VLSI Logic Test, Validation and Verification
Lecture 6, Sep 2, 2002
Boolean Function Representation

Instructor: Priyank Kalla
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT 84112
Email: kalla@ece.utah.edu

There are many different ways of representing Boolean functions, most popular of them can be classified into tabular forms, logic expressions, and binary decision diagrams. The *truth table* is the simplest and most prominent of tabular form representations. It is a complete listing of all points in the Boolean space and of the corresponding values of the outputs. While the truth table is a canonical[1] form representation, its size is exponential in the number of input variables, and hence its use is restricted to functions of small size.

Scalar Boolean functions can also be represented by expressions of literals linked by the + and $\cdot$ operators. Common examples of logic expressions are the *two-level* forms and *multiple-level* forms of logic expressions. Two-level forms are sum-of-products or product-of-sums of literals and they can be used to represent any Boolean function. Are SOP and POS forms canonical? Multiple-level forms involve arbitrary nesting of Boolean operators using parentheses. *Factored forms* representation are one example of multiple-level forms.

Definition .1: A *factored form* is one and only one of the following:

- A literal.
- A sum of factored forms.
- A product of factored forms.

While the above representations are quite compact, the computational complexity of checking for elementary properties of the functions (such as containment, satisfaction, tautology, etc.) from such representation is quite high [1] [2]. One representation that can represent a large class of functions succinctly, while simultaneously allowing efficient manipulation of these functions, is the Binary Decision Diagram [3] which is described below.

I. Binary Decision Diagrams

Binary Decision Diagrams, henceforth called BDDs, were originally proposed in [4] [5] to represent Boolean functions. A BDD represents a set of binary-valued decisions, culminating in an overall decision that can be either TRUE or FALSE. They are a graph-based representation based on trees or rooted Directed Acyclic Graphs (DAGs). There exists a one-to-one mapping between a truth-table and a full-blown BDD. Hence, both are canonical form representations. See the figure below. The nodes in the BDD represent variables, and the edges correspond to assignment of 0 (dotted-edge) or 1 (solid-edge) to the variable. The terminal nodes contain numeric leaf values: 0 or 1, corresponding to the value of the Boolean function. When you traverse any path from the root node to the terminal vertex, it is equivalent to selecting any minterm from the truth table. A BDD with an order imposed on the variables is called an ordered binary decision diagram, or OBDD.

[1] A representation is canonical if there exists one and only one, unique, representation for a function and all its equivalents.

Fig. 1. One to one mapping between truth table and BDDs

In [3], it was shown that by imposing a total order on the variables of the BDD and subsequently removing any redundancies from them, they get reduced in size and yet retain their **canonical form** representation. First, let us review basic definitions related to OBDDs, and analyze some interesting features that are the key to understanding BDDs, and the relationship between the structure of the BDD and the characteristics of the Boolean functions that it represents.

Definition I.1: An **OBDD** is a rooted directed graph with vertex set V. Each non-leaf vertex has as attributes a pointer index(v) $\in \{1, 2, \ldots, n\}$ to an input variable in the set $\{x_1, x_2, \ldots, x_n\}$, and two children low(v), high(v) $\in V$. A leaf vertex v has as an attribute a value, value(v) $\in$ **B**.

For any non-leaf vertex pair {v, low(v)} and {v, high(v)}, index(v) < index(low(v)) and index(v) < index(high(v)), respectively.

Definition I.2: An OBDD with root v denotes a function f^v such that:

- If v is a leaf with value(v) = 1, $f^v = 1$.
- If v is a leaf with value(v) = 0, $f^v = 0$.
- If v is a non-leaf node with index(v) = i, $f^v = x_i' \cdot f^{low(v)} + x_i \cdot f^{high(v)}$.

From the above definition it can be inferred that the decomposition principle associated with OBDDs corresponds to recursive application of the Shannon's expansion.

Two OBDDs are *isomorphic* if there is a one-to-one mapping between the vertex sets that preserve adjacency, indices and leaf values. Thus, two isomorphic OBDDs represent the same function. While an OBDD uniquely identifies a Boolean function, the converse is not true. In order to make an OBDD canonical, any redundancy in the OBDD must be eliminated.

Definition I.3: An OBDD is said to be **reduced** if it contains no vertex v with $low(v) = high(v)$, nor any vertex pair $\{u, v\}$ such that subgraphs rooted at u and v are isomorphic. A reduced OBDD is referred to as an **ROBDD**.

In [3], an algorithm to reduce the OBDDs was presented, and it was proved that ROBDDs are a canonical representation. The algorithm traverses the OBDD bottom-up, identifies and removes all the nodes that are redundant (if $low(v) = high(v)$) and merges all isomorphic subgraphs (if $low(u) = low(v)$ and $high(u) = high(v)$) present in the OBDD. The algorithm terminates when the root is reached. Fig. 2 depicts the reduce operation performed on the OBDDs. The isomorphic subgraphs are merged in Fig. 2(b) and the resulting redundant node is deleted in to obtain the ROBDD shown in Fig. 2(c).

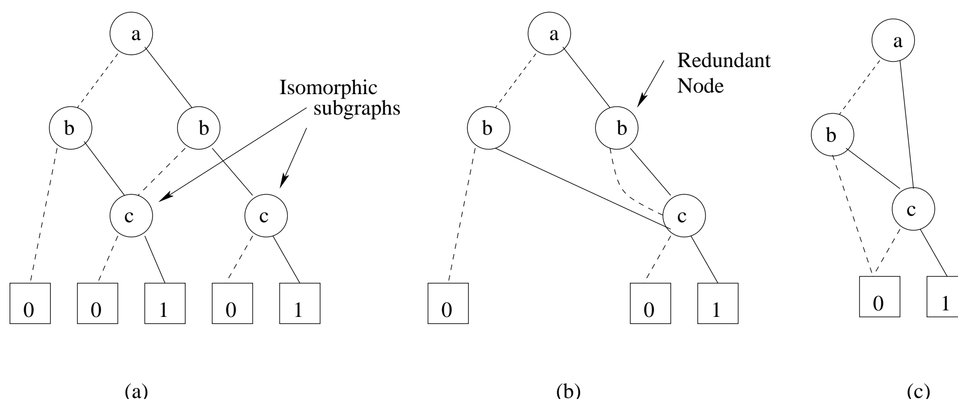

Fig. 2. Reduce operation on an OBDD. Dotted edges in the OBDD correspond to the assignment of 0 to the variable corresponding to the vertex. Solid edges represent an assignment of the value 1 to the variable.

A. Efficient Construction and Manipulation of ROBDDs

OBDDs of large designs can be prohibitive in size, and the reduction operation upon them can become infeasible. It is important to construct ROBDDs for the Boolean functions directly, thus avoiding the reduction step. This is made possible by using a hash table called the *unique table*, which contains a *key* for each vertex of the ROBDD. The key is a triple $\langle v, low(v), high(v) \rangle$. The function associated with each vertex is uniquely identified by the key. During

the OBDD construction, when a new vertex is to be created, a look-up in the unique table is performed to determine if another vertex in the table implements the same function. If such a vertex is found, its key is returned, otherwise a new vertex is added into the table. This disallows any duplication of equivalent vertices in the OBDD.

Let us follow the "direct" construction of ROBDDs for two equivalent functions $f_1 = ac + bc$ and $f_2 = a(b + c)$. We are going to cover this example in class. Please follow this example carefully to observe how the hash table disallows creation of duplicated vertices - and this is how it *guarantees* a canonical representation.

In [6], one such implementation of the unique table was presented and it was shown to be a *strong canonical form*. Strong canonical form is a form of pre-conditioning which reduces the complexity of an equivalence test between elements in a set. A unique identifier is assigned to each unique element in the set so that an equivalence test is a simple scalar test between the identifiers of each element.

B. How to operate upon two BDDs?

The ROBDD construction principle implemented in the unique table is defined according to the ITE (IF-THEN-ELSE) operator. ITE is a Boolean function defined for three inputs f, g, h as follows:

$$ite(f, g, h) = f \cdot g + f' \cdot h \tag{1}$$

ITE is the logical function performed at each node in the ROBDD and its recursive application can be used to perform any Boolean operation. Let $Z = ite(f, g, h)$ and let v be the top variable corresponding to the ROBDDs of f, g and h. Applying the Shannon's expansion on Z with respect to variable v, and using the ITE operator, we get:

$$\begin{aligned}
Z &= vZ_v + v'Z_{v'} && (2)\\
&= v(ite(f, g, h))_v + v'(ite(f, g, h))_{v'} && (3)\\
&= v(fg + f'h)_v + v'(fg + f'h)_{v'} && (4)\\
&= v(f_v g_v + f'_v h_v) + v'(f_{v'} g_{v'} + f'_{v'} h_{v'}) && (5)\\
&= ite(v, ite(f_v, g_v, h_v), ite(f_{v'}, g_{v'}, h_{v'})) && (6)\\
&= (v, ite(f_v, g_v, h_v), ite(f_{v'}, g_{v'}, h_{v'})) && (7)
\end{aligned}$$

The terminal cases for the above recursion are: $ite(1, f, g) = ite(0, g, f) = ite(f, 1, 0) = f$. Based on the above principle, any Boolean operation on functions can be performed using the *ITE* operator recursively. For example, it can be used to check implication between two functions. If $f \to g$, then $f' + g$ is tautology. Tautology checking can be performed by checking whether or not $ite(f, g, 1) = \text{TRUE}$. Similarly, $f \cdot g$ can be computed as $ite(f, g, 0)$. f' can be computed as $ite(f, 0, 1)$, and so on.

A few points before we proceed further: What the above equations mean, quite simply is that if v is the variable associated with the topmost nodes of f, g, h, then the resulting function $Z = ite(f, g, h)$ also has v as the top variable associated with its root. Furthermore, notice the "recursive" nature of the ITE operator. Following from the first to the last equation, it is clear that we began to apply the ITE operator on the top nodes (corresponding to variable v) of f, g, h, and ended-up applying the ITE operator to their cofactors: f_v, g_v, h_v and f'_v, g'_v, h'_v. Now, a question for you: Where do the co-factors of f, g, h reside in the BDD? In their children nodes, of course. This implies that the ITE operator applied to the top nodes results in its application to the children nodes. This would ultimately lead to the terminal nodes. Hence the recursion!

C. *Complexity of BDDs*

The complexity of the $ite(f, g, h)$ operation is polynomial in the number of nodes. To perform any Boolean operation on OBDDs, the ITE is invoked at most once for each combination of nodes in f, g, h, or $O(|f||g||h|)$ times. This means that in the worst case, the new ROBDD created is as large as the product of the sizes of the original ROBDDs. This complexity in the worst-case is quite high - it is polynomial (cubic) in the number of nodes of f, g, h. But you should realize, in the worst case, the size of each f, g, h can be *exponential* w.r.t. the number of variable in their support set. In other words, complexity of ROBDDs is exponential in the worst-case. Judged by their worst case performance, ROBDDs may not appear to be very useful, but in many commonly encountered functions their performance does not exhibit worst-case behaviour. This has enabled their application to a large number of problems encountered in the area of VLSI design, test and verification - check for equivalence, satisfiability and tautology, etc. In the next chapter, we will analyze the power and limitations of ROBDDs when applied in the context of many different problems in CAD.

References

[1] R. Boppana and M. Sipser, "The Complexity of Finit Functions", in J. van Leeuwen, editor, *Algorithms and Complexity*, vol. A of *Handbook of Theoretical Computer Science*. MIT Press, 1990.

[2] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[3] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. C-35, pp. 677–691, August 1986.

[4] C. Lee, "Representation of Switching Circuits by Binary Decision Diagrams", *Bell Systems Technical Journal*, vol. 38, pp. 985–999, July 1959.

[5] S. Akers, "Binary Decision Diagrams", *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

[6] K. S. Brace, R. Rudell, and R. E. Bryant, "Efficient Implementation of the BDD Package", *in DAC*, pp. 40–45, 1990.