# Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment

Sharad Malik Albert R. Wang Robert K. Brayton Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

## Abstract

This paper presents the results of a formal logic verification system implemented as part of MIS, the multi-level logic synthesis system developed at U. C. Berkeley. Combinational logic verification involves checking two networks for functional equivalence. Techniques that flatten networks or use cube enumeration and simulation cannot be used with functions that have very large cube covers. Binary Decision Diagrams (BDDs) as presented by Bryant are canonical representations for Boolean functions and offer a technique for formal logic verification. However, the size of BDDs is sensitive to the variable ordering. We consider ordering strategies based on the network topology. Using BDDs, we have been able to carry out formal verification for a larger set of networks than existing verification systems. Also, this method proved significantly faster on the benchmark set of examples tested.

## 1 Introduction

Combinational logic verification involves checking two Boolean networks for functional equivalence. It is used at several levels of the abstraction of a design to verify that the description at that level matches the initial specification.

*Heuristic* techniques use partial simulation and it is possible that a difference in two functions may not be uncovered. *Formal* techniques guarantee functional equivalence. Techniques based on flattening collapse the networks into PLAs and use PLA equivalence for verification (e.g.[6]). Techniques based on cube-enumeration and simulation enumerate a cover for one of the networks and simulate this on the second one (e.g. [10] [8]). Both these techniques do not perform well when the cube cover (sum of products representation) for the network is very large. Techniques based on multi-level cofactoring (e.g. [6]) can be used when the cube cover is large. The cofactoring tree constructed in that process has some similarities with a Binary Decision Diagram.

Bryant [2] presents Binary Decision Diagrams (BDDs) as canonical forms for Boolean functions. Checking for the equivalence of two functions reduces to checking that the canonical forms are identical. Several functions that have large cube covers have compact BDD representations. Conversely, it has been our experience that functions with compact cube covers never have very large BDDs. This extends the range of applicability of BDDs. However, the size of the BDD of a function is sensitive to the ordering of the input variables. Finding the optimum ordering is a co-NP-complete problem ([2]). In [4] an $O(n^2 3^n)$ algorithm is presented for finding the optimum ordering of $n$ variables. The complexity of the verification problem is $O(2^n)$ (verification by simulation of all the minterms). Hence this result is not useful for our purposes. Verification using BDDs has been presented in [2] and [9]. However, the variable ordering has been left to the user.

We have developed strategies for ordering the variables based on the topology of the multi-level network. A verification system using BDDs has been included as part of MIS. Initial results indicate that this technique is capable of handling a larger set of problems than existing systems and is significantly faster on a benchmark set of examples.

## 2 Definitions

The definitions in this section are informal. The terms being defined are in italics.

A *Boolean network* $\eta$ , is a directed acyclic graph (DAG) such that for each node $n_i$ in $\eta$ there is an associated Boolean function $f_i$, and a Boolean variable $y_i$. There is a directed edge from $n_i$ to $n_j$ if $f_j$ explicitly depends on $y_i$ or $\overline{y_i}$. Further, some of the variables in $\eta$ may be classified as *primary inputs* or *primary outputs*. A Boolean network is a representation of a combinational logic circuit. The primary inputs represent the inputs to the circuit and the primary outputs represent the outputs of the circuit.

A node $n_i$ is a *fanin* of a node $n_j$ if the edge $(n_i, n_j)$ is in $\eta$. $n_j$ is a *fanout* of $n_i$.

A node $n_i$ is a *transitive fanin* of a node $n_j$, if there is a path from $n_i$ to $n_j$ in $\eta$.

The *transitive fanin DAG* (TFI DAG), $\eta_n$ of a node $n$ in the network $\eta$, consists of $n$, nodes that are transitive fanins of $n$ and the edges between these nodes.

The *support* of a function $f$ is the set of variables that $f$ explicitly depends on.

The *cofactor* of a function $f$ with respect to a literal $l$, denoted by $f_l$, is the function when $l$ evaluates to 1 (and $\overline{l}$ evaluates to 0).

A *Binary Decision Diagram (BDD)* for a function $f$ is the reduced ordered decision diagram as presented by Bryant [2].

## 3 A Review of BDDs

We review some BDD related terms here. These terms have been defined in [2] and are explained here for ease of reference in subsequent sections.

Consider the decision graph in Fig. 1 for the Boolean function $f = x_1 x_2 + x_3$. Each vertex in the graph corresponds to an input variable. To evaluate $f$ at a given input vector $\mathbf{i} = (i_1, i_2, i_3)$, we traverse the graph from the root to the leaves in the following manner. At a vertex with index $j$ we take the **0** branch if $i_j$ is **0** and the **1** branch if $i_j$ is **1**. This process continues until we reach a terminal vertex (**0** or **1**), which is the value of $f$ for that input vector. For a vertex $u$ corresponding to a variable $x$ in the graph, *u.low* is the vertex reached by going down the **0** branch and *u.high* is the vertex reached by going

[0]This research was supported by NSF grant MIP 8352227, and grants from DARPA, Micro, Intel, AMD, Phillips and SGS

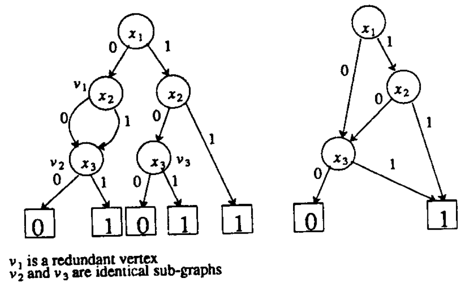
$v_1$ is a redundant vertex
$v_2$ and $v_3$ are identical sub-graphs
Decision Graph
BDD

Figure 1: Decision Graphs and BDDs

down the **1** branch. Let $f$ be the function corresponding to the graph rooted at $u$. By the definition of cofactor we see that the graph rooted at $u.low$ represents the function $f_{\overline{x}}$ and the graph rooted at $u.high$ represents the function $f_x$. The graph may have redundant vertices and duplicate subgraphs as shown. These are eliminated and the graph reduced to a canonical form (a BDD) by the *reduce* operation. The corresponding BDD is also shown in Fig. 1. In a BDD, an arc can only go from a vertex to one with a higher index. Thus, the indices of the variables need to be fixed before the BDD is constructed.

Let $f$ and $g$ be arbitrary Boolean functions and *op* any Boolean operator. The *apply* function constructs the BDD for $f$ *op* $g$ given the BDDs for $f$ and $g$. If $f$ is a function of $g$, *compose* constructs the BDD for $f$ without $g$ as an argument. In network terms, it creates the BDD for the function obtained after collapsing node $g$ into $f$.

## 4 Constructing the BDDs

From the description of a decision graph in the previous section, we see that it can be constructed by using the cofactor operation recursively until we are left with the constant functions **0** or **1**. The BDD may then be constructed by using the *reduce* operation. A function may have a compact BDD but the decision graph constructed by the above procedure may be exponential in the number of inputs. Thus, if the above technique were used there may be an explosion in the memory used before the BDD can be constructed.

Alternatively, in a multi-level network the BDDs for the primary outputs can be constructed as follows. The BDD for any node in the network is constructed by using the BDDs for its fanins and the *apply* or *compose* operation. The fanin BDDs are recursively constructed using the same method. This recursive process stops at the primary inputs, for which the BDDs are trivial. This method does not guarantee avoiding the intermediate memory explosion. However, since *reduce* is used at each stage in the construction process, the chances of an explosion occurring are significantly reduced. We used both *apply* and *compose* for constructing the BDDs.

## 5 Verification

To verify that a primary output $f$ of network $\eta$ is equivalent to primary output $f'$ of network $\eta'$ we construct the BDDs for both $f$ and $f'$ with the same input ordering. Since the BDD is a canonical form for a function (for a particular input ordering), $f$ and $f'$ are equivalent if and only if their BDDs are isomorphic. Since the BDDs are rooted DAGs, the isomorphism operation checks for isomorphism of the root and then recursively checks for isomorphism of the low and high DAGs. This operation takes time linear in the size of the BDDs [2].

Even if the primary output is an incompletely specified function, the equivalence operation can be performed fairly efficiently. Let **f** and **f′** be the incompletely specified functions in $\eta$ and $\eta'$ respectively. Let $f$ and $d$ be the on-set and don't-care-set for **f**, and $f'$ and $d'$ be the corresponding functions for **f′**. **f** and **f′** are equivalent if and only if $(f' \subset (f \cup d)) \cap (f \subset (f' \cup d'))$ is a tautology. Once the BDDs for $f$, $d$, $f'$ and $d'$ have been constructed, the BDD for the above expression can be constructed, using *apply*, in time that is polynomial in the sizes of these BDDs. The above expression is a tautology if and only if its BDD is a single terminal vertex with value **1**.

## 6 Ordering Strategies

The size of a BDD can be very sensitive to the variable ordering. For example, for an $n$ bit adder the BDD sizes may range from $O(n)$ to $O(2^{n/2})$ depending on the ordering chosen. Thus, we need to find a good ordering for the variables in a network. If *apply* alone is used in the BDD construction then only the primary inputs need be ordered. If *compose* is used then the primary inputs as well as the intermediate nodes need to be ordered. The primary inputs should have the same order for both $f$ and $f'$. In our verification process we use one of the networks to select the ordering of the primary inputs and the intermediate nodes are ordered for each network separately. For the rest of this section we will restrict ourselves to finding the ordering of variables for a single primary output node $n$, with function $f$, given its TFI DAG $\eta_n$.

Our ordering strategies are based on an observation and a result. First, we attempt to correlate the functions of intermediate nodes in a multi-level network and the vertices in a BDD. Consider the multi-level network and the BDD shown in Fig. 2. The function $f_2$ is the carry-out from a 2 bit adder. Note that at $n_2$ the only information needed about $a_0$, $b_0$ and $c_0$ is their carry out $f_1$, computed by $n_1$. Thus, $n_1$ encodes this information when it fans into $n_2$. Now, let us look at the corresponding BDD for $f_2$. The vertex $v_0$ corresponds to $f_1$ evaluating to **0**, and $v_1$ corresponds to $f_1$ being **1**. These are the only vertices that fan into subsequent levels in the BDD. Thus, $v_0$ and $v_1$ encode the information about $a_0$, $b_0$ and $c_0$ (the variables seen so far in the BDD) that is needed in the subsequent levels in the BDD. Thus, we observe a similarity in the functions of intermediate nodes in a multi-level network and the vertices in a BDD, viz. encoding information about variables that is needed in subsequent levels.

In general, the vertices at a particular level in a BDD encode the information about the variables seen so far. This is used in the subsequent levels to compute the value of the function. A good ordering will result in fewer vertices in the BDD implying an efficient encoding process. The correlation observed in the previous paragraph suggests that the network toplogy be used to guide the variable ordering.

To take this further, we need to define the *level* of a node in $\eta_n$. The level of a primary output node is 0. For any other node

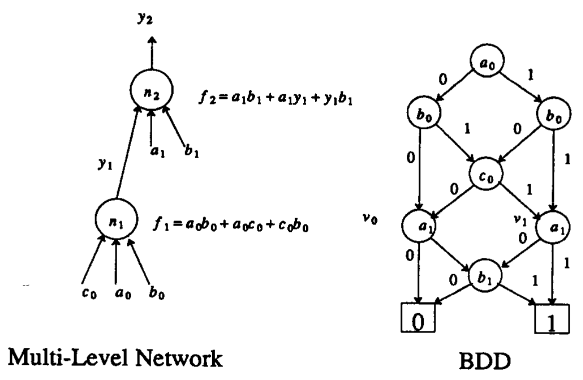

Figure 2: Correlating Intermediate Nodes and BDD Vertices

```
/* level heuristic */
for each node $n$ in $\eta_n$
    compute level($n$);
order_list = nodes sorted in decreasing levels;
```

Figure 3: The *level* Heuristic

the level is given by:

$$level(n_i) = \max_j(level(n_j)) + 1, \ n_j \text{ is a fanout of } n_i$$

Consider the following topology: A set of nodes $S$ does not fan out to any level less than $l$ and a node $n_i$ fans out only to levels less than $l$. Placing $n_i$ in the order after all the nodes in $S$ is a good choice, since we know that there is a small encoding (a subset of intermediate variables) that captures all the interesting combinations of the variables in $S$. This suggests that we order the nodes in decreasing order of levels. This ordering heuristic is termed the *level* heuristic and is outlined in Fig. 3. At the time we were working on this heuristic we became aware of other good results in finding good orderings [5] . This motivated us to look at other ordering strategies. Our second heuristic is based on the following result which we state here without proof.

```
/* fanin heuristic */
order_list = null;
faninOrder(n, order_list);

faninOrder(node, order_list)
{   if(node $\notin$ order_list)
        foreach fanin
            compute TFI DAG depth;
        sorted_fanin_list = fanins sorted in
        decreasing TFI DAG depths;
        foreach fanin in sorted_fanin_list
            faninOrder(fanin, order_list);
        append(order_list, node);
}
```

Figure 4: The *fanin* Heuristic

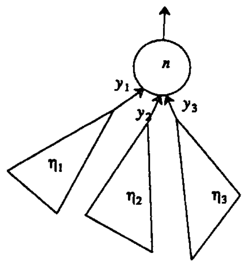

Figure 5: Non-overlapping TFI DAGs

**Lemma 1** *If a function $f$ can be written in the form:*

$$f = g(f_1, g_1(f_2, g_2(\cdots g_{n-1}(f_n, f_{n+1})\cdots)))$$

*where the $g_i$s are any two argument Boolean functions, and each $f_i$ has support that is disjoint from that of the others, then the optimum ordering for $f$ is the concatenation of the optimum orderings for the $f_i$s in the order 1, 2, ... n+1.*

In general it may not be possible to write a node function in the above form and the $f_i$s may not satisfy the disjoint support criterion. However, this could be used as a guiding heuristic. To see its implications in terms of the structure of the BDD, consider a node $n$ that has fanins whose TFI DAGs do not overlap. (See Fig. 5.) The only information needed at $n$ is what each of the fanins, $y_i$, evaluates to. This suggests an ordering process that orders the variables for each TFI DAG and concatenates these orderings. In terms of the BDD structure, the vertices of the BDD need store information only about the current TFI DAG and what the previous fanins evaluate to. However, we found that the order in which the transitive fanin DAGS were visited was important. The idea of the *level* heuristic was extended in visiting the TFI DAGs in order of decreasing depth. The depth of a TFI DAG is the maximum level of any of its nodes. This heuristic, termed the *fanin* heuristic is presented in Fig. 4. This is similar to the heuristic in [5] in that both have a depth-first nature.

Since the ordering heuristics use the topology of the multi-level network, multi-level decomposition may be used on the initial network before the ordering heuristics are used. Note that in this case the decomposition is used only to obtain better orderings and the verification is done on the original networks. For example, in the case of a 4 bit adder, a PLA description of the circuit did not yield any information for the ordering. However, after running the standard multi-level optimization script in MIS both the *level* and *fanin* heuristics generated one of the optimum orders.

## 7 Results

Table 1 shows the results using these heuristics on a large set of examples. The first nine are from the ISCAS benchmarks for testing [1]. The others are large synthesized combinational circuits available at Berkeley. All these circuits have very large cube covers. These results are for the verification of two multi-level circuits for each example. Each of these heuristics was applied first for each output separately and then for all the outputs together (by considering a dummy output to which each of the primary outputs is connected). Except for C6288 (a 16 bit multiplier), each of the other circuits could be verified by using one or more of the ordering strategies. *None of the above circuits could be verified using BDDs with random orderings.*

| Ckt. | level separate | | fanin separate | | level together | | fanin together | |
|---|---|---|---|---|---|---|---|---|
| | Time | Max. BDD | Time | Max. BDD | Time | Max. BDD | Time | Max. BDD |
| C432 | 1232 | 7993 | 1225 | 6881 | 617 | 7993 | 530 | 6881 |
| C499 | 1476 | 6779 | 1102 | 6597 | 599 | 6779 | 394 | 6741 |
| C880 | 374 | 5840 | 205 | 3102 | 270 | 5089 | - | - |
| C1355 | 8986 | 6779 | 4995 | 6597 | 2064 | 6779 | 1198 | 6741 |
| C1908 | 10059 | 5902 | 6016 | 3092 | 1388 | 8022 | 735 | 4037 |
| C2670 | - | - | 4702 | 796 | - | - | - | - |
| C3540 | - | - | 23580 | 68341 | - | - | - | - |
| C5315 | - | - | 7942 | 3248 | - | - | - | - |
| C6288 | - | - | - | - | - | - | - | - |
| C7552 | - | - | 6888 | 1299 | - | - | - | - |
| rot | 414 | 4013 | 400 | 1225 | 240 | 2390 | 238 | 4272 |
| des | 1773 | 340 | 1509 | 86 | 4594 | 412 | 4591 | 591 |

All times are in secs. on a VAX 8650.
A - indicates that the example ran out of memory before completion.
"Max. BDD" is the size of the largest BDD over all the primary outputs.

Table 1: Verification results for the Ordering Heuristics

The advantage of using the same order for all the outputs is that the BDDs for the intermediate nodes need not be recomputed for each primary output. However, the disadvantage is that it may not be possible to find a single good ordering for all the primary outputs. For the very large examples we found that to be the case. For the examples for which we could find a single ordering, the time for this was in general much less than that for the "separate" case. *Des* was an exception to this. In this case the relatively poor quality of the single ordering offset any advantage that was gained by not recomputing the intermediate BDDs. In terms of comparing the two heuristics, for the circuits in which both of them managed to complete, they were comparable, with *fanin* being slightly better. For the larger examples, only *fanin* was successful. It seems that for these circuits the relatively non-overlapping transitive fanin DAGs criterion mentioned in Section 6 holds well.

In comparison with other methods, we found that for circuits that could be flattened to PLAs, using PLA equivalence or cube-enumeration and simulation was much faster than our implementation of BDD verification. However, we feel this is a limitation of our implementation and not of the method. Several other researchers have worked on optimizations with BDDs ([3], [7]) which are not yet included in our implementation. These could speed up BDD verification to make it comparable to the other methods even for these circuits. For circuits with large cube covers BDD verification is significantly better, if not the only option. Only PROTEUS ([10]) and PLOVER ([8]) have presented results on very large examples. Comparisons with PROTEUS are given in Table 2. (Comparisons with PLOVER have been omitted since it is a parallel implementation of one of the PROTEUS algorithms and hence the results are similar.) None of the other circuits in Table 1 was successfully verified by these programs. The column "Max. Cover" indicates the maximum size of a disjoint cover over all the outputs of the circuit. As the size of the cover increases the relative performance of BDD-Verify improves.

| Ckt. | BDD-Verify Time | PROTEUS Time | Max. Cover |
|---|---|---|---|
| des | 1509 | 2580 | 15680 |
| C432 | 530 | 7536 | 569560 |
| C880 | 205 | 10320 | 5508862 |

All times are in secs. on a VAX 8650.

Table 2: Comparison with PROTEUS

## 8 Conclusions

We have presented two variable ordering techniques that are based on the network topology. We have demonstrated that Binary Decision Diagrams with automatic variable ordering are applicable for formal verification of a very large class of circuits. We have been able to successfully verify all but one (the 16-bit multiplier) of the circuit descriptions that we could find at Berkeley. For circuits with very large cube covers this technique appears to be significantly superior to other techniques, if not the only option.

## 9 Acknowledgements

We would like to thank Randal Bryant, Srinivas Devadas, Kevin Karplus, Randy Katz, Patrick McGeer and Richard Rudell for their support and assistance in this project.

## References

[1] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proceedings of the International Symposium on Circuits and Systems*, 1985.

[2] R. E. Bryant. Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[3] R. E. Bryant. *Personal communication*. 1988.

[4] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proceedings of the Design Automation Conference*, 1987.

[5] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, 1988.

[6] G. D. Hachtel and R. M. Jacoby. Verification algorithms for VLSI synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-7(5):616–640, May 1988.

[7] K. Karplus. *Personal communication*. 1988.

[8] H. T. Ma, S. Devadas, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the Design Automation Conference*, 1987.

[9] J. -C. Madre and J. -P. Billon. Proving cicuit correctness using formal comparision between expected and extracted behavior. In *Proceedings of the Design Automation Conference*, 1988.

[10] R. S. Wei and A. Sangiovanni Vincentelli. PROTEUS: A logic verification system for combinational logic circuits. In *Proceedings of the International Testing Conference*, 1986.