The disjunctive decomposition of logic functions

Maurizio Damiani
Synopsys, Inc.
700 E. Middlefield Rd.
Mountain View, CA 94043

Valeria Bertacco
Computer Systems Lab.
Stanford University
Stanford, CA 94305

Abstract

In this paper we present an algorithm for converting a BDD representation of a logic function into a multiple-level netlist of disjoint-support subfunctions. On the theoretical side, we show that the algorithm takes at most quadratic time in the size of the BDD and that the resulting netlist retains the canonicity properties of the original BDD. Experimentally, we found the algorithm to be extremely fast, taking at most a few minutes for the most complex benchmark circuits. The resulting netlist is also often better than what achieved by conventional synthesis tools.

1 Introduction.

A disjunctive decomposition of a function $F(x_1, \cdots, x_n)$ is a partitioning of the inputs $x_1, \cdots, x_n$ in disjoint subsets, so that F can be given a multiple-level representation as :

$$F(x_1, \cdots, x_n) = G(A(x_1, \cdots, x_A), B(x_{A+1}, \cdots, x_B), \cdots) \quad (1)$$

Discovering a disjunctive decomposition of F such as in Eq. (1) is beneficial in several respects:

First, a disjunctive decomposition evidences some parallelism in the (hardware or software) computation of F: one can evaluate the component functions $A, B, \cdots$, in parallel, as they share neither variables nor intermediate results.

Second, a disjunctive decomposition suggests implicitly a clustering of inputs and functions so as to reduce the amount of interconnect: Cells can be designed so as to compute separately $G, A, B, \cdots$. The designer can concentrate on the optimization of the cells implementing the individual functions, and the routing problems are simplified as only one wire needs be brought out of each cell. The reduction of interconnects is of increasing importance with the advent of deep submicron technology, where the interconnect delay tends to dominate the overall circuit performance.

Eventually, we observe that a disjunctive decomposition can be beneficial also for other logic functions representations, namely BDDs [1, 2], since it offers implicitly a good partial ordering of the primary inputs for BDD construction.

Previous work and contributions of this paper.

Decomposition algorithms are a classical research subject of switching theory. Exhaustive search procedures have been presented in [3, 4, 5], and were efficient for circuits with six or fewer inputs [4]. At the time, decomposition was regarded as the main path to multiple-level logic synthesis.

Algebraic factoring [6] is a form of disjunctive decomposition: one attempts to decompose a 2-level cover of F into a product $G * H$, where G and H have no variables in common. Factoring is a powerful step in passing from a Boolean cover to a multiple-level representation in multiple-level logic synthesis [7].

Recently, decomposition has also been considered in the context of technology mapping ([8]) and canonical Boolean function representation ([9, 10]).

In [10] it was in particular shown that a simple form of decomposition, based on NOR functions, is indeed canonical: A function can be decomposed into the NOR of disjoint-support components in a unique way. This result was used to develop a normal form (MLDDs) for Boolean functions based on Shannon- and NOR- decompositions. Algorithms for translating BDDs into MLDDs and for the direct manipulation of MLDDs were also presented. These algorithms are very efficient in terms of CPU time, as the decomposition of a function is inferred locally from that of the two cofactors. Moreover, the decomposition of cofactors of complex functions often evidenced common subfunctions, whose sharing resulted in memory savings with respect to BDDs. The resulting netlist, however, is not particularly appealing, because of the restriction to NOR decompositions only.

In this paper, we provide the following contributions. First, we present new results on canonicity. We introduce **prime** functions, functions that cannot be further decomposed. Some such functions are, for example, 2-input logic functions, the 2-input MUX function, and all non-trivial symmetric functions.

Clearly, when we attempt the decomposition of a function F as $F = L(A, B, \cdots)$ we only need consider prime functions L. We prove that logic functions can be classified as follows:

1. F is prime;
2. F is decomposable using a binary associative operator X (X = OR,AND,XOR) of disjoint-support functions:
$$F = X(A_0, A_1, \cdots, A_n) \quad (2)$$
In this case, F cannot be decomposed using any other function L; moreover, the functions A_i are identified uniquely (modulo permuttion, complementation) provided they cannot be further decomposed by X;
3. F is decomposable using a prime function L of three or more inputs:
$$F = L(A_0, \cdots, A_n); \;\; n \geq 3 \quad (3)$$

In this case, also L and the functions A_i are identified uniquely (modulo permulations/complementations), that is, no other prime function L can decompose F.

The above results indicate essentially the uniqueness of a disjunctive representation of F.

We then present an algorithm for inferring the disjunctive decomposition of F from the BDD representation. The algorithm is fast, as it is based on the local inference of the decomposition of a function F from that of its cofactors. It is worth noting that the success in identifying a decomposition does not depend on the topology of the BDD representation (as affected, for instance, by the variable ordering). Hence, the method differs in scope from topological techniques such as those presented in [11].

2 Disjoint-support decompositions.

This section provides the relevant definitions and theoretical results.

Let $\mathcal{B}$ denote the Boolean set $\{0,1\}$. A Boolean function is a mapping $F : \mathcal{B}^n \rightarrow \mathcal{B}$.

We say that a function F **depends** on a variable x_i if $\partial F/\partial x_i$ 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 $|S_F|$.

We say that two functions F and G are disjoint-support if they share no variables, *i.e.* $S_F \cap S_G = \phi$.

We say that a function F is **NP-equivalent** to a function G if it can be obtained from G by a permutation / complementation of some of the inputs of G.

Hereafter, lower-case and upper-case letters will denote Boolean variables and functions, respectively.

Definition 1. *A function $F(x_1, \cdots, x_n)$ is said to be* **reducible** *by L if it can be expressed as the composition of other* **non-constant** *disjoint-support functions $A, B, \ldots$:*

$$F = L(A, B, \cdots) \tag{4}$$

F is said to be **prime** *if it cannot be decomposed by* **any** *L.*

If Eq. (4) holds, we say that a function L **reduces** *F. We denote by F/L any set of functions $\{A, B, \cdots\}$ satisfying Eq. (4).* □

It is worth noting that the set F/L is, in general, not unique. Consider, for example, the function $F = x_1 + x_2 + x_3$ and a function $L = a + b$. Then one can write (for instance) $F = L(x_1, x_2 + x_3)$ or $F = L(x_1 + x_2, x_3)$. The two options result in $F/L = \{x_1, x_2 + x_3\}$ and $F/L = \{x_1 + x_2, x_3\}$, respectively.

We also observe that if L reduces a function F, then any other function NP-equivalent to L also reduces F. Moreover, L reduces F if and only if L' reduces F'.

We now present the main theoretical contributions of this paper. For reasons of space, the proofs have been omitted from the paper.

Theorem 1. *Consider an arbitrary function $F(x_1, \cdots, x_n)$, and a* **prime** *function $L(a, b, \cdots)$. Suppose L reduces F:*

$$F = L(A, B, C, \cdots); \tag{5}$$

Then F is reduced only by functions M that are NP-equivalent to L. □

The above Theorem states that the prime function reducing L is unique, modulo syntactic transformations. The following two results indicate that also the functions in F/L are essentially unique:

Theorem 2. *If a function F is reducible by L, with $|S_L| \geq 3$, then the functions F/L are unique, up to permutations and complementations.* □

Theorem 3. *If a function F is reducible by L, with $|S_L| = 2$, then it is reducible by an associative operator X. The operator X decomposing F is unique; moreover, the functions $A, B, \cdots$ in the decomposition*

$$F = X(A, B, C, \cdots)$$

are also unique, provided they are not further decomposable by X. □

Theorem (3) in particular indicates that if a function F can be decomposed as the (say) XOR of two disjoint-support components, then no other operator (AND, OR) decomposes it. Moreover, the component functions are identified uniquely, provided they're not further decomposable.

Since AND-decomposable functions are complements of OR-decomposable ones, Theorems (1 - 3) allow us to ultimately classify logic functions as follows:

1. prime;
2. decomposable by a prime function $L, |S_L| \geq 3$;
3. XOR-decomposable;
4. OR-decomposable;
5. complements of OR-decomposable functions.

3 BDD-based logic decomposition.

We now present algorithms for mapping a BDD representation of a function F into a multiple-level netlist of disjoint-support components. We will first describe the data structures employed for the function representation, and then present the actual relevant procedures.

A disjunctive decomposition could be represented just by introducing hierarchy in an ordinary BDD-based representation. This is obtained by allowing **splitting function** instead of a splitting variable.

Example 1. Consider the function $F = MAJORITY(a \oplus b, cd + e, ITE(fg, h, i))$. F has the following disjoint-support representation:

$$\begin{aligned} F &= MAJORITY(G, H, I) && (6)\\ G &= a \oplus b && (7)\\ H &= L + e && (8)\\ I &= ITE(M, h, i) && (9)\\ L &= cd && (10)\\ M &= fg && (11) \end{aligned}$$

Fig. (1) shows the representation of F. □

This representation, however, does not fit entirely our purposes, because complex functions generally lack significant decompositions, and consequently do not take advantage of hierarchy. It is often the case, however, where a function, albeit lacking a decomposition, has decomposable cofactors. It is then better to represent the function by Shannon decomposition, using the decomposed forms of the cofactors. A function is then here represented by two

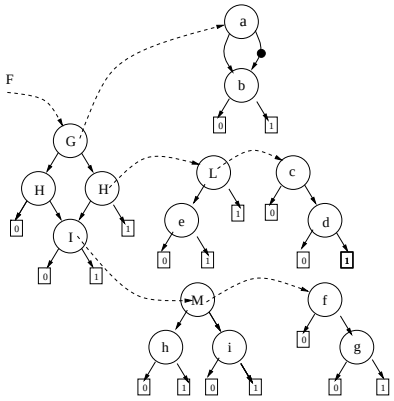

Figure 1. ITE-dag representation for Example (1)

distinct data structures, namely, one representing its decomposition structure (*i.e.***decomposition tree**), and the second its proper logic functionality.

4 Constructing the decomposition.

The algorithms of this paper aim at constructing a representation as in Fig. (2) from the BDD of F.

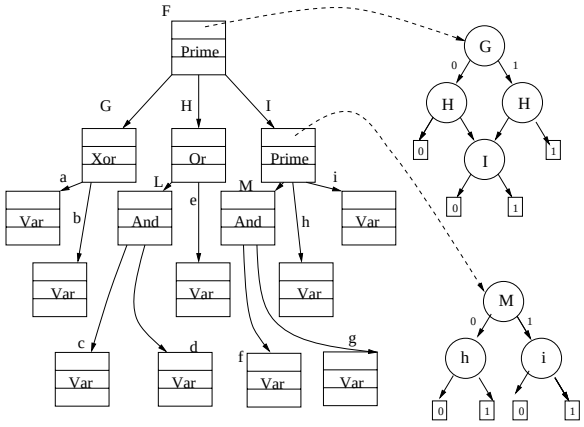

Figure 2. Decomposition tree for F.

This is accomplished by visiting each BDD node N and constructing the decomposition of the function rooted at N from that of its cofactors.

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

$$z'F_0(x_1, \cdots, x_n) + zF_1(x_1, \cdots, x_n) \quad (12)$$

where $F_0 = F(z = 0)$; $F_1 = F(z = 1)$. There exists an intuitive link between the decomposition of F and that of its cofactors F_0, F_1. In practice, we need to distinguish three cases, depending on the role of z in the decomposition of F. Below we describe how to infer the decomposition of F in each of the three cases.

Case 1.

Suppose z belongs to the support of a function (say, A) with nontrivial support $|S_A| \geq 2$. Suppose also that neither cofactor $A(z = 0), A(z = 1)$ is a constant. Then F_0, F_1 are:

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

In other words, F_0 and F_1 are decomposable by the same **prime** function L. Moreover, the decomposition lists of F_0 and F_1 differ **exactly** in one element (A(z =0) vs. A(z=1)).

Viceversa, suppose F_0, F_1 are expressed as

$$F_0 = L_0(A_0, B, \cdots); \quad F_1 = L_1(A_1, B, \cdots)$$

$$|F_0/L_0 \cap F_1/L_1| = |F_0/L_0| - 1 = |F_1/L_1| - 1 \quad (13)$$

(that is, F_0/L_0 and F_1/L_1 differ in exactly one element, A_1 in place of A_0) and suppose

$$L_0(A_0 = 0) = L_1(A_1 = 0);$$
$$L_0(A_0 = 1) = L_1(A_1 = 1); \quad (14)$$

then it is easy to verify that F can be written as in Eq.(5), where $L = L_0$ and $A = A_0z' + A_1z$.

The compliance of F_0, F_1 to Eqs. (13) is easily tested from their decomposition lists. The compliance to Eqs. (14) is tested by computing the four cofactors and verifying their identity.

Example 2. Consider again the function $F = MAJORITY(a \oplus b, cd + e, MUX(fg, h, i))$, and its cofactors $F_0 = F(g = 0), F_1 = F(g = 1)$. We consider available the representations of the cofactors :

$$F_0 = MAJORITY(G, H, h) \quad (15)$$
$$G = a \oplus b \quad (16)$$
$$H = L + e \quad (17)$$
$$L = cd \quad (18)$$
$$F_1 = MAJORITY(G, H, N) \quad (19)$$
$$N = MUX(f, h, i) \quad (20)$$

Their decomposition lists are (G, H, h) and (G, H, N), respectively. They differ in exactly one element, namely, N instead of h. We then check if Eqs. (14) hold. This check can be carried out by computing $F_0(h = 0), F_0(h = 1), F_1(N = 0), F_1(N = 1)$, and verifying the identities $F_0(h = 0) = F_1(N = 0)$, $F_0(h = 1) = F_1(N = 1)$. Indeed, both these identities hold in our case. We then form a representation of the function $I = g'h + gMUX(f, h, i)$ and construct a representation of F as $MAJORITY(G, H, I)$. □

Case 2.

Suppose z belongs to the support of a nontrivial function (say, A, with $|S_A| >= 2$), but such that exactly one of A_0, A_1 is constant (say, $A_1 = 1$). The cofactors are:

$$F_0 = L(A(z = 0), B, \cdots); \quad F_1 = L(1, B, \cdots) \quad (21)$$

In the expression of F_1 one of the arguments of L is a constant. Hence, F_1 is decomposed no longer by L but by some other prime function. Notice, however, that the functions $B, \cdots$ will be in the decomposition tree of F_1. Moreover, the two cofactors still have the property $F_0(A = 1) = F_1$.

Viceversa, suppose F_0, F_1 are decomposable as

$$F_0 = L_0(A, B, \cdots); \quad F_1 = L_1. \quad (22)$$

and that one can find an element in the decomposition list of F_0 (say, A) such that:

$$L_0(A = 1) = L_1; \quad (23)$$

one can then write

$$F = L_0(A + z, B, \cdots). \quad (24)$$

The inference of the decomposition for F is slightly more complex than the previous case. First, we need to identify all the possible candidate functions in the decomposition list of F_0. The functions not appearing in the de-

composition tree of F_1 are all candidates. For each candidate, we need to test if Eq. (23) holds. For n candidate functions, this entails the computation of $2n$ cofactors.

If one function A is found that satisfies Eq. (23), we form the representation of $A + z$ and represent F as in Eq. (24).

Example 3. Consider the functions $F_0 = ITE(A, CD, B + C)$, $F_1 = CD$. The decomposition list of F_0 contains A, B, C, D. The tree of F_1 contains F_1, C, D. The functions not appearing in the tree of F_1 are then A, B. We observe that by assigning $B = 1$, however, $F_0 = ITE(A, CD, B + C) \neq F_1$, and that assigning $B = 0$ results in $F_0 = ITE(A, CD, C) \neq F_1$. The function B is then discarded. Assigning $A = 1$ instead results in $F_0 = ITE(1, CD, B + C) = CD = F_1$. A new function $Z = A + z$ is constructed, and F is represented by $ITE(Z, CD, B + C)$. □

Case 3.

We call **pushing** z into A the operation of moving z from the support of F (as in Eq.(12)) to the support of a non-trivial function A. As we can push z only in cases 1 and 2, we recognize the third case by failing the push of z. In this case, F must have a decomposition of type

$$F = L(z, B, C, \cdots). \tag{25}$$

The two cofactors are expressible only as

$$F_0 = L_0(B, C, \cdots), \quad F_1 = L_1(B, C, \cdots) \tag{26}$$

where $L_0 \neq L_1$. Notice that in this case the argument functions $B, C, \cdots$ may appear in the decomposition trees of both F_0, F_1, but not necessarily directly in their decomposition lists.

In this case, a decomposition of F can be constructed as follows. Consider the decomposition trees T_0 and T_1 of F_0 and F_1, respectively, and their intersection T_{01}. Recall that the nodes of T_{01} represent essentially functions that are common to the decomposition of F_0, F_1.

Let $(P, Q, R \cdots)$ denote the root children of T_{01}. These functions are disjoint-support and we can write F_0 and F_1 as

$$F_0 = R_0(P, Q, \cdots); \quad F_1 = R_1(P, Q, \cdots) \tag{27}$$

where in general R_0, R_1 are not prime functions. Consider the function $R(z, P, Q, \cdots) = z'R_0 + zR_1$. Clearly,

$$F = R(z, P, Q, \cdots). \tag{28}$$

Moreover, R must be prime. If, by contradiction, R had a decomposition, say, as

$$R(z, P, Q, \cdots) = S(z, U(P, Q), \cdots) \tag{29}$$

then the function U would have appeared in the decomposition of F_0 and F_1, and hence in T_{01}. Eq. (29) then represent the prime decomposition of F.

Example 4. Consider the case $F_0 = ITE(abc, d + e + f, g \oplus h)$, $F_1 = ITE(ab, e + f + g, h \oplus c)$. The intersection tree, shown in Fig. (3), indicates that F is expressible as:

$$A = ab \tag{30}$$

$$E = e + f \tag{31}$$

$$R = R(A, c, d, g, h, z) \tag{32}$$

An expression of R is $A'(E + dz' + gz) + A(c'hz + c(h \oplus (g + z)))$. □

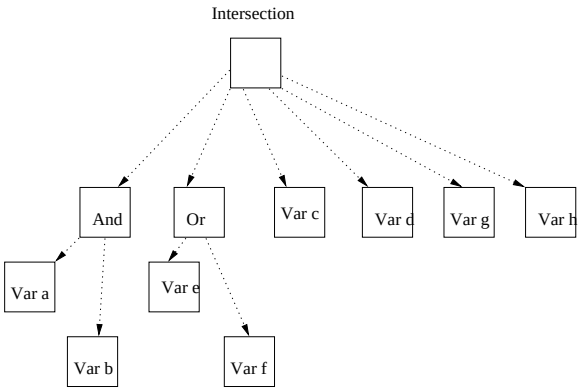

Figure 3. Intersection tree for cofactors of Ex. (4)

In practice, we do not construct an explicit representation of R, but only of its decomposition list.

5 Experimental results.

The procedures of this paper have been implemented in a C program named LODE, and tested on several combinational logic benchmarks. The CPU time was taken on a PC equipped with a 150 MHz Pentium, 256Kbyte of on-board cache and 32Mbyte of main memory.

Table (5) below compares the literal counts obtained by LODE against those obtained by `SIS` running `rugged_script`. Literals are counted from the factored form. It also reports the CPU time employed by LODE for the decomposition, in seconds. The CPU time includes the time required for the preliminary construction of the BDDs. IN all the reported examples, the time required by SIS was over one order of magnitude that of LODE.

6 Conclusions.

We have presented theoretical advances and procedures for the disjunctive decomposition of logic functions, starting from a BDD representation. The procedure returns a multiple-level netlist which, given the uniqueness of the proposed decomposition, is actually a normal form. A preliminary implementation of the ideas presented of this paper, has shown the practicality of the approach, in terms of CPU time and quality of results. We know of some simple improvements (like, the recognition of locally unate variables) which will further improve the quality of the circuit, at little CPU time expense.

We are currently exploring the implications of the canonicity of the proposed netlist in several directions, including rapid prototyping and sequential verification.

References

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

[2] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD*, pages 42–47, November 1993.

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

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

Benchmark	Literals		CPU
	LODE	SIS	time
MultiLevel			
9symml	74	223	0.14
alu2	357	357	0.12
alu4	843	689	0.31
apex6	901	744	0.46
apex7	291	246	0.26
C880	392	410	854
C1908	511	541	632
C432	1675	–	1231
CM138	32	31	0.11
CM150	47	51	0.11
CM151	25	26	0.10
CM162	56	49	0.09
CM42	36	34	0.11
CM82	28	24	0.08
CM85	67	46	0.11
cmb	38	51	0.12
f51m	102	91	0.08
frg1	195	–	0.18
lal	119	105	0.09
PARITYFDS	60	60	0.08
sct	91	78	0.10
term1	130	197	0.09
ttt2	223	216	0.16

Table 1. Experimental result and comparisons against SIS.

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

[6] 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.

[7] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on CAD/ICAS*, 6(6):1062–1081, November 1987.

[8] 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.

[9] V. Bertacco and M. Damiani. Boolean function representation using parallel-access diagrams. In *Sixth Great Lakes Symposium on VLSI*, March 1996.

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

[11] 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.