A BDD-based Approach to Multiple-level Combinational Logic Synthesis

Mats Carlsson
Swedish Institute of Computer Science
P.O. Box 1263, S-16428 KISTA, Sweden

January 4, 1995

1 Introduction

An experimental system, PLASYNT, for the synthesis of multiple-level combinational circuits is described. PLASYNT is written entirely in Prolog, but makes heavy use of formula manipulation primitives provided for a Boolean constraint solver integrated with the Prolog system [11].

PLASYNT starts from a description of a combinational circuit, specified as a truth-table in so-called Berkeley PLA format. Internally, the descriptions are stored and manipulated in a directed acyclic graphs representation known as *Binary Decision Diagrams* (BDDs[1]).

The program includes algorithms for finding optimal variable orderings in the BDDs, for assigning don't care values, for factoring Boolean functions to decrease the area of the circuits, and for mapping descriptions to specific technologies. PLASYNT currently maps descriptions to two different technologies: (i) networks of NAND-gates and inverters, and (ii) gates from Plessey's gate library [15].

Originally, the motivation for this work was to use a Boolean constraint solver integrated with Prolog in real-world applications. As it turned out, no constraint solving is used in this application; instead, the formula manipulation primitives of the constraint solver are heavily used. The current motivation for this work is to investigate BDD-based algorithms for factoring Boolean functions and for assigning don't care values, their performance, and how they relate to other approaches to multiple-level synthesis.

The paper is organized as follows: Section 2 describes the variable ordering problem and our approach to its solution; Section 3 describes our algorithm for assigning don't care values; Section 4 describes our approach to factoring Boolean functions; Section 5 discusses related and future work. We end with some conclusions.

2 BDDs and Optimal Variable Orderings

First of all, a brief description of BDDs is given. A BDD is a binary decision diagram where each node $\langle i, q, r\rangle$ is associated with a Boolean variable i and two fanouts q and r, where q corresponds to $i = 1$ and r to $i = 0$. Thus the node represents the function $i * q + \overline{i} * r$[2]. At the leaves of the graph are the two constants 1 and 0. A variable ordering is imposed such that all transitive fanouts of a node must have a strictly higher ordering index, except for the constant nodes. Furthermore all isomorphic subgraphs are shared. The resulting representation is unique for a given variable ordering. BDDs have been described at length in the literature [1, 9, 4, 3, 14], and allow very compact representation and efficient manipulation of Boolean formulas.

[1] The proper terminology for the BDDs we are using is Reduced Ordered Binary Decision Diagrams (ROBDDs) [9], but in this paper, we call them BDDs for short.

[2] We use the logical operators $p * q$ (and), $p + q$ (inclusive or), $p \oplus q$ (exclusive or), and $\overline{p}$ (not).

PLASYNT represents the specification of the current circuit as a vector of BDDs (one BDD for each output variable). The synthesized circuits are structurally related to the BDD representation. Therefore, the total size of the BDDs (number of nodes) is used as a prediction of the area of the resulting circuits.

Bryant observed in [9] that the size of the BDD for a function can vary radically for different variable orderings, and showed that finding a variable ordering that minimizes the size is NP-hard. Our approach to finding a suitable variable ordering is to use an alpha-beta search of the possible orderings, restarting the search each time a new local minimum is found.

3 Assigning Don't Care Values

The circuits are generally specified as incomplete functions, i.e., some output signals do not matter for some of the inputs $\vec{x}$. Such output signals are called *don't care values*. This is formalized by specifying each output signal as a function $f : \{0,1\}^{n+1} \mapsto \{0,1\}$ such that $f(\vec{x},0) \neq f(\vec{x},1)$ iff the output signal does not matter for the particular inputs $\vec{x}$. Any function $g : \{0,1\}^n \mapsto \{0,1\}$ satisfying $\forall \vec{x} : g(\vec{x}) \in \{f(\vec{x},0), f(\vec{x},1)\}$ will then be an admissible instance of the incompletely specified function. By choosing a suitable g, the size of the resulting circuit can be significantly decreased.

We have invented an algorithm which performs such an assignment of don't care values for functions represented as BDDs. Roughly, for each BDD node, the function tries to assign don't cares such that the two fanouts of the BDD become identical, in which case the BDD node can be replaced by the merged fanouts. If the two fanouts cannot be made identical, the BDD node is kept and the algorithm is applied recursively to its two fanouts.

Our algorithm is sensitive to the chosen variable ordering, and is integrated with the search for a good variable ordering as follows: For each variable ordering π to be considered, the BDD representation of the function vector is built, don't care assignment are assigned, and the total size of the resulting BDDs is used as the cost function for π.

Our top-down approach is in contrast to the *generalized cofactor* algorithm [16] which operates bottom-up. To evaluate the effectiveness of the don't care assignment algorithm, we ran it on some examples from the literature [17]. In the table below the effect of assigning all don't cares to 0 (**zeroalg**) is compared to the generalized cofactor algorithm (**gcalg**) and to our top-down algorithm (**tdalg**). The numbers show the BDD sizes for the best variable orderings found.

example name	**zeroalg** #nodes	**gcalg** #nodes	**tdalg** #nodes
inc	72	70	66
bw	106	101	92
misex3c	537	306	256
ex1010	1107	891	729
spla	693	621	613
pdc	677	404	271

4 Factoring

Factoring out common subexpressions from Boolean formulas is a crucial technique for minimizing circuits. A major part of the algorithms incorporated into the well-known multiple-level synthesis and minimization program MIS [7] is devoted to factoring.

The BDD representation does not seem particularly well suited for the methods used in MIS. Instead, we use a less powerful scheme for finding common subexpressions shared between the two fanouts of each BDD node. Our technique uses the functions q/r and $q - r$ denoting *restricted Boolean division* and *restricted Boolean subtraction*.

The function q/r is defined if $q = q * r$ as the solution X to the equation $q = X * r$. The function $q - r$ is defined if $q = q + r$ as the solution X to the equation $q = X + r$. The most general solutions to these equations can be derived [10] as:

$$\begin{aligned} q/r &\stackrel{\text{def}}{=} (U * \overline{r}) \oplus q \\ q - r &\stackrel{\text{def}}{=} (U * r) \oplus q \end{aligned}$$

and the don't care assignment algorithm described in Section 3 can be used to produce suitable solution instances. Our approach to factoring is based on the following equations:

$$\begin{aligned} \langle i, q, r \rangle &= (q + r) * \langle i, q/(q + r), r/(q + r) \rangle \\ \langle i, q, r \rangle &= (q * r) + \langle i, q - (q * r), r - (q * r) \rangle \\ \langle i, q, r \rangle &= q \oplus \langle i, 0, q \oplus r \rangle \\ \langle i, q, r \rangle &= r \oplus \langle i, q \oplus r, 0 \rangle \end{aligned}$$

where the division and subtraction operations are guaranteed to be defined, since $q = q * (q + r) = q + (q * r)$ and $r = r * (q + r) = r + (q * r)$ always hold. Other equations are also possible.

For every BDD node $\langle i, q, r \rangle$ which cannot be trivially translated to an input signal, an AND-gate, an OR-gate, an XOR-gate, or the complement of one of these, PLASYNT considers the unfactored form and the four factored forms obtained by the above equations, computes the total size of the BDDs for the five versions, and picks the version that yields the smallest total size. This greedy approach to minimization is somewhat risky: factoring a node might destroy other opportunities for sharing subexpressions, i.e. it can happen that a local decrease in size leads to a global increase.

In the table below, some circuits from [17] were translated by PLASYNT to networks of NAND-gates and inverters. Sizes obtained with factoring turned off ("nofact") or on ("fact") are given. As a comparison, we let MIS translate the same examples. We used the standard "Boolean" MIS script and its "minimal" gate library (NAND-gates, NOR-gates, and inverters).

example name	**nofact** #NAND	**fact** #NAND	**MIS** #NAND + #NOR
con1	25	23	17
squar5	60	52	62
inc	113	108	137
misex1	62	63	43
sao2	150	141	108
9sym	57	57	205
misex2	116	94	84
5xp1	73	77	101
clip	304	266	139
bw	174	180	176
rd84	96	96	362
vg2	172	119	75
duke2	581	543	336
table5	1373	1522	564
table3	1473	1993	598
e64	138	138	158
t481	74	32	33
misex3c	467	487	503
b12	102	84	69

Comments: The results of the factorization method varies a lot. In the best case (t481), it decreased the size 2.1 times, but in the worst case (table3), it increased it 1.35 times. This illustrates the risk of using a greedy algorithm instead of true optimization.

It was expected that MIS would get better results for all examples, and indeed in one case (table3) its solution is 3.33 times better than ours. To our surprise, in one case (rd84) our method got a solution which was 3.77 times better than that of MIS. This is partly due to our lack of knowledge about how to tune MIS to particular examples. More importantly, it illustrates the fact that all synthesis tools to some extent rely on heuristics, and sometimes one heuristic happens to be more successful than another.

5 Related and Future Work

The factorization of Boolean functions has been studied at least since 1969 [8, 6, 5, 12]. The treatment of don't cares in multiple-level logic synthesis has also been studied [13, 2]. The relation between the methods of the literature and our techniques remain to be clarified.

Another important technique, used in MIS and elsewhere, for further decreasing the size of a circuit is *phase assignment*. It consists in deciding for each gate whether to replace it by its complement (and add or delete the appropriate inverters) in order to minimize the total number of inverters. Currently, PLASYNT does not perform phase assignment.

At present, the supported gate libraries are wired into the program as Prolog clauses. It would be desirable to be able to dynamically load new gate libraries.

6 Conclusions

We have described an experimental system for the synthesis of multiple-level combinational circuits. It is written in Prolog and uses BDD primitives provided for a Boolean constraint solver integrated with the Prolog system. Thus such primitives can be very useful for real-world applications if made available to the user.

We have described novel approaches to don't care assignment and factorization of Boolean expressions, and presented benchmark results for these techniques. Boolean unification was used to derive restricted forms of Boolean division and subtraction.

Prolog proved to be an excellent language for this "symbol crunching" application. The size of the program is about 1000 lines of code.

References

[1] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[2] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Multilevel Logic Minimization Using Implicit Don't Cares. *IEEE Transaction on Computer-Aided Design*, 7(6):723–740, June 1988.

[3] J.P. Billon. Perfect Normal Forms for Discrete Functions. BULL Research Report No. 87019, Bull Research Center, June 1987.

[4] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45. ACM/IEEE, 1990.

[5] R.K. Brayton. Factoring Logic Functions. *IBM Journal of Research and Development*, 31:187–198, March 1987.

[6] R.K. Brayton and C.T. McMullen. The Decomposition and Factorization of Boolean Expressions. In *Proc. International Symposium on Circuits and Systems*, pages 49–54, April 1982.

[7] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transaction on Computer-Aided Design*, pages 1062–1081, November 1987.

[8] M.A. Breuer. Generation of optimal code for expressions via factorization. *Communications of the ACM*, 12(6), June 1969.

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

[10] W. Büttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[11] M. Carlsson. Boolean Constraints in SICStus Prolog. SICS Technical Report T91:09, Swedish Institute of Computer Science, 1991.

[12] G. Caruso. Near Optimal Factorization of Boolean Functions. *IEEE Transaction on Computer-Aided Design*, 10(8):1072–1078, August 1991.

[13] G. Hachtel and M. Lightner. Don't Care Conditions in Top Down Synthesis. In *IEEE International Conference on CAD*, pages 316–319, November 1987.

[14] J.C. Madre, O. Coudert, J.P. Billon, and C. Berthet. Formal Verification and Diagnosis of Digital Circuits Using a Propositional Theorem Prover. In *IFIP TC10 WG10.2 Working Conference on CAD Systems using AI Techniques*, pages 107–114. Tokyo, 1989.

[15] Plessey Semiconductor Ltd., CLA60000 Design Manual Volume 1, 1990.

[16] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE International Conference on CAD*, pages 130–133, November 1990.

[17] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. Technical Report, Microelectronics Center of North Carolina, January 1991. Benchmark examples for the 1991 MCNC International Workshop on Logic Synthesis.