Compilation of Optimized OBDD-Algorithms

Stefan Höreth
Department of Electrical Engineering / Computer Systems
Technical University of Darmstadt

Merckstr. 25, 64283 Darmstadt / Germany
E-mail: sth@rs.E-Technik.TH-Darmstadt.DE

Abstract

According to Bryant there exist basically two OBDD construction methods, namely Apply- and Compose-based approaches. In this paper we describe a compilation method that generates an optimized Apply-based OBDD-algorithm from a given combinational circuit description. The method is particularly useful in library-based synthesis- and verification environments. We also present a concise, machine independent measure for the efficiency of OBDD-construction methods. Experiments with our new method indicate a speedup of up to a factor 19 in the construction time for OBDDs while the maximum memory requirements are typically slightly smaller in comparison to conventional approaches.

1. Motivation

In recent years there has been excellent research on symbolic representations for boolean functions. Graph types like OBDDs[15, 2, 6, 7, 8] and *BMDs[14, 3, 5] have become state-of-the-art since their size is reasonable in many cases and there exist efficient basic manipulation algorithms. Various extensions have been proposed to adapt these graph types for special purposes. However, their impact was mostly on graph size and little has been done to improve complexity measures for construction- and manipulation-algorithms. Experiments reveal a hundredfold increased run-time for OBDD-construction compared to the ideal case where the final graph is known in advance and its nodes are simply introduced to the unique-table. Obviously it is impractical to strive for the latter case, however, it constitutes a lower bound and marks out the range for possible improvements against current methods. Typically OBDD-construction is based on the *ite*-Operator[4] and OBDDs are generated from netlists in a bottom-up manner. This can be seen as the most generic method, yet it is very inefficient in terms of (worst-case) time-complexity. In this paper we propose a compilation [1] method that generates optimized *Apply*-based OBDD-algorithms for the functional units of a combinational circuit description. While in the conventional approach (see Fig. 1) the *ite*-operator is applied to a flat gate-network, we rely on the compiled OBDD-operators which are much more efficient with respect to time complexity. The compiled algorithms

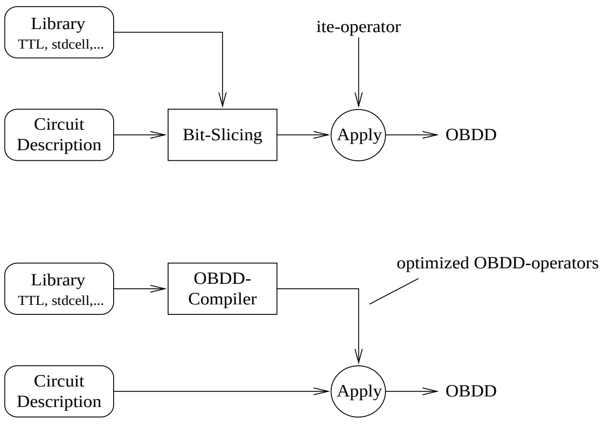

Figure 1. OBDD-construction: conventional- vs. compilation method

also avoid the computation of useless intermediate results, and are therefore preferable in order to diminish temporary memory requirements.

Our method is particularly useful in library-based synthesis- and verification environments. If the circuit is given as a large flattened net-list of basic gates, our method involves a preprocessing step for partitioning. In addition, we suggest a machine independent measure for the efficiency of OBDD manipulation algorithms that are based on a unique-table. The measure is very succinct and considers time- as well as space-requirements.

The organization of the paper is as follows: In section 2, we give a theoretical foundation for the compilation of optimized OBDD algorithms. The compilation method itself is described in section 3 followed by an outline of the preprocessing- and the assembling- step (section 4). Section 5 describes our efficiency measure in order to assess the algorithms we obtain. We also report experimental results in section 6, and conclude in section 7 with a discussion of future extensions.

2. Theoretical Background

The *if-then-else-* (*ite-*) Operator is commonly used as a basis for OBDD-construction. Another -more general- method is the Apply algorithm proposed by Bryant[6, 7]. It can be used for an arbitrary Boolean operation together with a pair of Boolean

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

functions (represented as OBDDs). The worst-case time-complexity of both methods is optimal since it is simply the product of the size of the argument graphs. However, if the corresponding operators are applied to combinational circuits, the resulting time-complexity is typically far from being optimal. This is due to the limited scope of both approaches: if the circuit has reconvergent paths, the computation of useless intermediate results cannot be avoided. For further illustration we consider the carry-output $c_{out} = (a \oplus b) \cdot c + a \cdot b$ of a full-adder:

With the *ite*-Operator one might use the nested calls

$$ite(\underbrace{ite(\underbrace{ite(\mathcal{A}, \bar{\mathcal{B}}, \mathcal{B})}_{a \oplus b}, \mathcal{C}, 0)}_{(a \oplus b) \cdot c}, 1, \underbrace{ite(\mathcal{A}, \mathcal{B}, 0)}_{a \cdot b})$$

in order to obtain the OBDD for output c_{out}. The corresponding worst-case time-complexity for this particular procedure is

$$\mathcal{O}(((n_A \cdot n_B) \cdot n_C) \cdot (n_A \cdot n_B)) = \mathcal{O}(n_A^2 \cdot n_B^2 \cdot n_C).$$

where n_A, n_B, n_C are the graph size of the OBDDs $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ respectively.

In contrast to that, the theoretically optimal algorithm needs at most $\mathcal{O}(n_A \cdot n_B \cdot n_C)$ steps according to the maximum number of different 3-tuples that can be generated from the set of nodes of the argument OBDDs.

In general there is a twofold reason for the compilation of functions with multiple outputs and multiple inputs. The first one has been mentioned above and is to avoid the computational overhead for an individual single-output function:

Theorem 1 *The optimal worst-case time-complexity for an (arbitrary) OBDD-operator is equal to the product of the graph size of its arguments.*

In general, binary OBDD-operators as well as the ternary *ite*-operator can not be optimal if applied to combinational circuits (except if the circuit has tree structure). Also, optimal worst-case time-complexity cannot be attained by functional composition. Worst-case time complexity for functional composition is quadratic in the size of the Boolean operator; it might as well involve a large constant overhead that does not show in the worst-case measure (see [7] for further details on functional composition).

The second reason is that, if multiple outputs can be computed simultaneously (i.e. depend on a common subset of input variables) a constant speedup can be achieved by avoiding repeated traversals of the argument graphs.

Theorem 2 *The computation of multiple outputs at a time saves at most a constant factor of the run-time.*

This constant can be quite large. Memory access is much more costly on modern microprocessors than register usage. It is important to notice that most of the OBDD-time is due to memory access/management. Therefore, OBDD-algorithms should make efficient use of the architecture of modern microprocessors by avoiding unnecessary *load* and *save* instructions and by better register utilization.

3. Compilation of OBDD-Algorithms

The Apply-Algorithm for two-input/single-output OBDD-operators has been described in [7]. It can be easily extended to work on argument-lists of arbitrary length. The pseudo-algorithm for such a generalized OBDD operator is outlined in Fig. 2.

It takes a list of OBDDs representing the argument functions of the operator 'op', and produces a list of output OBDDs (thus implementing a boolean function $f_{op} : \mathbb{B}^m \to \mathbb{B}^n$). The algorithm first checks the termination criteria for the multiple-output OBDD-algorithm. If this predicate is not satisfied the arguments of the operator are sorted according to symmetries of the boolean function f_{op}. Although sorting of the argument list is principally not necessary, it facilitates an efficient test of the computed table in the subsequent step and allows for better reuse of previously computed values (the computed table is typically implemented as a hash-based cache and stores results for a boolean operator and its set of arguments). Finally - if the search of the computed table was not successful - the algorithm requires two recursive function calls in order to evaluate (the cofactors) of the output OBDDs.

```
Apply( op, argument_list ) {
  if terminateP( op, argument_list )
    return corresponding_OBDD_list;

  /* sort argument list <=> symmetries( op ) */
  sorted_args = sort( op, argument_list );

  /* already computed ? */
  if computedP( op, sorted_args )
    return previously_computed_OBDD_list;

  /* recursive part */
  i = next_variable( sorted_args );
  c0_list = Apply( op, cof0_list(sorted_args, i));
  c1_list = Apply( op, cof1_list(sorted_args, i));

  /* find_or_add_node for pairs from c0/1_list */
  OBDD_list = edges( i, c0_list, c1_list );

  /* insert in computed table */
  insert_in_CT( op, sorted_args, OBDD_list );

  return OBDD_list;
}
```

Figure 2. Pseudo-Code for Apply-Algorithm

As can be seen from Fig. 2, algorithms for boolean operators differ in their termination criteria as well as in the symmetries of the corresponding boolean function. Compilation therefore mainly consists of two procedures, namely the generation of the termination criteria for the recursive multiple-output OBDD-algorithm and the detection of symmetries [19, 13, 18]. The efficiency of these procedures depends on the OBDD graph type under consideration. In the sequel we will focus on shared OBDDs which allow for equivalence test and negation in constant time. The compiled algorithms benefit from these properties since they can be used as a basis for succinct tests of the termination criteria.

3.1. Generation of the Termination Criteria

The termination criteria for an OBDD operator is a sufficient set of test sequences that can be applied to the arguments of the operator in order to implement all test cases of the truth table of the corresponding boolean function. A test sequence in turn is a set of OBDD-equivalence tests of the form $x = y$ or $x = \bar{y}$ where x is an OBDD from the operators argument list and y is either another argument or represents the boolean constant '1'. Note that both tests take constant time if shared OBDDs [11, 16, 17] with complement edges are used.

The termination criteria can be represented as a binary test graph where the terminal nodes are labeled with the return values and each inner node is labeled with an equivalence test. A special *empty*-leaf exists to indicate that no return value could be deduced. A path in the test graph from the root to a leaf constitutes a test sequence. The test graph can be reduced in the same manner as OBDDs if we disallow different nodes with the same label and the same set of predecessors and if we remove nodes where both predecessors are rooted at the same subgraph.

In Fig. 3 two different test graphs for an *add-with-carry-* (*adc*) operator are shown. Both graphs are sufficient to implement the truth-table of the full-adder function $f_{adc} = ((a \oplus b) \cdot c + a \cdot b, a \oplus b \oplus c)$. However, both graphs could be extended with further equivalence tests for the argument c. Although this is not necessary in order to implement the adder-function it could lead to an earlier detection of a return value for the *adc*-operator during the recursive traversal of its argument OBDDs. The second test graph has been derived from an expansion of the OBDD for the adder function. We therefore conjecture, that functions with a compact OBDD-representation also have a compact test graph. However, in the case of the *adc*-operator, there is a more succinct representation using pairwise equivalence tests of the arguments as it is also shown in Fig. 3 (adc_1).

Our method constructs the test graph using a top-down algorithm. First a total ordering on the equivalence tests is defined. This ordering is based on an optimized ordering of the variables in the OBDD [12, 20] for the operators function. Then the algorithm uses backtracking in order to evaluate the function under different equivalence assumptions for the arguments. Recursion stops if either a return value could be deduced or if a previously analyzed function is discovered. This case analysis is itself based on a BDD-package.

In the following we answer the question how many

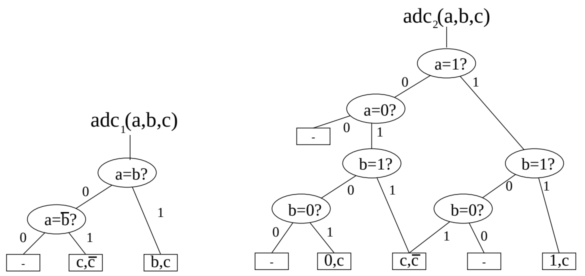

Figure 3. Different test-graphs for the adc-operator

test sequences exist (at most) for an arbitrary boolean operator.

Since every possible subset of arguments of a boolean operator might probably be used to compute a return value, we have to consider the power-set of the set of arguments. For each subset we can test each of its elements against the boolean constants or for pairwise equivalence (possibly negated) with a common argument. Since each of the elements of the argument subset can also be used as a return value, we have to check at most $n-1$ elements resulting in 4^{n-1} tests for a subset of size n. Since $\binom{N}{n} = N!/((N-n)! \cdot n!)$ is the number of different subsets of size n from an argument list of length N, we have:

Theorem 3 *The maximum number of different test sequences (based on binary equivalence-tests) for an N-argument OBDD operator is given by:*

$$\sum_{n=1}^{N} \binom{N}{n} 4^{n-1}$$

This number can be further reduced if we restrict the algorithm to pairwise equivalence test of the arguments or if we use the restriction to test the arguments against boolean constants. In both cases the termination criteria reduces to a maximum of $\sum_{n=1}^{N} \binom{N}{n} 2^{n-1}$ test sequences. Additionally, if we just consider the original set of arguments (instead of its power-set) the worst-case for the size of the minimal sufficient test set becomes 2^{N-1}. Clearly, the length of the longest test-sequence grows linear in the latter case, while it is quadratic in the worst-case.

Note that the actual size of the test graph depends on the function under consideration. In the sequel we refer to a test graph that utilizes all possible test sequences as optimized for (execution) speed. A test graph derived by an expansion of the functions OBDD is denoted as optimized for size.

3.2. Normalization of recursive Function-Calls

It is essential for an efficient implementation of an OBDD operator to use a computed-table [4]. The hit-ratio of the computed table can be improved if we identify symmetry sets among the set of variables of the corresponding boolean function. A symmetry set is a set of variables that are pairwise interchangeable (possibly using negation) without affecting the boolean function. Once a symmetry set has been identified, the corresponding arguments of the operator can be sorted according to an (total) ordering defined on the OBDDs (in the implementation for shared OBDDs we can simply use the pointer to a memory location).

In our current implementation we use an exhaustive symmetry detection but restrict the search to functions depending on a limited number of variables. Note that the detection of symmetry sets interacts with the generation of the termination criteria: if some equivalences of the arguments of the operator exist, the underlying boolean function is possibly revealing new symmetries.

For example the boolean function $f(a,b,c) = a \cdot \bar{b} + \bar{a} \cdot c$ has no symmetries, but if $b = c$ (e.g. during OBDD traversal for the operator) the function reduces to $f(a,b,b) = a \oplus b$ which is symmetric in a and b (c).

Therefore the test for termination and sorting of arguments are actually meshed. To make sure that the compiled algorithms are competitive against binary operators it is sufficient to restrict symmetry detection to functions depending on a small number of variables. Empirically we have found that the restriction to $k \leq 4$ variables is feasible and does not increase compilation time significantly. Moreover, if symmetry-detection is extended to a large number of variables, there is a trade-off between the time needed for sorting and the improvement of the traversal algorithm.

4. Other Compilation Steps

In section 3.1 we have shown how the test graph for an operator can be generated. If the number of arguments of an operator is too large it becomes impractical to optimize the test graph for execution speed. In this case we can either reduce the number of test sequences or use a preprocessing step to partition the boolean function.

This partitioning step is equivalent to a synthesis problem for Field-Programmable Gate-Arrays (FPGAs) which are based on Look-Up tables (LUT). In our implementation we use a modified bin-packing procedure that has been presented in [10]. The algorithm was originally developed for FPGA synthesis and uses dynamic programming to (heuristically) optimize the partitioning into LUTs of a given size. Empirically we use LUTs with $n \leq 8$ inputs as a preprocessing parameter. Each LUT is then compiled into an OBDD-operator.

Besides the preprocessing- and the compilation step there is a final assembling stage where code for the test graph and the symmetry information is generated. The code is output in the programming language 'C' and further code-optimizations are left to a commercial C-compiler.

5. Efficiency of the compiled Algorithms

Efficiency measures that focus on either time or space requirements are not adequate to assess a graph-based verification methodology. We complete the traditional set of criteria by a concise measure for the efficiency of OBDD construction algorithms. This measure is based on the area $A_{\mathcal{G}}$ (product of time and active nodes $n(t)$) required during OBDD traversal. The value $A_{\mathcal{G}}$ can be obtained by sampling the number of active nodes. If related to the minimal area A_{opt}, this measure becomes machine independent. A_{opt} is obtained from the mean access time for the unique-table T_U/N_U and the size $n_{\mathcal{G}}$ of the OBDD $\mathcal{G}$:

$$\frac{A_{opt}}{A_{\mathcal{G}}} = \frac{\frac{1}{2} n_{\mathcal{G}}^2 \frac{T_U}{N_U}}{\int\limits_0^{t_{\mathcal{G}}} n(t)dt} \approx \frac{\frac{1}{2} n_{\mathcal{G}}^2 \frac{T_U}{N_U}}{\sum\limits_i \frac{n_i + n_{i+1}}{2} \cdot (t_{i+1} - t_i)} \qquad (1)$$

Figure 4 shows time- and space-requirements during OBDD construction for a 32-bit Binary-to-BCD-to-7-segment converter. This circuit is based on the TTL-library[21] modules SN74185 and SN7449 which were compiled and OBDD construction for the circuit (Modules) has been compared to an implementation based on binary gates (Gates). Figure 4 also indicates the optimum algorithm, that would simply hash the nodes of the final OBDD $\mathcal{G}$. The number of active nodes $n(t)$ includes the graph in the computed table as well as OBDDs kept temporarily at fanout-nodes of the circuit description.

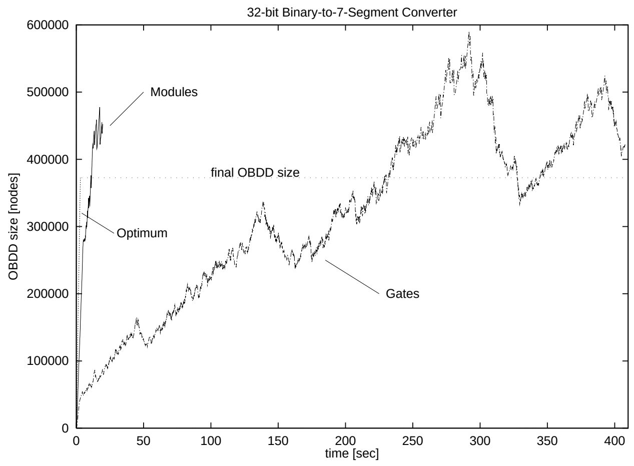

Figure 4. Sampling of time- and space-requirements

From the samples of Figure 4 we can compute the efficiency $A_{opt}/A_{\mathcal{G}}$. Table 1 shows the results for a 16- and 32-bit Binary-to-7-Segment converter. The efficiency for the traditional method (Gates) is less than one percent in either example whereas the compiled algorithms (Modules) make much better use of the hardware resources.

Table 2 shows the results for run-time and graph-size. Run-times for the methods 'Gates' and 'Modules' have been obtained on a SPARCstation20/71;

Efficiency	$A_{opt}/A_{\mathcal{G}}$		
	Gates	Modules	Improvement
bin7seg16	0.2%	9.0%	45
bin7seg32	0.44%	8.2%	18.6

Table 1. Efficiency Measures

'max.Gates' and 'max.Modules' report the maximum graph size during OBDD construction for the respective method. The size of the final OBDD is given in column 'final'.

time	Gates	Modules	Speedup
bin7seg16	1.92s	0.75s	2.5
bin7seg32	6m50s	21s	19.5
size [nodes]	max.Gates	max.Modules	final
bin7seg16	27526	7127	3785
bin7seg32	590346	477498	372646

Table 2. Run-time and graph size

Table 3 shows the size of the object code for the SPARC-microprocessor as well as compilation times. In column 'speed' the operators have been optimized for execution speed while in column 'size' the code-size was the main concern. The run-times and efficiency measures that have been achieved for different code-optimizations of the TTL library-modules are reported in table 4. Again, columns 'speed', and 'size' refer to the different code optimizations from table 3.

Compilation	speed		size	
	code	time	code	time
ite	18k	1.3s	0.7k	0.3s
and2,ior2,...	3.5k	0.5s	0.7k	0.3s
sn74185	29k	57s	1.5k	1.5s
sn7449	10k	11s	1.5k	0.8s

Table 3. Compilation time and size of object code

All compilation results have been obtained with a prototype implementation written in Common-Lisp. Compilation time does not include the time needed by the commercial C-compiler which generated the object code from our intermediate C-code.

6. Experimental Results

We have applied our compilation technique to modules from different design libraries. Table 5 presents the results for TTL- as well as standard-cell designs.

The TTL-modules used in our examples include code converters, decoders, full adders, multipliers and Wallace trees. Circuits `bin7segx` and `bcdxbin` are code converters, `wallx` is a Wallace-tree multiplier for two's complement numbers. These circuits as well as the library modules are described in [21]. TTL-modules that are specified by a truth-table have been replaced by a circuit description based on an exact minimization obtained with the Espresso logic-minimization tool.

Code	speed	efficiency	size	efficiency
bin7seg16	0.75s	9.0%	0.9s	6.1%
bin7seg32	21s	8.2%	28.2s	5.4%

Table 4. Different Code-Optimizations

The standard-cell examples use design primitives such as multiplexers, demultiplexers, 1-bit adders and basic gates. Most of the standard-cell designs have been generated with the Synopsys synthesis tool, except for `mcalu32b` which is a complex industrial ALU. The remaining circuits are an unsigned array-multiplier (`mulx`), divider (`divx`) and subtracter (`subx`). A description of the standard-cell library can be found in [9].

In all the experiments an OBDD for the primary-outputs was constructed in two different ways, once using the compiled OBDD-operators (TTL, stdcell) and again using the conventional approach based on the *ite*-operator (gates). Column 'final' gives the size of the final OBDD while columns 'time' presents the run-time for both methods on a SPARCstation20/71. 'Efficiency' describes the usage of hardware resources for both construction methods in percent of the (theoretical) optimum algorithm (please refer to section 5 for a definition of 'efficiency'). Columns 'speedup' and 'improve.' are the relative improvements for the measures from column 'time' and 'efficiency' respectively. Columns 'in', 'out' give the number of primary inputs and primary outputs of the circuit.

It is interesting to see that in all examples the relative speedup for a particular circuit improves with the size of the OBDD for its output functions. At the same time efficiency drops if the 'granularity' of the library modules is small compared to the size of the circuit. Therefore run-times could be further improved if larger portions of the circuit were compiled.

7. Conclusions and Future Research

We have presented a compilation method for library-based verification- and synthesis environments. Our method obtains large speedups for MSI functions like TTL modules. Even for standard-cell designs where the library-cells contain just a few logic gates, the method significantly improves the OBDD construction time. Our method does not increase memory requirements. Compiled operators generate much less intermediate results, thus the number of entries in the computed table can be reduced. Overall memory requirements are typically slightly smaller compared to conventional methods. Compilation of OBDD-operators is fast and could therefore be used as an on-the-fly technique in incremental designs. This requires a more sophisticated preprocessing technique which is a concern of our current research. We also investigate an improved symmetry detection for boolean functions which

circuit	in	out	Library	final [nodes]	time	efficiency	speedup	improve.
bin7seg16	16	35	TTL gates	3785	0.75s 1.92s	9.0% 0.2%	2.5	45
bin7seg32	32	70	TTL gates	372646	21s 6m50s	8.2% 0.44%	19.5	18.6
bcd5bin	20	16	TTL gates	2016	0.82s 2.1s	4.8% 0.1%	2.5	48
bcd10bin	40	34	TTL gates	226999	27s 6m58s	2.7% 0.14%	15.5	19.3
wall8	16	16	TTL gates	22744	3.0s 7.1s	2.4% 0.6%	2.4	3.3
mcalu32b	77	35	stdcell gates	18709	8.6s 25.8s	2% 0.1%	3	20
mul8	16	16	stdcell gates	9257	1.8s 3.0s	1.4% 0.6%	1.7	2.3
mul12	32	32	stdcell gates	605882	127s 287s	1.3% 0.7%	2.26	1.8
div8	16	9	stdcell gates	3195	2.8s 4.1s	0.08% 0.04%	1.46	2
div12	24	13	stdcell gates	87241	290s 455s	0.02% 0.01%	1.57	2
sub64	129	65	stdcell gates	6432	1.3s 2.0s	21% 2.8%	1.54	7.5

Table 5. Experimental Results

can further speed-up the compilation process itself as well as the generated code.

References

[1] A. Aho and J. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C(27):509–516, June 1978.

[3] Bahar, Frohm, Gaona, Hachtel, Macii, and Somenzi. Algebraic decision diagrams and their applications. *International Conference on Computer Aided Design*, pages 188–191, 1993.

[4] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD-package. *Design Automation Conference*, pages 40–45, 1990.

[5] R. Bryant and Y.-A. Chen. Verification of arithmetic functions with Binary Moment Diagrams. *Design Automation Conf.*, pages 535–541, 1995.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C(35):677–691, Aug. 1986.

[7] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, Feb. 1991.

[8] R. E. Bryant. Symbolic boolean manipulation with Ordered Binary-Decision Diagrams. *ACM Comp. Surveys*, 24:293–318, 1992.

[9] European Silicon Structures Ltd. *ES2 ECPD07 Library Databook*, 1993.

[10] R. Francis, J. Rose, and Z. Vrasenic. Chortle-crf: Fast technology mapping for Look-Up Table-based FPGAs. *Design Automation Conference*, pages 227–233, 1991.

[11] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on Binary Decision Diagrams. *International Conference on Computer Aided Design*, Nov. 1988.

[12] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of Binary Decision Diagrams for the application of multi-level logic synthesis. *Proceedings of The European Design Automation Conference*, pages 50–54, Feb. 1991.

[13] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. *International Conference on Computer-Aided Design*, pages 472–475, 1991.

[14] Y. Lai and S. Sastry. Edge-valued Binary Decision Diagrams for multi-level hierarchical verification. *International Conference on Computer-Aided Design*, pages 188–191, 1993.

[15] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Tech. J.*, 38:985–999, July 1959.

[16] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. *Proceedings of the 25th ACM/IEEE DAC*, pages 205–210, June 1988.

[17] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with attributed edges for efficient Boolean function manipulation. *Proceedings of the 27th ACM/IEEE DAC*, 1990.

[18] D. Möller, J. Mohnke, and M. Weber. Detection of symmetry of boolean functions represented by ROBDDs. *International Conference on Computer Aided Design*, pages 680–684, 1993.

[19] S. Panda, F. Somenzi, and B. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. *International Conference on Computer-Aided Design*, pages 628–631, 1994.

[20] R. Rudell. Dynamic variable ordering for Ordered Binary Decision Diagrams. *International Conference on Computer Aided Design*, pages 42–47, 1993.

[21] Texas Instruments. *The TTL Data Book for Design Engineers.*