The Disjunctive Decomposition of Logic Functions

Valeria Bertacco
Computer Systems Lab.
Stanford University

Maurizio Damiani
Synopsys, Inc.
Mountain View, CA

Abstract

We present an algorithm for extracting a disjunctive decomposition from the BDD representation of F. The output of the algorithm is a multiple-level netlist exposing the hierarchical decomposition structure of the function. The algorithm has thoretical quadratic complexity in the size of the input BDD. Experimentally, we were able to decompose most synthesis benchmarks in less than one second of CPU time, and to report on the decomposability of several complex ISCAS combinational benchmarks. We found the final netlist to be often close to the output of more complex dedicated optimization tools.

1 Introduction

The decomposition of a logic function $F(x_1, \cdots, x_n)$ is the identification of a set of functions $A_i(x_i)$ with no variables in common , and a function L such that [1, 2, 3]:

$$F = L(A_0, \cdots, A_i, \cdots)$$

The decomposition of a function is desirable for several reasons. First, it is obviously one method for passing from a flat representation of F to a multiple-level one. It leads to a clustering of the inputs in simpler - and smaller - logic blocks, thereby reducing area and interconnect. This latter property is especially desirable in many implementation technologies, such as FPGAs or deep-submicron, where wiring resources are scarce or costly in terms of delay. Moreover, it exposes a parallelism in the computation of F, by evidencing the independence in the computation of the functions A_i. This parallelism can be exploited both in hardware, leading to faster realizations, as well as during simulation.

Because of its importance, decomposition is a classic subject of switching theory,still being researched [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

Known exact decomposition algorithms are based on the *decomposition chart* method of Ashenhust and Curtis [1]. The method is based on constructing *trial partitions* of the set of variables and then of verifying on the decomposition chart that F can indeed be decomposed using that clustering of its inputs. Unfortunately, because of the potentially exponential number of trial partitions, exact decomposition is usually confined to functions of few variables. It is worth noting, however, that simplified techniques, such as algebraic factorization [4], have been extremely successful in transforming large two-level covers in multiple-level representations, and have been extended in various ways to include other forms of decomposition.

This paper presents two contributions. First, we present a novel, efficient exact decomposition procedure. The procedure is BDD-based , and it determines the decomposition of function directly from that of its cofactors. Its complexity is bounded by $|F|^2 \times n_F$, where $|F|$ and n_F denote the number of BDD nodes and input variables of F, respectively. In practice, we found the procedure to be very fast, as we were able to determine the decomposition of most benchmark circuits in less than five seconds on a PC. We have also derived -we believe for the first time- the decompositions of several combinational ISCAS benchmarks.

The decomposition engine is the core of a logic optimization tool, `LODE`. `LODE` outputs a netlist for a function $F(x_1, \cdots, x_n)$ based on its decomposition. If F has no disjunctive decomposition, the output is constructed using the decomposition of the cofactors F_0, F_1 of F with respect to its first variable, x_1. Interestingly, since the decomposition of F is essentially unique [1], it could be shown that the netlist output by `LODE` is also canonical. Moreover, it is again generated in time proportional to $|F|^2$. Although `LODE` is currently missing many optimization opportunities, we found it in practice to produce netlists of comparable quality - sometimes better - to SIS, in a small fraction of the CPU time, for several synthesis benchmarks.

2 Terminology.

Hereafter, we assume the reader be familiar with BDDs [11, 12]. Let $\mathcal{B}$ denote the Boolean set $\{0, 1\}$. A logic function is a mapping $F : \mathcal{B}^n \rightarrow \mathcal{B}^m$. Hereafter, lower-case and upper-case letters will denote logic variables and functions, respectively. We will be mostly concerned with **scalar** functions $F : \mathcal{B}^n \rightarrow \mathcal{B}$. We use boldface to indicate vector-valued functions. The i^{th} component of a vector function $\mathbf{F}$ is indicated by $\mathbf{F}_i$.

We say that a function F **depends** on a variable x_i if $\partial F / \partial x_i$ [13] is not the constant function 0. We call **support** of F the set S_F of variables F depends on. The **size** of S_F is the number of its elements, and it is indicated by $|S_F|$. Two functions F, G are termed **disjoint-support** if they share no support variables, *i.e.* $S_F \cap S_G = \phi$.

2.1 Disjunctive Decompositions.

The decomposition of a function F consists of finding other, simpler functions L and A_i such that

$$F(x_1, \cdots, x_n) = L(A_1(x_1, \cdots), A_2(x_1, \cdots), \cdots) \quad (1)$$

Definition 1. *A function $L(a_1, \cdots, a_k)$, $n > k \geq 2$ is said to* **reduce** *a function $F(x_1, \cdots, x_n)$ if there are k non-constant disjoint-support functions $A_1, \cdots, A_k$ that satisfy*

Eq. (1). F is said to be **prime** *if it cannot be decomposed by* **any** L. $\square$

The functions A_i will be termed **formal inputs** of the decomposition of F. The list of formal inputs to F will be indicated as F/L, and termed **decomposition list** of F. We call **disjunctive decomposition** of F any pair $(L, F/L)$ that satisfies Eq. (1).

If F is decomposable, it is possible to characterize its decomposition as follows:

- there is a **unique** prime function L decomposing it, up to permutations/complementations of its formal inputs.
- if L has support size $|S_L| > 2$, then also the functions in F/L are uniquely determined, up to complementation/permutation.
- if L has support size $|S_L| = 2$ then F is decomposable in **exactly one** of the following ways :
  - as the OR of disjoint-support functions; the functions of F/L are uniquely identified for the decomposition of finest granularity [14];
  - as the complement of a OR-decomposed function ; again, the functions of F/L are uniquely identified for the decomposition of finest granularity [14];
  - as the XOR/XNOR of disjoint-support functions; the functions in F/L are identified modulo complementation.

If a nontrivial function A belongs to F/L, we take the **cofactor** F_A (or $F(A = 1)$) to be the function $L(A = 1, \cdots)$. Notice that, because of the uniqueness of L and of the other elements in F/L, this function is unique and shares no variables with A.

2.2 Representing decompositions.

We represent the decomposition of the function F rooted at a BDD node N by annotating N with a decomposition type field `type`; and a sorted list `list` of BDD nodes, representing the decomposition list of F. The field `type` can take on the values `OR, NOR, XOR, XNOR, PRIME, UNDEF`, depending on the type of decomposition. Initially, `type` is `UNDEF`. After application of the decomposition procedure, `type` can only take one of the other values.

Each node of `list` points to a BDD node. This node is the root of one of the formal inputs A_i.

The ambiguity due to permutations and complementations is resolved by sorting the functions according to their supports, and by not allowing complement edges in `list`.

Example 1. Consider the function $F = MAJORITY(a \oplus b, c + d, ef)$. Its BDD is shown in Fig.(1), along with the annotation of the root node and of significant other decompositions. $\square$

In order to resolve ambiguities in the representation, we adopt the following rules:

1. Lists are sorted by order of top variable, that is , a function A_i appears before A_j if the top variable of A_i is ranked higher than that of A_j.
2. pointers to BDDs in decomposition lists of type XOR, XNOR, or PRIME cannot be complemented.

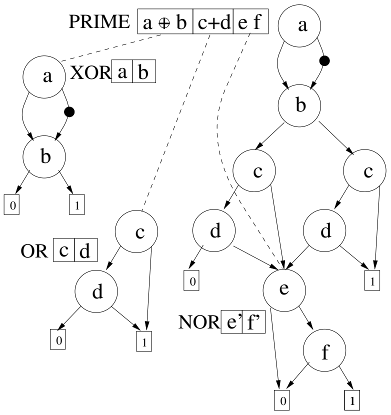

Figure 1. BDD and decomposition annotation for the function of Example (1).

As each function in the decomposition list F/L may be itself decomposable, the lists associated with the decomposition of F form a tree, hereafter called the **decomposition tree** of F.

3 Shannon- and disjunctive- decompositions.

A BDD node, with root variable z, represents a function $F(z, x_1, \cdots, x_n)$ as

$$z' F_0(x_1, \cdots, x_n) + z F_1(x_1, \cdots, x_n) \quad (2)$$

where $F_0 = F(z = 0)$; $F_1 = F(z = 1)$. Our goal is to infer a decomposition

$$F = L(A, B, \cdots) \quad (3)$$

from the decompositions of F_0, F_1. This inference mechanism is the basis of the decomposition algorithm of Section (4).

The link between the decomposition of F and that of its cofactors is conceptually simple. For example, if a function A belongs to the decomposition tree of both F_0 and F_1, then it must belong to the decomposition tree also of F. Unfortunately, several cases and sub-cases need be considered in practice. These are enumerated below.

We now describe in detail how the decomposition structure of F is reflected in that of its cofactors. Moreover, we show how the information on the decomposition of the two cofactors is sufficient to infer the decomposition of F.

The top variable z of F can appear in the decomposition of a function F in only one of the four following ways :

1 : F is the OR/NOR of z/z' with a function A, $z \notin S_A$;

2 : $F = z \oplus A$, $z \notin S_A$.

3 : Cases 1 and 2 do not hold, and z belongs to the support of one formal input of F (say, A), with $|S_A| \geq 2$;

4 : Cases 1 and 2 do not hold, and z is one of the formal inputs of F.

We now analyze the consequences of these cases on the decomposition of F_0, F_1.

Case 1.

In this case, one of $F(z = 0)$, $F(z = 1)$ is a constant. Viceversa, suppose that one cofactor is a constant (say,

$F_1 = 1$, the other cases being symmetric). In this case, trivially $F = F_0 + z$, hence `F.type = OR` and the decomposition list of F is that of F_0 plus z.

Case 2.

Clearly, in this case, $F_0 = F_1'$. Verifying $F_0 = F_1'$ is sufficient to infer the decomposition of F as $F = z \oplus F_0$.

Case 3.

Suppose z belongs to the support of a function (say, A) with nontrivial support $|S_A| \geq 2$. We need to distinguish two sub-cases :

3.a Neither cofactor of A ($A_{z=0}$, $A_{z=1}$) is a constant;

3.b Exactly one cofactor of A is a constant.

Notice that the case of both cofactors being constants rules out the possibility that $|S_A| \geq 2$.

Case 3.a

In this case, F_0, F_1 are :

$$F_0 = L(A(z=0), B, \cdots); \quad F_1 = L(A(z=1), B, \cdots) \quad (4)$$

In other words, F_0 and F_1 are decomposable by the same function L. Moreover, the decomposition *lists* of F_0 and F_1 will coincide, except for *at most* one element ($A(z=0)$ vs. $A(z=1)$).

Viceversa, suppose that F_0 and F_1 have decompositions containing functions A_0 and A_1, respectively, and such that

$$\begin{aligned} F_0(A_0 = 0) &= F_1(A_1 = 0) \quad \text{and} \\ F_0(A_0 = 1) &= F_1(A_1 = 1) \end{aligned} \quad (5)$$

it then follows that, by forming $A = z'A_0 + zA_1$, one can write

$$F = F_0(A, \cdots) = F_1(A, \cdots) \quad (6)$$

Example 2. Consider function $F = MAJORITY(\bar{a}b + ad, c + e, fg)$, and its cofactors $F_0 = F(a=0), F_1 = F(a=1)$. Both cofactors are annotated as `PRIME` functions, with decomposition lists $b, c+e, fg$ and $c+e, d, fg$, respectively. The two lists differ in exactly one element (b instead of d), and moreover, Eq. (5) holds for $A_0 = b, A_1 = d$. Hence the inference of the decomposition $\bar{a}b + ad, c + e, fg$ for F. □

Example 3. Consider the function $F = MAJORITY(a \oplus b, c + e, fg)$. In this case, F_0 and F_1 have identical decomposition lists $b, c+e, fg$. We infer the decomposition by observing that the only possibility is that in Eq. (5), $A_1 = A_0'$. We then test Eq. (5) using first $A_0 = b$, $A_1 = b'$. Since the test succeeds, we list $A = z'A_0 + zA_1 = z \oplus A_0$ in the decomposition list of F, along with $c + e, fg$. □

Case 3.b

We only consider the case where the cofactor $A(z=1) = 1$ (*i.e.* $A = z + A_0$), the other cases being symmetric.

Suppose that F has decomposition $F = L(A, B, C, \cdots)$. Then

$$F_0 = L(A_0, B, C, \cdots); \quad F_1 = L(1, B, C, \cdots). \quad (7)$$

In particular, the element A_0 is missing from the decomposition of F_1, and $F_0(A_0 = 1) = F_1$.

Viceversa, suppose that F_0 has a decomposition containing a function A_0, that $S_{A_0} \cap S_{F_1} = \phi$, and that $F_0(A_0 = 1) = F_1$. In this case, one can write

$$F = F_0(A_0 + z, \cdots). \quad (8)$$

Example 4. Consider the function $F = MAJORITY(a + b, c, d + e)$. Consider the cofactors $F_0 = F(a=0) = MAJORITY(b, c, d+e)$ and $F_1 = F(a=1) = c+d+e$. In this case, b is the only function of F_0 (just a variable, in this case) not in the support of F_1. The test $F_0(b=1) = F_1$ is satisfied; hence $A = a + b$ and the decomposition list of F is that of F_0 with A replacing b. □

Case 4.

F must be of type `PRIME`, and the function z belongs to its list of formal inputs. The decomposition tree - and hence the decomposition list - of F can be constructed from those of F_0, F_1 as follows:

1. If a function A belongs to the decomposition tree of F_0 and $S_A \cap S_{F_1} = \phi$, then A belongs to the decomposition tree of F, along with its descendants.
2. If a `PRIME` function A belongs to the decomposition tree of F_0, F_1, then it belongs to that of F, along with all its descendants.
3. If a OR/NOR function A belongs to the decomposition tree of F_0, and a OR/NOR function B belongs to the decomposition tree of G, then the OR of the functions common to the decomposition lists of A and B belongs to the tree of F;
4. If a XOR function A is in the tree of F_0, and a XOR function is in the tree of F_1, then the XOR of the common terms belongs to the tree of F.
5. The decomposition list of F contains all functions obtained by rules 1-4 that do not have ancestors.

Rule (3) above accounts for the fact that if a function $A = a+b+c$ decomposes F_0 and , say $B = a+b+d$ decomposes F_1, then $a + b$ certainly decomposes F. Similarly for rule (4).

Example 5. Consider the function $F = a'(b + c)(d + e + f) + a(b + c + d)(e + f + g)$. By rule (1), the function g belongs to the tree of F. By rule (3), the functions $b + c$ and $e + f$ belong to the decomposition of F. Eventually, the function d belongs to the decomposition of F by Rule (2). The decomposition list of F is then $a, b + c, d, e + f, g$. □

Rules 1-5 are sufficient to construct the decomposition tree/list of F:

Theorem 1. *If a function A belongs to the decomposition tree of F, then one of conditions $1 - 4$ above must hold true for A.* □

4 A BDD-based decomposition procedure.

In this section we present the algorithm for annotating the decomposition of a function F on its BDD representation. The annotation is accomplished in a single sweep of

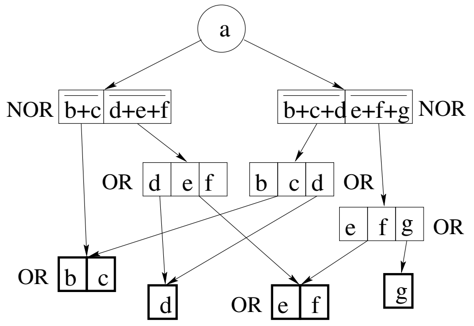

Figure 2. Construction of the decomposition tree for the Example (5).

the BDD nodes, sorted topologically in a linear array. During the sweep, each node is inspected, and the decomposition of the function rooted at that node is determined, from that of the children. The node is then labeled with a pointer to its decomposition list and annotated with its decomposition type.

The decomposition method is enumerative. We try to verify Cases 1 to 3b of Section (3), in order. If we fail, a node is annotated `PRIME` and its decomposition list constructed as in Case 4.

```
DEC decompose(BDD* node) {
   DEC left = node->left->decomposed;
   DEC right = node->right->decomposed;
   VAR var   = node->argument;

   // check if any cofactors are constant:
   res = decompose_ornor(var, left, right);
   if (res) return(res);

   // check if the cofactors
   // are each other's complements
   res = decompose_xor(var, left, right);
   if (res) return(res);

   // check if var belongs to a subfunction
   res = decompose_case3(var, left, right);
   if (res) return(res);

   // construct the new decomposition list
   res = decompose_prime(var, left, right);
   assert(res);
   return(res);
}
```

4.1 Synthesis.

`LODE` generates a netlist from a depth-first traversal of the decomposition of F as follows:

```
synthesis(F) {
  if (F.synthesized) return(F.synthesized);
  switch (F.type) {
    case PRIME :
      F_0 = disjoint_cof(F, F.dec.first');
      F_1 = disjoint_cof(F, F.dec.first);
      N0 = synthesis(F0);
      N1 = synthesis(F1);
      NC = synthesis(F.dec);
      return(MUX(NC, N0, N1));
    case OR, XOR, NOR :
      for F_0 in F.dec
          F[i] = synthesis(F_0); i++;
      return(OR/NOR/XOR(F[])); }
}
```

5 Experimental results.

The procedures of this paper have been implemented in C and tested on several combinational logic benchmarks. The CPU time was taken on a PC equipped with a 150 MHz Pentium and 96Mbyte of main memory.

Table (1) below reports results on the decomposability of some large combinational and sequential benchmarks. Column `DEC` reports the number of decomposable outputs, while `CPU time` is reported in seconds. The decompositions themselves are often of interest. For instance, in some sequential circuits, such as s1494, the next-state outputs are often ANDed with the same signal (`v13_D_0C`). Most of the outputs of C880 contain complex functions of the same formal input AND(72GAT[12], 68GAT[11], 13GAT[2], 73GAT[13]). The output is also often the OR of these complex functions with 255GAT[54], 259GAT[55], etc... Synthesis benchmarks are mostly decomposable. The decomposition of some circuits, such as DES, *pair*, etc ... also evidences regular designs, that we cannot describe here for reasons of space. For some ISCAS circuits, we have also been able to decompose some of the outputs corresponding to complex functions.

Table (2) below compares the literal counts obtained by `LODE` against those obtained by `SIS` running `rugged_script`. Finally, table (3) reports the CPU time employed by `LODE` for the decomposition.

For the largest benchmarks, the limited set of BDD transformations of `LODE` do not compensate for the exceptional growth of the BDD representation with respect to the original representation. We are currently working towards a partitioning strategy and enriching the transformation set.

6 Conclusions.

We have presented an algorithm for the disjunctive decomposition of logic functions, starting from a BDD representation. The algorithm has worst-case complexity quadratic in the BDD size. We found it very fast in practice, as we were able to obtain the decomposition of all benchmark functions in a few minutes in the worst case. We have been able to present results on the decomposability of complex benchmarks for the first time. Moreover, we have been able to generate a very compact, canonical multiple-level circuit directly from a BDD representation.

We are currently investigating some implications of the present work: First, the possibility of generating the decompositio form of this paper directly from a netlist. Second, we are exploring the applicability of this representation to rapid prototyping, technology mapping and reachability analysis.

References

[1] R. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, pages 74–116, April 1957.

[2] J. P. Roth and R. M. Karp. Minimization over boolean graphs. *IBM Journal*, pages 661–664, April 1962.

Circuit	Inputs	Outputs	DEC	CPU time
C1355	41	32	0	91.25s
C1908	33	25	7	7.58s
C3540	50	22	0	21.10s
C432	36	7	1	1.23s
C499	41	32	0	83.47s
C880	60	26	25	2.71s
CM42	4	10	10	0.15s
CM85	11	3	3	0.27s
alu4	14	8	4	0.37s
apex6	135	99	99	2.62s
apex7	49	37	36	1.03s
comp	32	3	3	0.71s
count	35	16	16	0.73s
frg2	143	139	126	2.86s
k2	45	45	35	1.04s
pair	173	137	125	4.02s
rot	135	107	77	22.62s
vda	17	39	20	0.40s
x3	135	99	99	2.69s
x4	94	71	65	1.90s
apex1	45	45	35	1.01s
apex2	38	3	3	1.14s
apex4	9	19	4	0.33s
apex5	114	88	82	2.34s
e64	65	65	64	1.31s
misex2	25	18	17	0.57s
seq	41	35	34	1.10s
s1196	32	32	14	0.71s
s1238	32	32	14	0.75s
s1423	91	79	71	12.48s
s1488	14	25	19	0.36s
s1494	14	25	19	0.34s
s420	35	18	18	0.75s
s444	24	27	21	0.54s
s641	54	42	35	1.12s
s953	45	52	22	20.97s

Table 1. Decomposability of some large benchmark circuits.

Circuit	IN	OUT	Original literals	LODE literals	SIS literals
9symml	9	1	277	76	223
CM150	21	1	77	47	51
PARITY	16	1	60	60	60
alu2	10	6	453	354	357
cmb	16	4	62	36	51
f51m	8	8	169	98	91
lal	26	19	221	134	105
mux	21	1	92	47	51
term1	34	10	624	165	197
ttt2	24	21	341	258	216
s1494	14	25	1393	793	661
s298	17	20	244	146	114
s526	24	27	445	257	191
s832	23	24	769	431	352

Table 2. Experimental result and comparisons against SIS.

Circuit	IN	OUT	LODE CPU time	SIS CPU time
9symml	9	1	0.26s	25.83s
CM150	21	1	0.51s	0.51
PARITY	16	1	0.38s	0.27
alu2	10	6	0.28s	106.56s
cmb	16	4	0.36s	0.31
f51m	8	8	0.26s	2.23
lal	26	19	0.55s	2.21
mux	21	1	0.48s	0.55
term1	34	10	0.75s	16.22s
ttt2	24	21	0.55s	5.10
s1494	14	25	0.34s	65.77s
s298	17	20	0.40s	1.83
s526	24	27	0.52s	5.62
s832	23	24	0.54s	27.66s

Table 3. CPU time for decomposition and comparisons against SIS.

[3] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[4] R.K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *ISCAS, Proceedings of the International Symposyium on Circuits and Systems*, pages 49–54, 1982.

[5] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis fro programmable gate arrays. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 620–625, June 1990.

[6] Kevin Karplus. Representing boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for ComputerEngineering & Information Sciences, 1988.

[7] Kevin Karplus. Using if-then-else dsgs for multi-level logic minimization. Technical Report UCSC-CRL-88-29, Baskin Center for ComputerEngineering & Information Sciences, 1988.

[8] Kevin Karplus. Using if-then-else dsgs to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Baskin Center for ComputerEngineering & Information Sciences, 1990.

[9] K.-R. Pan Y.-T.Lai and M. Pedram. Obdd-based functional decomposition: algorithms and implementation. *IEEE Trans. on CAD/ICAS*, 15(8):977–990, August 1996.

[10] T. Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Trans. on Computers*, C-30(9):635–643, September 1981.

[11] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[12] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[13] Edward J. McCluskey. *Logic Design Principles With Emphasis on Testable Semicustom Circuits*. Prentice-Hall, 1986.

[14] V. Bertacco and M. Damiani. Boolean function representation based on disjoint-support decompositions. submitted to *ICCD 96*, to appear, October 1996.

[15] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD*, pages 126–129, November 1990.