Using BDDs to Verify Multipliers

Jerry R. Burch
School of Computer Science
Carnegie Mellon University

Abstract

Bryant's Binary Decision Diagrams (BDDs) [6] have been successfully used for verifying combinational circuits [12, 18]. However, multiplier circuits are difficult to verify using BDDs since the size of the BDD representing multiplication grows exponentially in the number of input bits [6, 7].

This paper presents a method for using BDDs to verify multipliers that avoids this exponential complexity. The method has been used to verify a 16 by 16 bit combinational multiplier, the C6288 circuit from the ISCAS85 benchmarks [5]. This is the only ISCAS85 benchmark circuit that could not be verified using the techniques described by Fujita *et al.* [12] and Malik *et al.* [18].

1 Introduction

Bryant's Binary Decision Diagrams (BDDs) are a method for efficiently representing boolean functions [6]. They have been successfully used for verifying combinational circuits [12, 18]. However, multiplier circuits are difficult to verify using BDDs; examples in the literature are no larger than 8 by 8 bit multipliers [10, 13, 17, 19]. This is because the size of the BDD representing multiplication grows exponentially in the number of input bits [6, 7].

This paper presents a method for using BDDs to verify multipliers that avoids this exponential complexity. Normally the outputs of an n by n bit multiplier circuit are represented by BDDs with $2n$ variables, since the circuit has $2n$ inputs. In the method described here, the outputs of the circuit are represented by a BDD with $2n^2$ variables, instead. The size of this BDD is cubic in n.

The method is *conservative*. The verifier classifies circuits as being either *correct* or *possibly incorrect*. It will never classify an incorrect circuit as correct. However, the method may classify a correct circuit as possibly incorrect. We characterize a large class of multiplier circuits for which the method is *exact;* these circuits are classified as correct if and only if they are, in fact, correct. We also discuss ways the method can be modified to make it exact for a larger class of circuits.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976. The National Science Foundation also sponsored this research effort under contract numbers CCR-8722633 and MIP-8858807

2 Verifying combinational circuits

Formal verification involves checking that an implementation satisfies some specification. If the implementation in question is a combinational circuit with input wires I and output wires O, then the specification is a set of boolean functions over I, one for each element of O. The circuit is correct relative to the specification if the function produced by the circuit at each output node is equivalent to the corresponding function in the specification. The specification can be generated by analyzing another circuit, or constructed by a special purpose program. In some applications it is useful to also keep track of a set of input vectors (the *don't care* set) for which the output of the circuit is irrelevant. For the multiplier circuits considered in this paper, the don't care set is empty.

Since BDDs are a compact representation for many of the functions that arise from combinational circuits, and have efficient algorithms for equivalence checking and boolean operations, they can be the basis for powerful verification tools. The standard method for this kind of verification is described in [12] and [18]. These papers also consider the problem of automatically generating good variable orderings for BDDs. Our tools do not address this problem; variable orderings must be produced by hand.

Multipliers are an important class of circuits that cannot be verified efficiently with the standard BDD-based verification method because multiplication cannot be represented compactly with BDDs. The new method described here makes BDD-based verification of multipliers much more efficient. The empirical results in section 6 show a speed up of more than two orders of magnitude for 16 by 16 bit multiplication.

3 Representing multiplication

If $x_0, \ldots, x_{n-1}$ are binary values, we write

$$\langle x_{n-1}, \ldots, x_0 \rangle$$

to denote the n-bit unsigned integer

$$\sum_{i=0}^{n-1} 2^i x_i.$$

Given input variables $\{x_i\}$ and $\{y_i\}$, define $\{z_i\}$ such that

$$\langle z_{2n-1}, \ldots, z_0 \rangle = \langle x_{n-1}, \ldots, x_0 \rangle \times \langle y_{n-1}, \ldots, y_0 \rangle.$$

Then $\langle z_{2n-1}, \ldots, z_0 \rangle$ is also equal to

$$\sum_{i=0}^{n-1} 2^i \times \langle x_i \wedge y_{n-1},\ x_i \wedge y_{n-2}, \ldots, x_i \wedge y_1,\ x_i \wedge y_0 \rangle.$$

Bryant has shown that the BDDs needed to represent z_n and z_{n-1} have size exponential in n, for any variable ordering [7]. Thus, it is impractical to verify multipliers using the set $\{z_i\}$ as a specification.

The key idea of this paper is to avoid using $\{z_i\}$ as the specification, but instead to use a different set of formulas $\{z'_i\}$ that depend on $2n^2$ variables instead of $2n$ variables. The set $\{z'_i\}$ is defined by setting $\langle z'_{2n-1}, \ldots, z'_0 \rangle$ equal to

$$\sum_{i=0}^{n-1} 2^i \times \langle x_{i,n-1} \wedge y_{n-1,i},\ x_{i,n-2} \wedge y_{n-2,i}, \ldots, x_{i,1} \wedge y_{1,i},\ x_{i,0} \wedge y_{0,i} \rangle.$$

In order to show that $\{z'_i\}$ is a useful specification for verifying multipliers we must show that the basic verification method described in section 2 can be modified to use $\{z'_i\}$, and that $\{z'_i\}$ can be represented by BDDs that are relatively small. This is done in the following two sections.

4 Verifying multipliers

Figure 1 is a simple multiplier circuit. We verify this circuit by verifying the modified version shown in figure 2. This circuit is produced by making a new input for each of the fanouts of the inputs in figure 1. We call this operation *fanout splitting.* The modified multiplier satisfies the specification $\{z'_i\}$ described in the previous section. Thus, the verification procedure consists of two steps. First, modify the multiplier circuit by doing fanout splitting on each input node. This modification could be automated in order to insure that no errors are introduced. Second, verify the resulting circuit against the specification $\{z'_i\}$ using the standard method described in section 2.

We have shown how to verify a particular multiplier circuit using our method, but this leaves the question of how general the method is. The method can be used without modification for multiplier circuits that satisfy the following condition.

Condition 1 *The multiplier is unsigned and combinational, and computes products by adding together the partial products $x_i \wedge y_j$ for all inputs x_i and y_j.*

The kind of adder circuit used to add the partial products $x_i \wedge y_j$ does not matter; it could be a simple multiplier array, a binary adder tree, a Wallace tree, unary to binary converters (Dadda multiplier), or any other kind of adder. Any such circuit, when modified according to the first step of the verification method, satisfies the specification $\{z'_i\}$.

Not all multiplier circuits satisfy condition 1, however. If the method is applied to such a circuit, it will report that the circuit is incorrect even if the circuit is, in fact, correct. Thus the method is not very useful in these cases, but it is guaranteed to be *conservative.* That is, it will never report that a circuit is correct when it is not. Modifying our verification method so that it can be applied more usefully to other kinds of multipliers is discussed in section 7.

5 BDD sizes

Our method for verifying multipliers is efficient because the BDDs representing $\{z'_i\}$ are much smaller than the BDDs for $\{z_i\}$. This is somewhat counter-intuitive since the BDDs for $\{z'_i\}$ have many more variables. For example, Cerny and Mauras [8] apparently assumed that verifying a multiplier was made more difficult by modifying the circuit in a way similar to applying fanout splitting (each partial product was treated as an independent input).

We consider two different variable orderings for BDDs representing $\{z'_i\}$. The orderings are described by listing the variables, with the first variable being the label of the root of the BDD. The first is the low-to-high ordering:

$x_{0,0}, y_{0,0},$
$x_{0,1}, y_{1,0}, x_{1,0}, y_{0,1},$
$\ldots,$
$x_{0,n-1}, y_{n-1,0}, \ldots, x_{i,n-1-i}, y_{n-1-i,i}, \ldots, x_{n-1,0}, y_{0,n-1},$
$x_{1,n-1}, y_{n-1,1}, \ldots, x_{i,n-i}, y_{n-i,i}, \ldots, x_{n-1,1}, y_{1,n-1},$
$\ldots,$
$x_{n-2,n-1}, y_{n-1,n-2}, x_{n-1,n-2}, y_{n-2,n-1},$
$x_{n-1,n-1}, y_{n-1,n-1}$

The second is the high-to-low ordering:

$x_{n-1,n-1}, y_{n-1,n-1},$
$x_{n-2,n-1}, y_{n-1,n-2}, x_{n-1,n-2}, y_{n-2,n-1},$
$\ldots,$
$x_{1,n-1}, y_{n-1,1}, \ldots, x_{i,n-i}, y_{n-i,i}, \ldots, x_{n-1,1}, y_{1,n-1},$
$x_{0,n-1}, y_{n-1,0}, \ldots, x_{i,n-1-i}, y_{n-1-i,i}, \ldots, x_{n-1,0}, y_{0,n-1},$
$\ldots,$
$x_{0,1}, y_{1,0}, x_{1,0}, y_{0,1},$
$x_{0,0}, y_{0,0},$

The total size of the BDD needed to represent all of the outputs of an n by n bit multiplier using the low-to-high variable ordering is exactly $2n^4 - 13\frac{2}{3}n^3 + 198n^2 - 2144\frac{1}{3}n + 9894$ for $n \geq 17$. This result has been verified empirically for n up to 24.

The quartic rate of growth in the node count can be understood by considering the circuit in figure 2. There are $2n$ BDDs needed to represent the outputs of this circuit, and each BDD depends on up to $2n^2$ variables. So, in order to explain the quartic rate of growth, we must show that the number of nodes at each level of each of these BDDs is no larger than $O(n)$. Consider the number of nodes at the $x_{1,3}$ level of the BDDs for each of the $\{z'_i\}$. For $0 \leq i \leq 3$, the width of the BDD is zero, since these outputs do not depend

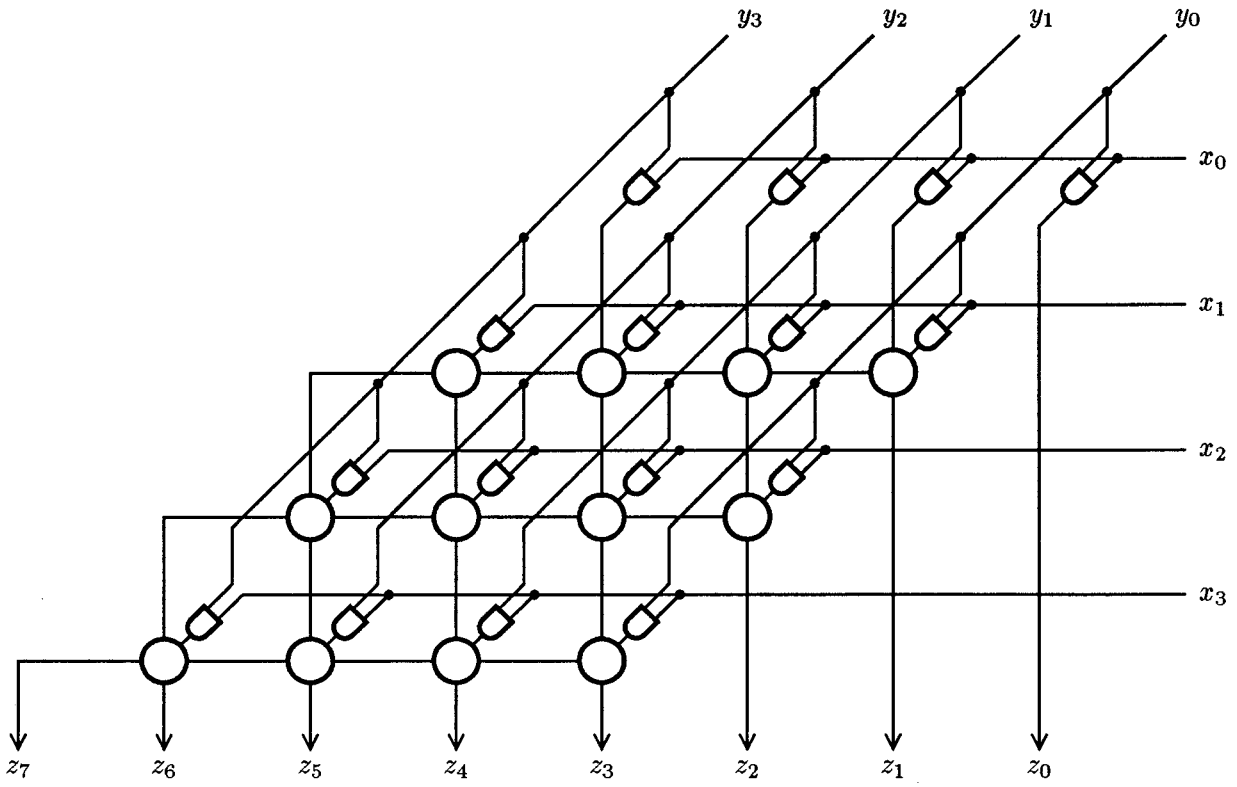

Figure 1: Simple 4 by 4 bit multiplier circuit. The circles represent adders that take up to three addend inputs from the top, right and upper-right; and output the sum to the bottom and the carry to the left.

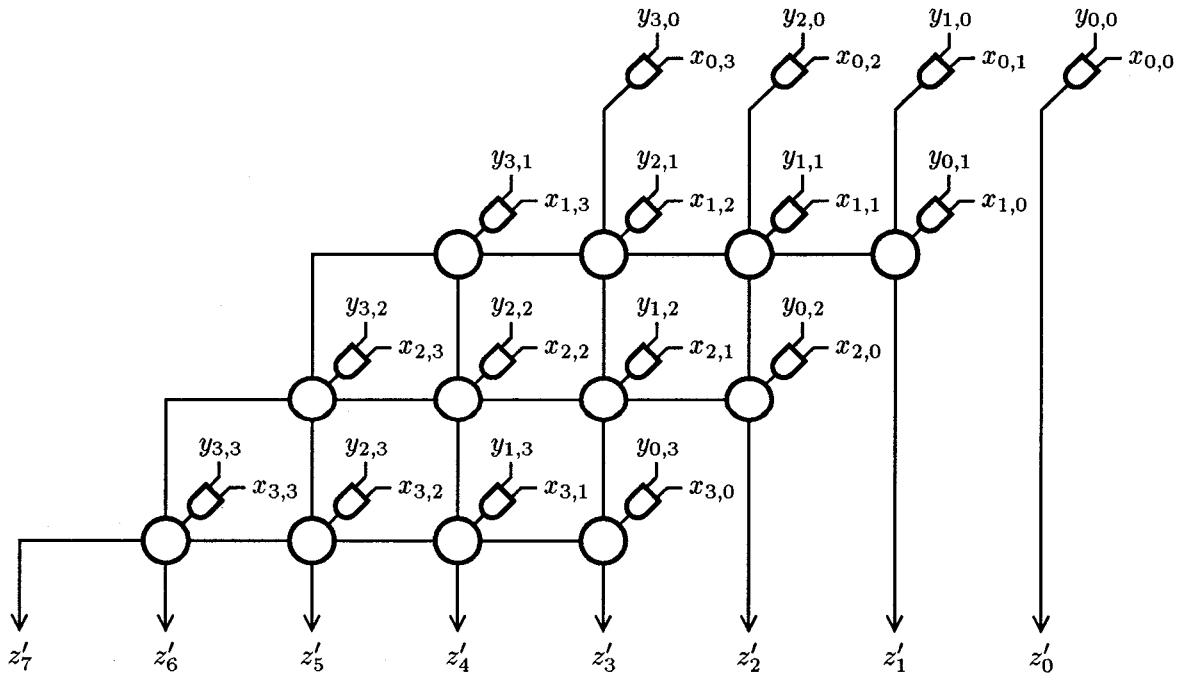

Figure 2: Result of applying fanout splitting to each input node of the multiplier in figure 1.

on $x_{1,3}$. For $i \geq 4$, all that has to be "remembered" from the variables preceding $x_{1,3}$ is the value of the carry from the z'_3 column to the z'_4 column. Since there are only 3 carry lines, this value ranges between 0 and 3. Thus, there are no more than 4 nodes at the $x_{1,3}$ level. A similar argument applies to the remaining variables of the BDD. This explains the quartic growth in the number of BDD nodes.

One way to reduce the number of BDD nodes needed to represent a circuit with multiple outputs is to arrange the variable ordering so that the BDDs for different outputs can share nodes. For example, the BDD representing all the outputs of an m bit adder circuit has $O(m^2)$ nodes if the low order bits are closest to the root in the variable ordering, but has $O(m)$ nodes if the high order bits are closest to the root [1, 2]. As an analogy to this, one might conjecture that the high-to-low ordering would reduce to $O(n^3)$ the number of BDD nodes needed to represent all of the outputs in figure 2.

This conjecture is correct. The total size of the BDD needed to represent all of the outputs of an n by n bit multiplier (that satisfies condition 1) using the high-to-low variable ordering is exactly $4n^3 - 6n^2 - 4n + 12$ for $n \geq 2$. This result has been verified empirically for n up to 44. Because the high-to-low ordering has a slower growth rate, we use this ordering in our verification tools.

6 Experimental results

This section describes the results of applying our verification methodology to the C6288 circuit in the ISCAS85 benchmark suite. C6288 is a 16 by 16 bit unsigned multiplier circuit. Let $x_0, \ldots, x_{15}$ and $y_0, \ldots, y_{15}$ be the names of the inputs to the circuit. Some simple analysis of the netlist reveals that the circuit computes all of the partial products $x_i \wedge y_j$. All of these partial products are added together to compute the outputs of the circuit, so the circuit satisfies condition 1. This is sufficient to insure that our verification method will indicate the circuit is correct if and only the circuit is actually correct. If the circuit did not have these properties, then the reverse implication might not be true. Since the method is conservative, the forward implication is true for any circuit.

Attempts to verify C6288 using methods like that in section 2 are described in [12] and [18]. In both cases the verification could not be completed because the BDDs that had to be constructed were too large. All of the other circuits in the suite were successfully verified by both groups.

Our implementation is based on the BDD package of Brace, Rudell and Bryant [4], which is written in C. There have been several modifications made to the package [16], including the ability to call routines from programs written in the LISP dialect T [20]. The verification programs we use are written in T, with only the BDD package being written in C.

We were able to verify C6288 in about 40 minutes on a 12 Mbyte Sun 3/60. About 3 minutes of this time was used to construct the BDDs for the specification. Nearly all of the remaining time was spent constructing the BDDs to represent the functions computed at each node in the circuit. The algorithm for computing the specification is specifically designed to reduce the size of the intermediate BDDs that are constructed. This is why such a small fraction of the verification time was used for computing the specification. The algorithm is analogous to a multiplier circuit that passes the partial products through one stage of unary to binary converters (like those used in a Dadda multiplier) and adds the results with a binary, ripple-carry adder tree.

Several researchers have reported performance figures for using standard BDD-based verification techniques to verify small combinational multipliers [10, 13, 17, 19]. The fastest of these was able to verify an 8 by 8 bit multiplier in 36.1 seconds [19]. However, results on smaller examples show that the verification time is growing exponentially with a base of more than 3.5. Extrapolating this gives a conservative estimate of at least 225 hours to verify a 16 by 16 bit multiplier using this method. Thus, the method described in this paper offers a speed improvement of more than 300 times.

7 Other multiplication algorithms

The specification $\{z'_i\}$ can be used to verify combinational multipliers that satisfy condition 1 (see section 4). However, slightly different methods are necessary for signed multipliers, and for multipliers that do not compute the partial products of all the circuit inputs. In these cases, analogous to the case described earlier, we modify the circuit by applying fanout splitting, which increases the number of inputs. However, a different specification must be used.

As an example, consider a signed, radix 2, Booth encoded multiplier [14]. We apply fanout splitting to each of the inputs, and to the outputs of the recoders. Fanout splitting on nodes that are not inputs requires replicating the circuitry (the recoders in this case) that drives each node being split. We have constructed a specification for an n by n bit multiplier of this kind that requires $3n^2$ variables and exactly $10n^3 - 11n^2 + n + 8$ BDD nodes, when $n \geq 2$. This result has been empirically confirmed for n up to 16.

However, the size of the BDDs depends on details of the adder/subtractors used in the multiplier. To understand this, notice that the functionality of the circuit in figure 2 is not changed by permuting the carry signals between any two adjacent columns. Thus, in the low-to-high variable ordering, the BDDs for the circuit only have to "remember" how many carry signals are active, not complete information about exactly which carry signals are active (see section 5). The carry/borrow signals in the Booth multiplier described above also have this property. However, if we reverse the sense of (negate) the carry/borrow signals when subtracting, but not when adding, then the resulting circuit does not have this property, and the size of its BDDs is exponential in n. Thus, our method is less robust for this class of multipliers than for multipliers that satisfy condition 1.

8 Related work

Though discovered independently, the method described here is similar, on the surface, to the verification method of Friedman [11]. That method uses pBDDs, which are a generalization of BDDs that allow variables to appear more

than once along a path from the root to a leaf. The pBDD used for representing multiplication is analogous to the BDD in our method with the low-to-high variable ordering. Thus, the total number of nodes required in Friedman's method to represent all outputs of a multiplier is $O(n^4)$, not $O(n^3)$ as in our method.

The most important differences between Friedman's method and the method presented here are due to the properties of pBDDs. Since pBDDs are not a canonical form, there is no guarantee that the pBDD formed by analyzing a multiplier circuit will have small size. Also, equivalence checking for pBDDs is NP-complete.

Friedman gave empirical results for verifying output bits 3 through 8 of a 16 by 16 bit multiplier. The verification times grew exponentially in the order of the output bit, even though the size of the multiplier remained constant.

Simonis has described a method for verifying multipliers that is uses constraint logic programming [21]. The method has very good performance; certain classes of n by n bit multipliers can be verified in $O(n^2)$ time. However, Simonis' basic method does not apply to as broad of a class of multipliers as the method here. Simonis describes extensions to the method, but it is not clear that they are as fast as the original method. Also, it appears that Simonis' method would not integrate as well as our method with BDD-based approaches for verifying combinational logic in circuits that contain multipliers.

9 Summary and discussion

We have described a method for using BDDs to verify multipliers that avoids constructing BDDs of exponential size. The method can be applied to a large class of unsigned combinational multipliers and has been used to verify a 16 by 16 bit multiplier. The key idea of the method is to use fanout splitting to systematically increase the number of variables in the BDDs. Despite increasing the number of variables, this makes it possible to represent all of the outputs of an n by n bit multiplier with a total of $O(n^3)$ BDD nodes.

Fanout splitting may be a useful technique for verifying other classes of combinational circuits that cannot be compactly represented with BDDs directly. It might also be applied to sequential circuits by treating the use of the same input value in several different cycles as a form of fanout. One weakness is the need to construct specifications that are equivalent to the circuit outputs after fanout splitting. Another alternative [3] is to construct a specification that may not be equivalent unless each original circuit input is constrained to be equal to the new inputs split off of it. There are potentially efficient methods for verifying that the BDD for the specification is equivalent to the BDD for the implementation when these constraints are included.

References

[1] C. L. Berman. Circuit width, register allocation, and reduced function graphs. Research Report RC 14129, Mathematical Sciences Dept., IBM T. J. Watson Research Center, 1988.

[2] C. L. Berman. Ordered binary decision diagrams and circuit structure. In *ICCD*, 1989.

[3] C. L. Berman, 1990. Personal Communication.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC*, 1990.

[5] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *ISCAS*, 1985.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[7] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2), 1991.

[8] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability. In *ICCAD*, 1990.

[9] L. Claesen, editor. *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, Nov. 1989.

[10] A. L. Fisher and R. E. Bryant. Performance of COSMOS on the IFIP workshop benchmarks. In Claesen [9].

[11] S. Friedman. *Data Structures for Formal Verification of Circuit Designs.* PhD thesis, Dept. of Computer Science, Princeton Univ., Jan. 1990. Technical Report CS-TR-236-90.

[12] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *ICCAD*, 1988.

[13] M. Fujita, Y. Matsunaga, and H. Fujisawa. On the application of binary decision diagrams to formal hardware design. In Claesen [9].

[14] D. Goldberg. Computer Arithmetic. Appendix A in [15], 1990.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[16] D. E. Long, 1990. Personal Communication.

[17] J. C. Madre. Benchmarks for tautology checking: Experimental results. In Claesen [9].

[18] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD*, 1988.

[19] S. Minato, N. Ishiura, and S. Yajima. Fast tautology checking using shared binary decision diagrams: Benchmark results. In Claesen [9].

[20] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual.* Yale University, 4th edition, 1984.

[21] H. Simonis. Formal verification of multipliers. In Claesen [9].