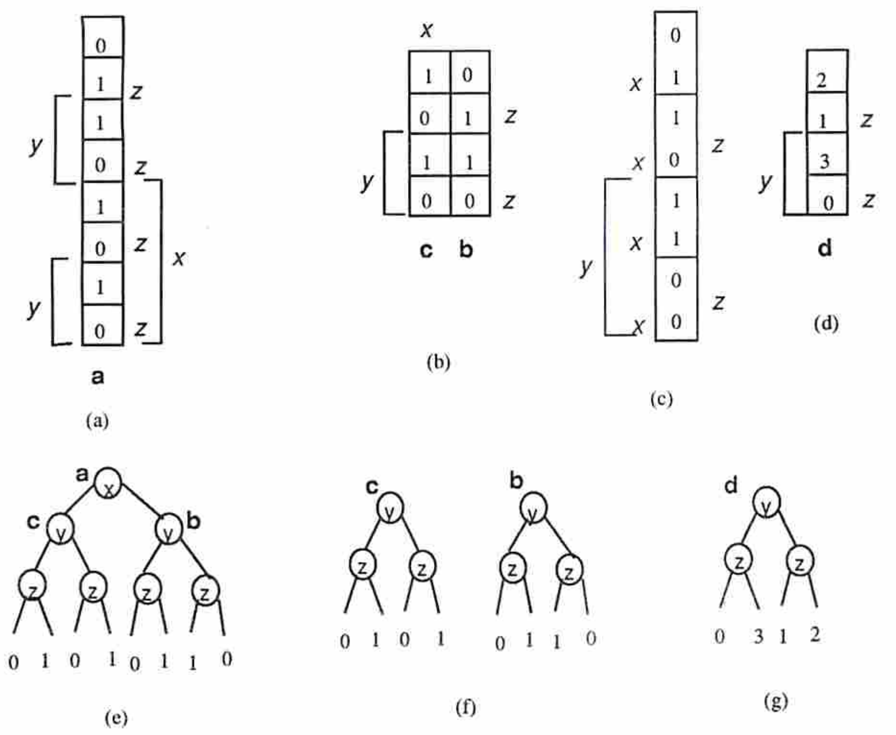


Figure 10: Correspondence between columns and EVBDD nodes

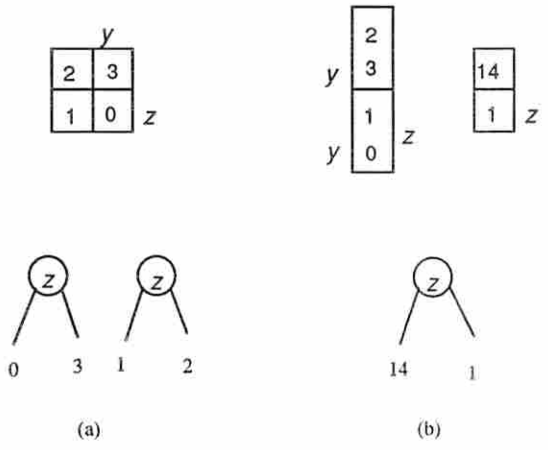


Figure 11: Continued example

```
$decomp\_all(var\_set, node\_set, level)$
{
    if $(level == n)$
        return($\{\langle var\_set, \mid node\_set \mid\rangle\}$);
    $inc\_set = include\_in\_set(node\_set, level)$;
    $exc\_set = exclude\_from\_set(node\_set, level)$;
    $l = decomp\_all(var\_set \cup \{x_{level}\}, inc\_set, level + 1)$;
    $r = decomp\_all(var\_set, exc\_set, level + 1)$;
    return($l \cup r$);
}
```

```
$include\_in\_set(node\_set, level)$
{
    $new\_set = \phi$;
    for each node $\mathbf{u} \in node\_set$ {
        if $(index(\mathbf{u}) == level)$
            $new\_set = new\_set \cup \{child_l(\mathbf{u}), child_r(\mathbf{u})\}$;
        else /* $index(\mathbf{u}) > level$ */
            $new\_set = new\_set \cup \{\mathbf{u}\}$;
    }
    return $new\_set$;
}
```

```
$exclude\_from\_set(node\_set, level)$
{
    $new\_set = \phi$;
    for each node $\mathbf{u} \in node\_set$ {
        if $(index(\mathbf{u}) == level)$
            $new\_set = new\_set \cup \{2^{2^{level}} * child_l(\mathbf{u}) + child_r(\mathbf{u})\}$;
        else /* $index(\mathbf{u}) > level$ */
            $new\_set = new\_set \cup \{(2^{2^{level}} + 1) * \mathbf{u}\}$;
    }
    return $new\_set$;
}
```

Since there are 2^n different bound sets for an n variable function, the computation of the *cut_set* for all of them is very expensive. We can however replace the first if-statement in *decomp_all* by

if $(level == n \mid\mid \mid var\_set \mid == k)$,

then *decomp_all* becomes a routine for computing the cardinality of *cut_set* for every bound set whose size is less than or equal to k. Further speed-ups can be obtained, if we restrict the search to simple disjunctive decomposition.

To assign a unique encoding for each EVBDD node, we need integers with 2^{level} bits. This is clearly very expensive. One way to overcome this difficulty is to relax the uniqueness condition. Then, two different EVBDD nodes representing different functions may be assigned the same encoding. As a result, the size of *cut_set* for a bound set may be underestimated. The algorithm *decomp_all* can then be used as a filter of searching for all bound sets whose *cut_set* have size less than some value k.

A naive way to compute the *cut_set* for every bound set is to move the bound variables to the top of the BDD. Compared to this approach, the EVBDD-based approach has the following advantages. Firstly, it is well known that the size of BDD is very sensitive to the variable ordering (at least in many practical applications [8].) By moving bound variables to the top will change the variable ordering. The EVBDD-based method will not change the variable ordering. Secondly, after the variable inclusion or exclusion operation, the number of EVBDD variables will decrease by 1. On the other hand, the number of variables in the direct variable exchange approach is never reduced. The performance comparison of these two approaches is presented in Section 9.

Without using the encoding scheme as described above, we can also use vectors of BDDs to compute the size of *cut_set* for all bound sets. What we need to do is to replace every EVBDD node by a vector of BDD nodes in algorithms *decomp_all*, *include_in_set* and *exclude_from_set*. For example, EVBDD node **d** in Figure 10 (g) represented by a vector of BDDs is $\langle \mathbf{c}, \mathbf{b} \rangle$. Unfortunately, this is very expensive because the length of vectors is $O(2^n)$.

Instead of using *decomp_all* to compute the *cut_set* size for every bound set, we can modify it to compute the *cut_set* size for any given bound set in place. Again, we can use both representations, BDD or EVBDD. In the following, we present algorithm *cut_set_in_place* which uses EVBDD representation.

```
$cut\_set\_in\_place(bound\_set, node\_set, level)$
/* assuming bound variables are ordered by their indices */
{
   if ($bound\_set == \phi$)
      return(| $node\_set$ |);
   if ($index(first(bound\_set)) == level$) {
      $new\_node\_set = include\_in\_set(node\_set, level)$;
      return($in\_place\_cut\_set(rest(bound\_set), new\_node\_set, level + 1)$);
   }
   else {
   $new\_node\_set = exclude\_from\_set(node\_set, level)$;
      return($in\_place\_cut\_set(rest(bound\_set), new\_node\_set, level + 1)$);
   }
}
```

8 Application to FPGA Mappings

Much work has been done on the synthesis of Look-Up Table (LUT) based FPGAs [4, 6, 9]. However, most of the work focuses on the XC3000 type device. The latest generation of the

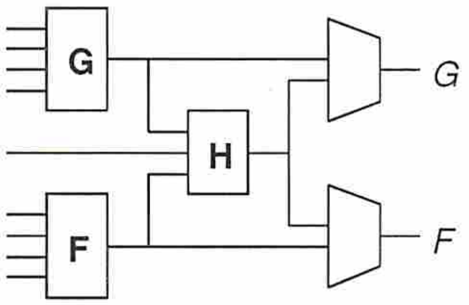


Figure 12: The Xilinx XC4000

Xilinx FPGA devices, i.e., XC4000, contains a number of architectural and technological improvements that allows densities up to 20K equivalent gates and support clock rates up to 60MHz. Among the important architectural improvements that contribute to the XC4000 family's, increased logic density and performance is a more powerful and flexible configurable logic block (CLB). A simplified block diagram of the combinational logic part of this CLB is shown in Figure 12. One key issue in synthesis for XC4000 device is to obtain maximal utilization of the CLBs provided on the device. The function decomposition theory developed in previous sections provides an efficient and effective solution to this problem.

Nine different patterns of XC4000 device are recognized for mapping to difenert types of functions (Figure 13). Among these patterns, the first two patterns are the most interesting and cost effective. Note that the part enclosed by dotted box in the second pattern Figure 14 can be interpreted as an instance of non-disjunctive decomposition. To have such an interpretation, consider the following decompositions:

$$\begin{aligned} f(X_f, X_g, x_h, \ldots) &= f_1(F(X_f), G(X_g), x_h, \ldots) \\ &= f_1(x_f, x_g, x_h, \ldots) \\ &= f_2(H(x_f, x_g, x_h), x_f, \ldots) \end{aligned}$$

In the first decomposition (f to f_1), variables X_f and X_g are bound variables with respect to the functions F and G. In the second line of the above equation, we replace $F(X_f)$ and $G(X_g)$ by variables x_f and x_g. Then, in the second decomposition (f_1 to f_2), variable x_f is both a bound variable and free variable.

We show how to use the techniques introduced in this paper to map Boolean functions to the first two patterns of Figure 13. Given a BDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$, two sets of variables X_f and X_g each containing at most 4 variables, and a variable x_h, the following algorithm *pattern*$(\mathbf{v}, X_f, X_g, x_h)$ returns 1 if $\{X_f, X_g, x_h\}$ can be mapped to the pattern in Figure 13 (a); returns 2 if it can be mapped to the pattern in Figure 13 (b); otherwise it returns 0. The sets of variables X_f and X_g can be computed by the method described in Section 7.

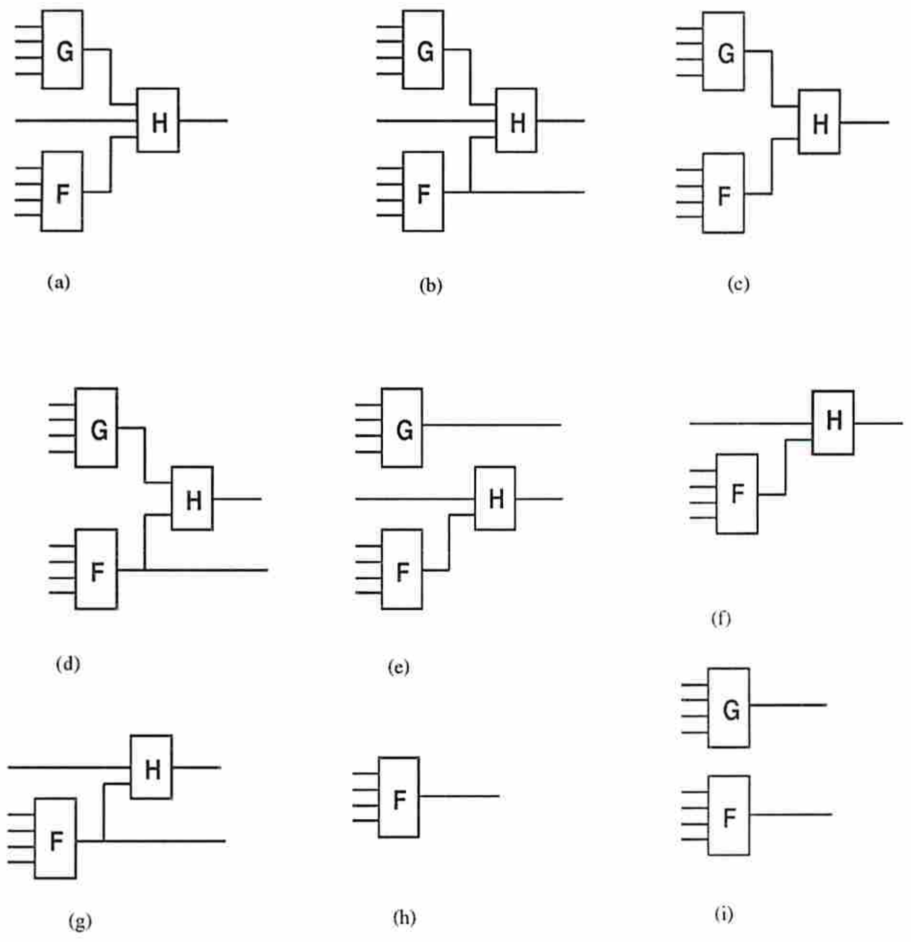


Figure 13: XC4000 patterns

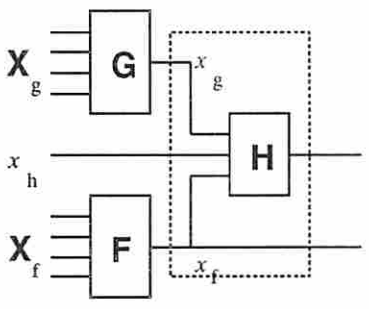


Figure 14: Non-disjunctive decomposition point of view of pattern (b)

```
$match\_pattern(\mathbf{v}, X_f, X_g, x_h)$
{
1     for $(i = 0; i <\mid X_f \mid; i{+}{+})$
2         $\mathbf{v} = rotate(\mathbf{v}, X_{f_i});$
3     $cset = cut\_set(\mathbf{v}, \mid X_f \mid -1);$
4     if $(\mid cset \mid > 2)$ return 0;
5     $\mathbf{v} = decomp\_f(cset, \{x_f\});$
6     for $(i = 0; i <\mid X_g \mid; i{+}{+})$
7         $\mathbf{v} = rotate(\mathbf{v}, X_{g_i});$
8     $cset = cut\_set(\mathbf{v}, \mid X_g \mid -1);$
9     if $(\mid cset \mid > 2)$ return 0;
10    $\mathbf{v} = decomp\_f(cset, \{x_g\});$
11    $\mathbf{v} = rotate(\mathbf{v}, x_h);$
12    $cset = cut\_set(\mathbf{v}, 2);$
13    if $(\mid cset \mid > 4)$ return 0;
14    if $(\mid cset \mid \leq 2)$ return 1;
15    if $(\mid cut\_set\_nd(\mathbf{v}, 2, 2, 0) \mid \leq 2$ && $\mid cut\_set\_nd(\mathbf{v}, 2, 2, 1) \mid \leq 2)$
16        return 2;
17    else return 0;
}
```

The algorithm *match_pattern* is straightforward. The first stage (lines 1-5) is to move the variables X_f to the top and compute the *cut_set* with respect to X_f. If the *cut_set* size is greater than 2, X_f cannot be mapped to a single LUT. The second stage (lines 6-10) is the same as the first stage except the variables in X_g is used. The third stage is simple because x_h is a single variable. Line 12 computes the *cut_set* with respect to variables x_h , x_g and x_f. If the *cut_set* size is greater than 4, then it requires more than two outputs. Neither patterns can be mapped (line 13). On the other hand, if the *cut_set* size is less than or equal to 2, the first pattern is detected in line 14. Finally, if the conditions imposed in line 15 for non-disjunctive decomposition are satisfied, then the second pattern is detected; otherwise no pattern can be mapped.

To cover the case of non-disjunctive decomposition such as the pattern shown in Figure 15, lines 3-5 in *pattern* are modified as follows:

```
3'  $cset[0] = cut\_set\_nd(\mathbf{v}, \mid X_f \mid -1, \mid X_f \mid -1, 0);$
    $cset[1] = cut\_set\_nd(\mathbf{v}, \mid X_f \mid -1, \mid X_f \mid -1, 1);$
4'  if $(\mid cset[0] \mid > 2$ || $\mid cset[1] \mid > 2)$ return 0;
5'  $\mathbf{v} = nd\_decomp\_f(cset, \{x_f\});$
```

Note that line 3' assumes that there is only one variable (the last one in X_f) which is both in the bound set and free set. Modifications to allow more than one shared variable are straightforward.

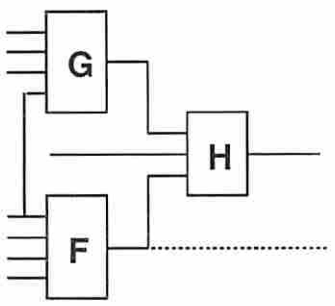


Figure 15: Non-disjunctive mapping for the XC4000 device

9 Experimental Results

The BDD-based decomposition algorithms described in this paper have been implemented in C and incorporated into the SIS package. In order to evaluate the effectiveness of these algorithms, we compared our program with the Roth-Karp decomposition algorithm implemented in SIS as part of the Xilinx FPGA synthesis tool. In particular, we used the following commands on a number of mcnc91 benchmark sets:

1. "xl_k_decomp[1] -n 4 -d -f 100" which for every node in the Boolean network, finds the the first bound set of size ≤ 4 that reduces the node's variable support after decomposition, and then decomposes the node, and modifies the network to reflect the change.
2. "xl_k_decomp -n 4 -e -d -f 100" which is similar to (1) except that it finds the best bound set (the one that has the least cost based on a quick estimation procedure) among all the bound sets of size ≤ 4.

We provided equivalent BDD-based implementation of these two SIS commands.

Results are shown in Tables 1 and 2. We stopped the processes which took more that 5000 cpu seconds on a Sun Sparc-Station 2. We obtain significant speed-ups in either case (by an average factor of 17.2 in the first case and 35.4 in the second case). We believe this is due to our efficient BDD and EVBDD representations. Note that although faster, our program has more memory requirements.

When checking for the decomposability of a given function (examining all bound sets), we implemented two options: a straight-forward variable rotation scheme that moves variables of the bound set to the top of the BDD (ROTATE) and an encoding scheme using EVBDDs (EVBDD). Table 3 shows our comparative results. Here, we only searched for bound sets of size 4 and simple decomposability (*cut_set* size of ≤ 2). The EVBDD based approach is faster by a factor of 1.5.

10 Conclusions

We presented BDD-based algorithms for disjunctive and non-disjunctive decomposition of a Boolean function given a bound set of variables. We also presented the theory for incom-

[1]*xl_k_decomp* does not cover the circuits whose input sizes are greater than or equal to 32.

Circuit	inputs	outputs	karp	EVBDD	Speed-up
5xp1	7	10	1.6	0.7	2.3
9sym	9	1	3.3	0.9	3.7
apex4	9	19	1317.6	45.2	29.2
misex3	14	14	1638.8	160.2	10.2
misex3c	14	14	177.4	24.9	7.1
Z5xp1	7	10	2.7	2.0	1.4
misex2	25	18	1.8	0.6	3.0
sao2	10	4	6.0	0.8	7.5
xor5	5	1	0.3	0.2	1.5
b12	15	9	2.2	0.6	3.7
ex1010	10	10	> 5000	49.4	> 101.2
squar5	5	8	0.6	0.5	1.2
Z9sym	9	1	4.8	1.5	3.2
t481	16	1	36.8	2.6	14.2
alu4	14	8	4458.8	138.2	32.3
table3	14	14	838.7	410.0	2.0
table5	17	15	1213.4	1027.0	1.2
cordic	23	2	397.9	5.9	67.4
vg2	25	8	44.3	1.3	34.1
seq	41	35	NA	59.1	—
apex1	45	45	NA	17.4	—
apex2	39	3	NA	94.3	—
apex3	54	50	NA	58.3	—
e64	65	65	NA	7.7	—
Average					17.2

Table 1: Finding the first decomposable form with bound set size ≤ 4

Circuit	karp	EVBDD	Speed-up
5xp1	31.1	2.9	10.7
9sym	513.6	3.7	138.8
apex4	2544.8	49.8	51.1
misex3	> 5000	381.9	13.1
misex3c	> 5000	131.8	37.9
Z5xp1	31.4	4.3	7.3
misex2	416.8	87.7	4.8
sao2	337.9	23.6	14.3
xor5	2.0	0.3	6.7
b12	74.3	6.4	11.6
ex1010	> 5000	48.5	> 103.1
squar5	4.3	0.9	4.8
Z9sym	508.2	4.2	121.0
t481	> 5000	62.1	> 80.5
alu4	> 5000	124.5	> 40.2
table3	> 5000	318.6	> 15.7
table5	> 5000	1225.8	> 4.1
cordic	> 5000	1331.7	> 3.8
vg2	> 5000	2051.3	> 2.4
seq	NA	> 5000	—
apex1	NA	> 5000	—
apex2	NA	> 5000	—
apex3	NA	> 5000	—
e64	NA	> 5000	—
Average			35.4

Table 2: Finding all decomposable form with bound set size ≤ 4

Circuit	ROTATE	EVBDD	Speed-up
5xpl	1.1	1.1	1.0
9sym	1.9	1.2	1.6
apex4	33.9	29.4	1.2
misex3	228.6	79.0	2.9
misex3c	39.9	22.9	1.7
Z5xpl	2.4	2.4	1.0
misex2	19.7	11.5	1.7
sao2	3.9	2.9	1.3
xor5	0.2	0.3	0.7
b12	2.5	1.2	2.1
ex1010	44.4	31.8	1.4
squar5	0.5	0.6	0.8
Z9sym	2.3	1.8	1.3
t481	17.0	15.3	1.1
alu4	95.8	53.1	1.8
table3	173.8	88.2	2.0
table5	410.4	286.3	1.4
cordic	107.4	67.7	1.6
vg2	146.7	105.4	1.4
seq	> 5000	2588.2	> 1.9
apex1	> 5000	> 5000	1.0
apex2	> 5000	> 5000	1.0
apex3	> 5000	> 5000	1.0
e64	> 5000	> 5000	1.0
Average			1.5

Table 3: Comparing methods based on ROTATE and EVBDD

pletely specified and multi-output function decomposition using BDDs. We described an EVBDD-based procedure for efficient computation of all the bound sets that make the function decomposable. We showed that these ideas can be used to detect the most interesting and cost effective patterns of the Xilinx XC4000 device on a BDD representation of the function. This provides the first step toward developing an FPGA synthesis tool that directly maps BDDs into Look-Up Table based FPGA devices without resorting to conventional (algebraic) optimization and restructuring techniques.

In comparing our results with the SIS implementation of the Roth-Karp technique, we achieves an average of 17.2 speed-ups in runtime in searching for the first decomposable form and an average of 35.4 speed-ups when all the decomposable forms are examined.

Our future work will focus on developing a BDD-based FPGA synthesis tool.

Acknowledgement – This work was supported in part by the NSF's Research Initiation Award under contracts No. MIP-9111206 and No. MIP-9211668.

References

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

[2] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, C-35(8): 677-691, August 1986.

[3] H.A. Curtis, "A new approach to the Design of Switching Circuits," Princeton, N.J., Van Nostrand, 1962.

[4] R.J. Francis, J. Rose and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs,", Proc. 28th DAC, June 1991, pp. 227-233.

[5] M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.

[6] K.Karplus, "Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays," Proc. 28th DAC, June 1991, pp. 240-243.

[7] Y-T. Lai and S. Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," Proc. of 29th Design Automation Conf., pp. 608-613, 1992.

[8] H-T. Liaw and C-S Lin, "On the OBDD-Representation of General Boolean Functions," *IEEE Transactions on Computers*, C-41(6): 661-664, June 1992.

[9] R. Murgai, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," ICCAD, 1991.

[10] J.P. Roth and R.M. Karp, "Minimization Over Boolean Graphs," IBM Journal, April 1962, pp. 227-238.

[11] V.Y. Shen and A.C. McKellar, "An algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Transaction on Computers*, C-19(3): 239-248, March 1970.