DATA STRUCTURES FOR FORMAL VERIFICATION OF CIRCUIT DESIGNS

Steven Friedman

CS-TR-236-89

January 1990

Data Structures for Formal Verification of Circuit Designs

Steven Friedman

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JANUARY 1990

Contents

Abstract 1

Chapter 1—Introduction and Summary of Results 2

Chapter 2—The Binary Decision Diagram

2.1 Properties 5

2.2 Operations on BDDs 8

Chapter 3—Minimizing Ordered Binary Decision Diagrams

3.1 Overview 10

3.2 Preliminaries 12

3.3 The algorithm 14

3.4 Complexity analyses 18

3.5 Remarks 21

Chapter 4—Combinational Logic Verification

4.1. The Projective Binary Decision Diagram (pBDD) 22

4.2. Operations on pBDDs 24

4.3. The Multiplier Template 30

4.4. Combinational Logic Verification in Practice 31

Chapter 5—Sequential Logic Verification

5.1. Overview 34

5.2. The problem 35

5.3. The n-ary Deterministic Finite Automaton (n-DFA) 36

5.4. Our solution 37

5.5. Some implementation notes 41

Chapter 6—Conclusions and Future Work 43

References 44

Diagrams 47

ABSTRACT

We address the problem of logic verification, for both combinational and sequential logic. Our focus is on increasing the size and range of circuits for which formal verification, as opposed to test vector generation and simulation, is feasible. We discuss the Binary Decision Diagram (or BDD) as a means of representing boolean functions. This representation is canonical with respect to the variable ordering and is usually compact. We present an original algorithm for finding the optimal variable ordering for a BDD. This algorithm is additionally useful in the synthesis of a certain type of circuit. We introduce a data structure called the projective BDD which is not canonical but can be more compact than the BDD. We develop an algorithm for verifying multiplier circuits using this structure. Finally, we present a form of deterministic finite automaton, the n-DFA, whose transitions are labeled with boolean functions. Using this structure, we develop an algorithm for the problem of sequential logic verification. For each of these algorithms—to do BDD minimization, multiplier verification, and verification of sequential circuits—we report the running times of actual implementations and compare them to benchmarks in order to demonstrate the practicality of our methods.

Chapter 1: Introduction and Summary of Results

Gaining confidence in the correctness of circuit designs is taking a large and increasing portion of the total design time. Current practice is to simulate circuits with a very large amount of test data. Determination of an appropriate set of tests to exercise a circuit is often a formidable task, hence only partial correctness of the design is typically established.

The scope of this dissertation is limited to formal design verification, which is essentially the verification of designs without resorting to exhaustive simulation. Currently, it is too time-consuming to apply formal verification to any but the smallest components of circuits. Our goal here is to apply formal verification to more circuits than are currently practical.

In the special case of purely combinational logic, much work has been done on formally proving the equivalence of two logic networks. For example, in [SB] the authors use a "Differential Boolean Analyzer", in which they heuristically choose a variable v, and verify the circuit recursively setting $v = 0$ and setting $v = 1$. They are able to accelerate this heuristic by detecting cases that they have already encountered. In [KN], the authors use an XOR-of-ANDs canonical form for boolean expressions, and test for satisfiability (or validity) using term-rewriting. Despite the NP-hardness of even this special case, some of these approaches have achieved success for a large class of interesting circuits and (especially [SB]) have become widely used.

As for the problem of verifying sequential designs, several techniques have been proposed in recent years, but none has yet gained wide acceptance in practice. Some ([Da], [PS]) of these apply techniques developed in software verification, and require the designer to provide assertions as well as the design. Another [Wo] utilizes a resolution based theorem prover, to which the designer must provide the axioms. Bryant [B1] symbolically simulates a switch-level design, producing a sequence of boolean expressions representing the sequence of circuit outputs. Proving the equivalence of

two designs is addressed in [BC], which requires the designer to specify his design in temporal logic. Also see [Ba], [Ge], [Mc].

In [Ak] and [B2], the authors discuss a data structure called the Binary Decision Diagram, or BDD. The BDD is a means of representing boolean functions, and certain of its properties make it particularly applicable to the problem of design verification. Operations upon BDDs are quick in the sizes of the BDDs, and BDD representations are compact for common functions such as addition (any bit or carry) and tally (and derived functions such as parity or majority). Furthermore, testing a BDD for satisfiability can be done in constant time. In Chapter 2, we discuss the BDD and its properties. Briefly, a BDD consists of a compacted tree of decision nodes which is traversed from the root to the leaves in order to apply the function to a particular bit vector.

With varying degrees of success, we have extended the usefulness of the BDD in three ways, which we present in the remaining chapters. First, for quick design verification we require that operations on BDDs be quick, which in turn requires that the BDDs be compact. The compactness of an ordered BDD is heavily dependent on one of its parameters, the variable ordering. In Chapter 3, we present an original algorithm for minimizing the number of nodes in ordered BDDs. The time-complexity of this algorithm is an exponential improvement over that of previous efforts (though still exponential, of course). We were able to minimize BDDs in sixteen variables using this algorithm, but only ten variables using a benchmark (from [NB]). The actual running times may be found in Section 3.1.

One obstacle to the use of ordered BDDs in formal verification is that certain commonly used functions, such as multiplication, have an ordered BDD representation that is exponential in size regardless of the variable ordering. We address this problem in Chapter 4. We introduce the projective BDD, in which we order the decision nodes according to our expectation of the final form of the data structure. This can lead to a more compact representation than with the ordered BDD. We develop a basic algorithm for combinational logic verification using this data struc-

ture, and discuss several possible refinements to this algorithm. We construct an $O(n^3)$ representation for a multiplier.

We verified the fourth through ninth least significant bits of a multiplier circuit using this representation and the basic algorithm, and compared our implementation's performance to that of our implementation of the Differential Boolean Analyzer. The results are listed in Section 4.4. Even without the refinements, our algorithm's running time grows more slowly than for the benchmark; furthermore, the running times are comparable. We believe that the refinements could lead to further speedup.

Finally, in Chapter 5, we address the problem of sequential logic verification. We present the n-DFA, a more efficient way of representing certain deterministic finite automata, which we introduced in [SF]. In this chapter we also introduce the Batman, which is a more general form of BDD with several roots and several terminals. This structure saves both time and space when an operation is performed on many BDDs at once. We use the Batman to advantage in representing the transition function of an n-DFA. Finally, we outline a method for verifying sequential circuit designs using these data structures.

Our approach to sequential logic verification is to compare the circuit for functional equivalence to another circuit which is known to work correctly. The two circuits may differ in certain attributes such as the number of clock cycles required to complete a computation. Except for one ([DM]), all the papers we found on the topic approached the problem by comparing the circuit in question to a formal specification of the circuit's functionality (in some language such as temporal logic). Bryant (in [B1]) notes that arithmetic circuits lend themselves nicely to description in these languages, but many other circuits, such as RAMs, do not. Thus, our approach applies to situations where language-based approaches do not. Nonetheless, our running times compare favorably to those reported elsewhere. Experimental results appear in Section 5.5.

Chapter 2: The Binary Decision Diagram

2.1 Properties

We begin by describing binary decision trees, binary decision diagrams, and ordered binary decision diagrams.

A binary decision tree representing the boolean function $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$ is depicted in Fig. 1. To evaluate the function at a vector $\bar{b} = (b_1b_2b_3b_4)$, we begin at the root and descend through the tree until hitting a leaf (or *terminal*). In particular, when at a node labeled i, we go to the left son (or *0-link*) if $b_i = 0$ and the right son (or *1-link*) otherwise. This process ends at a terminal labeled with the value $f(\bar{b})$. [*NOTE*: Sometimes we use $\{0, 1\}$, and sometimes $\{FALSE, TRUE\}$, as the possible values of a boolean function. We will consider the two notations to be interchangeable.]

Unfortunately, this representation for boolean functions is no more concise than a complete truth table, since a tree representing a function of n variables has $2^{n+1}-1$ nodes ($2^n - 1$ decision nodes and 2^n terminals). However we can collapse the tree in such a way that the function can still be evaluated in the same manner, yet the resulting acyclic digraph (called a binary decision *diagram*) may be much smaller (see Fig. 2(a)). Binary decision diagrams were introduced in [Le] and further popularized by [Ak]; much work on binary decision diagrams and their applications (including logic synthesis, verification and test generation) has been reported (e.g. [Mo], [AR]).

Note that in the particular decision tree of Fig. 1, regardless of the path that we take (i.e. regardless of the value of $\bar{b}$), we will be evaluating the bits in the same order (namely, b_3, b_4, b_1, b_2). Therefore, at each level of the tree, all nodes have the same label For example, at level 1 (near the terminals), all nodes are labeled 2. Call such a tree an *ordered decision tree.* Then we define (following [B2]) an *ordered binary decision diagram*, or ordered BDD, as the binary decision

diagram resulting from starting with an ordered decision tree and applying the two collapsing operations described below, until they no longer apply. In this chapter and the next, any mention of a BDD may be understood to refer to an ordered BDD, unless otherwise specified.

In order to describe these collapsing operations, we make use of the following definition. Two nodes of a decision diagram are ***equivalent*** if they are either

(1) both terminals with the same value (TRUE or FALSE), or

(2) both internal nodes having the same label and their left sons are equivalent and their right sons are equivalent.

There are two operations for collapsing (called "reducing" in [B2]):

(i) If the two sons of a node a are equivalent then delete node a and direct all of its incoming edges to its left son.

(ii) If nodes a and b are equivalent then delete node b and direct all of its incoming edges to a.

Thus, the first operation avoids the testing of variables on which the function does not depend; the second gets rid of nodes representing functions already represented by another node in the diagram.

The diagram may be collapsed even further through the use of ***Typed Shannon's canonical form***, described in [MB], in which the two terminals, FALSE and TRUE, are also considered equivalent for collapsing purposes. The effect is that each node of the collapsed BDD corresponds to two boolean functions which are complements of each other. Whether the "positive" function or the "negative" function is desired is indicated by marking each incoming edge. In this context, a function is ***positive*** iff setting all the variables to TRUE makes the function evaluate to TRUE. Fig. 3 shows an ordered BDD along with its corresponding Typed Shannon's canonical form. This extra collapsing step can reduce the size of the BDD by as much as

half. It can be thought of as being at the "bottom level" of any BDD algorithms we present; for simplicity we will use the standard BDD model in our discussion.

As observed in [B2], the diagram that results from collapsing an ordered decision tree does *not* depend on which of the nodes (there may be many possibilities) is eliminated at each step of the collapsing process (thus it is well-defined). In other words, the ordered binary decision diagram is a canonical representation for a given boolean function, given an ordering on its variables. However, it is computationally most efficient to collapse the nodes of a BDD beginning at the terminals, and proceeding toward the root. This is because of the recursive definition of "equivalent": if we collapse from the terminals to the root, we need only check whether the sons are equal, because the equivalent nodes at that level have been collapsed together at a previous step.

Henceforth, all binary decision diagrams referred to in Chapters 2 and 3 will be ordered (unless otherwise specified), and we call them simply *BDD*s. Thus we define BDD(f, π) as the BDD representing function f, given ordering π on its variables. As examples, BDD($x_1x_2 + x_3x_4, \langle 2,1,4,3\rangle$) and BDD($x_1x_2 + x_3x_4, \langle 2,4,1,3\rangle$) are illustrated in Fig. 2(a) and 2(b), respectively. Note that we list the variables by level from the terminals to the root, which is the direction opposite that involved in traversing the BDD to apply its function to a bit vector. †

Bryant [B2] discusses the advantages of constraining binary decision diagrams to the ordered variety, arguing essentially as follows. This representation facilitates many of the most useful operations on functions (such as AND, OR, NOT, testing for equivalence, and testing for satisfiability), and is canonical. The disadvantage of this constraint is that there exist functions whose smallest — measured by number of nodes — ordered diagram is strictly larger than its smallest (unordered) diagram. However, the difference is usually slight for most functions arising in circuit design;

---

† We denote the ith value of an ordering (that is, permutation) π by $\pi[i]$; thus if $\pi = \langle 2,1,4,3\rangle$ then $\pi[1] = 2$, $\pi[2] = 1$ and so forth.

indeed, most useful circuits seem to have small (say, of size polynomial in n) BDDs. For example, the BDD representing an output bit of an addition function has size linear in the number of input bits.

BDD algorithms are impractical for those functions for which any collapsed decision tree is exponential in size, such as multiplication (see [B2]). However, we may still be able to achieve compression by allowing decisions to be repeated along some paths. This idea is somewhat counterintuitive, but it works because the tree does not need to "remember" as much intermediate computation. We handle this case in Chapter 4.

Also, we will find it efficient to collapse a large number of BDDs into one data structure with many roots and many terminals, using the same collapsing operations as for a single BDD. We call this structure a *Batman* because a schematic outline of it resembles the logo on the Batsignal (see Figure 13). This structure saves both time and space when an operation is performed on many BDDs at once. In addition, it provides a compact way of storing the transition function for an n-DFA, as will be discussed in Chapter 5.

2.2 Operations on BDDs

Although we defined a BDD as the result of the collapsing process started on an entire decision tree, it can usually be computed more quickly (given a particular ordering) by starting with a concise boolean expression for the function and building up the diagram in pieces, as follows (following [B2]):

The BDD representation for a single variable v is a single node labelled v with a 0-link to FALSE and a 1-link to TRUE.

The BDD representation for $\neg B$, where B is a boolean expression, is obtained by recursively computing the BDD representation for B, and then swapping the two terminals TRUE and FALSE.

The BDD representation for $B_1 \odot B_2$, where $\odot$ is any binary operation, is obtained by recursively computing the BDD representations for B_1 and B_2, and

then applying the following algorithm to the roots of the two resulting BDDs:

(1) Call the two nodes n_1 and n_2. If they are both terminals, then return the terminal labelled $n_1 \odot n_2$.

(2) If n_1 and n_2 are both labelled with the same variable v, then return a node labelled v with the i-link (for $i = 0, 1$) pointing to the node that results from recursively applying this algorithm to the i-link of n_1 and the i-link of n_2.

(3) Otherwise, assume without loss of generality that the label of n_2 precedes the label of n_1 in the variable ordering (that is, n_2 is nearer to the terminals, and its variable evaluated later, than n_1). Then, return a node labelled with n_1's variable, and with the i-link (for $i = 0, 1$) pointing to the node that results from recursively applying this algorithm to the i-link of n_1, and the node n_2.

Note that in steps (2) and (3), we save computation time by maintaining a table of (n_1, n_2) pairs already encountered, along with the BDD node that resulted. With this table, the algorithm runs in time proportional to the product of the sizes of the two BDDs.

Finally, apply the two collapsing operations given above, starting with the terminals, and proceeding upward to the root. This is only done after the operator has been applied to the entire BDDs, and not at each level of recursion.

This algorithm extends naturally to operations on more than two BDDs, but the time complexity is proportional to the product of the sizes of all the BDDs.

Since the BDD representation is canonical with regard to a given ordering, testing the equivalence of two BDDs can be done, using (for example) depth-first search, in time proportional to the number of nodes they contain. In particular, testing satisfiability of a BDD merely involves checking whether its root is the terminal FALSE.

Chapter 3: Minimizing Ordered Binary Decision Diagrams

3.1 Overview

Although ordered BDDs can be quite compact, the size of the BDD for a given function is extremely sensitive to the choice of an ordering on the variables; for example, a circuit representing the carry-out of an adder has size $O(n)$ under some orderings and $O(2^{n/2})$ under others. This sensitivity can be understood by thinking of the BDD as a state machine performing a computation, in which the only information at each step is the identity of the current state. If, say, ten bits are input before any useful computation can be done with them, then the machine needs 1024 states to store the ten bits (compare Figures 2(a) and 2(b)).

Thus we are left with the question of how to find the best variable ordering (i.e. leading to the smallest BDD) for a given function. Bryant [B2] leaves this task to the human user; however for many applications such intervention may not be practical.

Certain heuristics for finding a good, but not necessarily optimal, ordering are presented in [NB]. They present a class of heuristics of time complexity $O(n2^n)$, $O(n^2 2^n)$, $O(n^3 2^n)$, and so forth, with progressively better approximations to the optimum. Looked at another way, the first heuristic gives good results for a small class of functions, the next gives good results for a somewhat larger class, etc.

In this chapter we present an original algorithm (first reported in [FS]) for finding an optimal ordering, which runs in $O(n^2 3^n)$ time, an exponential improvement over the (essentially brute-force) $O(n!\,2^n)$ optimizing algorithm reported in [NB]. Although the function $n^2 3^n$ grows very quickly with n, it does so dramatically more slowly than does $n!\,2^n$. For example, here are some running times for C implementations on a DEC 8650:

n	from [FS]	from [NB]
8	< 1 sec	21 sec
10	6 sec	2.1 hours
12	1 min	[44 days]
14	18 min	[91 years]
16	3.5 hours	[48 millenia]

(The bracketed numbers are extrapolations based on the algorithm's time-complexity, and should not be taken too seriously.)

Thus for a certain range of n (say 8 through 10) our algorithm is much faster than the brute-force, and for the next few values (say 12 through 16) our algorithm is feasible whereas the brute-force is not.

One application for our algorithm relates to the synthesis of Differential Cascode Voltage Switch (DCVS) trees (see [HG], [YH], [NB]). These are essentially hardware embodiments of ordered BDDs, and the task of minimizing the number of transistors in their implementation turns out to be essentially one of finding the best ordering for a BDD. Our algorithm is particularly well-suited for this application, for two reasons:

(1) Since BDDs are directly realized as hardware, minimizing the number of nodes of a BDD corresponds roughly to minimizing the layout area of a DCVS tree.

(2) Functions of about 11 or fewer variables are typical in this application; hence our algorithm would enable one to actually find the optimum rather than resorting to heuristics.

Another application for finding exactly optimal BDDs arises in designing heuristics for finding good BDDs. In particular, we might wish to evaluate these heuristics by comparing their performance (relative to the optimum) on various test data. Therefore, a more efficient optimizing algorithm permits larger test cases.

Other applications utilizing BDDs are reported in [B1], and we make extensive use of BDDs here.

3.2 Preliminaries

In the following definitions and in the Lemma, f denotes a boolean function over variables $x_1, x_2, \ldots, x_n$.

(1) If $b \in \{0,1\}$ and $1 \leq i \leq n$, then $\mathrm{f}\,|_{x_i=b}$ denotes the boolean function of n variables such that for all $x_1, x_2, \ldots, x_n$,

$$\mathrm{f}\,|_{x_i=b}(x_1, x_2, \ldots, x_n) = \mathrm{f}(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

Extend this concept of the *restriction* of f as follows: if $b_1, b_2, \ldots, b_r \in \{0,1\}$ and $i_1, i_2, \ldots, i_r$ are distinct members of $\{1, 2, \ldots, n\}$ then inductively define

$$\mathrm{f}\,|_{x_{i_1}=b_1,\ \ldots,\ x_{i_r}=b_r} = (\mathrm{f}\,|_{x_{i_1}=b_1,\ \ldots,\ x_{i_{r-1}}=b_{r-1}})|_{x_{i_r}=b_r} \quad .$$

In other words, an expression for $\mathrm{f}\,|_{x_{i_1}=b_1,\ \ldots,\ x_{i_r}=b_r}$ can be derived from an expression for f by replacing each occurrence of x_{i_j} by the constant b_j for each $1 \leq j \leq r$.

(2) If $I \subseteq \{1, 2, \ldots, n\}$ then define $\Pi(I)$ as the set of orderings on $\{1, 2, \ldots, n\}$ whose first $|I|$ members constitute I, that is

$$\Pi(I) = \{\pi : \pi \text{ is an ordering on } \{1, 2, \ldots, n\} \quad \text{and } \{\pi[1], \pi[2], \ldots, \pi[|I|]\} = I\}$$

(3) If $v \in \{1, 2, \ldots, n\}$ and π is an ordering on $\{1, 2, \ldots, n\}$ then $\mathrm{cost}_v(\mathrm{f}, \pi)$ denotes the number of nodes labelled v in the diagram $\mathrm{BDD}(\mathrm{f}, \pi)$. Thus, our problem is to find a π that minimizes

$$\sum_{v=1}^{n} \mathrm{cost}_v(\mathrm{f}, \pi) \ .$$

Our algorithm depends heavily on the following result:

Lemma: Let $I \subseteq \{1, 2, \ldots, n\}$, $k = |I|$, and $v \in I$. Then there is a constant c such that for each $\pi \in \Pi(I)$ satisfying $\pi[k] = v$ we have

$$\mathrm{cost}_v(\mathrm{f}, \pi) = c\,.$$

Proof: Let $\pi \in \Pi(I)$ be such that $\pi[k] = v$. Let $J = \{i_1, i_2, \ldots, i_{n-k}\} = \{1, 2, \ldots, n\} - I$. Then for each $\bar{b} = \langle b_1, b_2, \ldots, b_{n-k} \rangle \in \{0,1\}^{n-k}$ a node representing the restricted function $\mathrm{f}_{\bar{b}} = \mathrm{f}\,|_{x_{i_1}=b_1, \ \ldots, \ x_{i_{n-k}}=b_{n-k}}$ must appear in BDD(f, π). Note that the set $S = \{\mathrm{f}_{\bar{b}} \ : \ \bar{b} \in \{0,1\}^{n-k}\}$ remains constant over all $\pi \in \Pi(I)$, since J depends only on I. Furthermore, there is exactly one node of BDD(f, π) corresponding to each member of S, because of collapsing operation (ii) (see section 2.1). Now, the node corresponding to a given $\mathrm{f}_{\bar{b}}$ is the root of BDD($\mathrm{f}_{\bar{b}}$, π'), where $\pi' = \langle \pi[1], \pi[2], \ldots, \pi[k] \rangle$. Clearly, if a node labelled v appears in this diagram, it must be at the root. In particular, the only nodes labelled v correspond to those functions in S that depend on x_v, because of collapsing operation (i). Thus, for any $\pi \in \Pi(I)$, the number of nodes labelled v is equal to the number of functions in S that depend on x_v, which is determined only by I and v. This number then is the constant c required by the Lemma. ■

This lemma will allow us to avoid testing most of the $n!$ permutations. For example, if we know that

$$\pi_{\{1,2,3\}} = \langle 2, 3, 1 \rangle$$

$$\pi_{\{1,2,4\}} = \langle 4, 1, 2 \rangle$$

$$\pi_{\{1,3,4\}} = \langle 1, 4, 3 \rangle$$

$$\pi_{\{2,3,4\}} = \langle 4, 3, 2 \rangle$$

(π_I, defined formally below, is an ordering of the variables on the $|I|$ levels nearest the terminals), then $\pi_{\{1,2,3,4\}}$ must be either $\langle 2,3,1,4 \rangle$, $\langle 4,1,2,3 \rangle$, $\langle 1,4,3,2 \rangle$, or $\langle 4,3,2,1 \rangle$. That is, the knowledge of the optimal ordering for the various sets of three indices allows us to restrict our search to only four (rather than $4! = 24$) candidates for the optimal ordering of this set of four indices. Furthermore, the value of (say) $\pi_{\{1,2,3\}}$ contributes to the computation of $\pi_{\{1,2,3,i\}}$ for any i. Thus we use this lemma to develop a dynamic programming algorithm, below.

3.3 The algorithm

Our algorithm is shown in Fig. 4. We process each subset of the variables' indices $I \subseteq \{1, 2, \ldots, n\}$ in ascending order of their cardinalities $k = |I|$. In particular, we compute the following three values for each I:

(1) MinCost_I, which is the minimum of $\sum_{v \in I} \text{cost}_v(\text{f}, \pi)$ over all $\pi \in \Pi(I)$. Initially, we set $\text{MinCost}_\emptyset = 0$.

(2) π_I, which is a member of $\Pi(I)$ achieving the minimum described above. The key fact (a consequence of the Lemma) on which the algorithm is built is that the cost of the variables on the first k levels depends only on their ordering, i.e. it does not depend on the ordering of the remaining $n - k$ variables. In particular, each ordering π such that $\pi[i] = \pi_I[i]$ for all $1 \leq i \leq k$ satisfies

$$\sum_{v \in I} \text{cost}_v(\text{f}, \pi) = \text{MinCost}_I \quad .$$

Initially, $\pi_\emptyset = \langle \rangle$, the empty sequence. Thus, our problem is to find $\pi_{\{1,\ldots,n\}}$, which yields a BDD having $\text{MinCost}_{\{1,\ldots,n\}}$ internal nodes.

(3) $TABLE_I$, which is the truth table for a mapping from $\{0,1\}^{n-k}$ to those nodes of $\text{BDD}(\text{f}, \pi)$ that either are terminals (the nodes TRUE or FALSE) or are internal nodes labelled with members of I. Note that there are MinCost_I such internal nodes, and hence precisely $\text{MinCost}_I + 2$ distinct values in the table; we identify the internal nodes with integers between 1 and MinCost_I. The interpretation of the mapping is as follows: each element $\bar{b} = (b_1, b_2, \ldots, b_{n-k})$ of the domain represents a truth assignment to the variables with indices $i_1, i_2, \ldots i_{n-k} \notin I$, and it is mapped to the node of $\text{BDD}(\text{f}, \pi_I)$ that corresponds to the function $\text{f}\,|_{x_{i_1}=b_1,\ \ldots,\ x_{i_{n-k}}=b_{n-k}}$. Thus initially, $TABLE_\emptyset$ is the full truth table for f, mapping $\{0,1\}^n$ to $\{\text{TRUE}, \text{FALSE}\}$.

To compute MinCost_I, π_I and $TABLE_I$, we look at each $v \in I$ and compute $\text{cost}_v(\text{f}, \pi)$ for some $\pi \in \Pi(I)$ such that $\pi[k] = v$, i.e. some ordering with v at

position k and the other members of I at the lower positions. As a consequence of the Lemma, it does not matter which such π we use; $\text{cost}_v(\text{f}, \pi)$ will be the same. Therefore for convenience, we use the ordering $\langle v, \pi_{I-\{v\}}\rangle$ as this π, since we have already computed $TABLE_{I-\{v\}}$.

In particular, to compute $\text{cost}_v(\text{f}, \langle v, \pi_{I-\{v\}}\rangle)$ we do a folding operation on the truth table $TABLE_{I-\{v\}}$. We call it "folding" because it involves comparing the corresponding elements of two halves of the truth table to arrive at another truth table (stored as $TempTable$) with half the number of lines. In particular, if $i_1, i_2, \ldots, i_{n-k}$ are the elements of $\{1, 2, \ldots, n\} - I$, then for each length $n-k$ binary vector $\bar{b}$ we compare $TABLE_{I-\{v\}}(x_{i_1} = b_1,\ \ldots,\ x_{i_{n-k}} = b_{n-k},\ x_v = 0)$ (which we store as t_0) to
$TABLE_{I-\{v\}}(x_{i_1} = b_1,\ \ldots,\ x_{i_{n-k}} = b_{n-k},\ x_v = 1)$ (which we store as t_1). By this notation, we mean the value of $TABLE_{I-\{v\}}$ obtained by assigning b_j to variable x_{i_j} for $j = 1, 2, \ldots, n-k$ and assign 0 to variable x_v. In other words, this gives us an identifier for the restricted function

$$f|_{x_{i_1}=b_1,\ \ldots,\ x_{i_{n-k}}=b_{n-k},\ x_v=0}\ .$$

For each such pair (t_0, t_1) we determine whether we need a new node labelled v in $\text{BDD}(\text{f}, \langle v, \pi_{I-\{v\}}\rangle)$, using the following three criteria:

(1) If $t_0 = t_1$ then we do not create a new node since its left son and right son links would point to the same node (see collapsing operation (i) in section 2.1).

(2) If $id(t_0, t_1)$ is non-nil then it holds some node m labelled v and having the same left and right sons that the new node would have; thus we do not create a new node since it would be equivalent to m (see collapsing operation (ii) in section 2.1).

(3) If $id(t_0, t_1)$ = nil then we create a new node named with the next available node number; this is achieved by incrementing count and assigning its value to $id(t_0, t_1)$.

Thus, in the first case, we assign to $TempTable(x_{i_1} = b_1, \ \ldots, \ x_{i_{n-k}} = b_{n-k})$ the value t_0; in either of the other two cases we assign it $id(t_0, t_1)$.

After the folding operation (i.e. after examining each $\bar{b}$), we have

$$\text{count} = \sum_{v \in I} \text{cost}_v(\text{f}, \langle v, \pi_{I-\{v\}} \rangle) \ .$$

If this is less than the current minimum then we save count as MinCost_I, $\langle \pi_{I-\{v\}}, v \rangle$ as π_I, and $TempTable$ as $TABLE_I$.

Thus, after examining each v in this way, we set

$$\text{MinCost}_I \leftarrow \min_{v \in I}(\text{cost}_v + \text{MinCost}_{I-\{v\}}) \ .$$

We let π_I be the ordering $\langle v, \pi_{I-\{v\}} \rangle$ that achieves this minimum, and we let $TABLE_I$ be the truth table corresponding to $\text{BDD}(\text{f}, \pi_I)$.

```
FOR k ← 1 TO n DO
   FOR each k-element subset I ⊆ {1,2,...,n}
      [[ Compute π_I, TABLE_I and MinCost_I ]]
       BEGIN
          MinCost_I ← ∞ ;
          FOR each v ∈ I DO
             BEGIN
                [[ Evaluate cost_I with the ordering ⟨π_{I-{v}}, v⟩ ]]
(*)              id(t_0,t_1) ← nil, for each pair (t_0,t_1) ;
                 count ← MinCost_{I-{v}} ;
                 Let i_1, i_2, ..., i_{n-k} denote the elements of {1,...,n} - I ;
                 FOR each b̄ ∈ {0,1}^{n-k} DO
                    BEGIN
                       t_0 ← TABLE_{I-{v}}(x_{i_1} = b_1, ..., x_{i_{n-k}} = b_{n-k}, x_v = 0) ;
                       t_1 ← TABLE_{I-{v}}(x_{i_1} = b_1, ..., x_{i_{n-k}} = b_{n-k}, x_v = 1) ;
                       IF t_0 = t_1
                          THEN TempTable(x_{i_1} = b_1, ..., x_{i_{n-k}} = b_{n-k}) ← t_0
                          ELSE BEGIN
                                  IF id(t_0,t_1) = nil
                                     THEN BEGIN
                                             [[ the pair (t_0,t_1) is new ]]
                                              count ← count + 1
                                              id(t_0,t_1) ← count ;
                                          END ;
                                  TempTable(x_{i_1} = b_1,...,x_{i_{n-k}} = b_{n-k}) ← id(t_0,t_1)
                               END
                    END [[ for each b̄ ]] ;
                IF count < MinCost_I THEN BEGIN
                                             MinCost_I ← count ;
                                             π_I ← ⟨v, π_{I-{v}}⟩ ;
                                             TABLE_I ← TempTable
                                          END
          END [[ for each v ]]
      END [[ for each I ]] ;
Return π_{1,2,...,n} .
```

Figure 4

3.4 Complexity analyses

We first analyze the time required by the algorithm. For each k from 1 to n, we process each of the $\binom{n}{k}$ sets I of cardinality k. For each of the k indices v in each such I, we do a folding operation on $TABLE_{I-\{v\}}$. Processing each of the 2^{n-k} table entries requires two lookups in $TABLE_{I-\{v\}}$ and one lookup in *id*. Lookups in $TABLE_{I-\{v\}}$ are easily performed in $O(n-k+1)$ time. We could implement *id* as a balanced tree (such as an AVL tree [AH]) and do insertions and lookups in time proportional to the logarithm of its maximum number of entries. Thus, instead of initializing *id* to nil explicitly for each pair (t_0, t_1) as is suggested in line (*), we simply initialize *id* to the empty set. Since at most one entry is made into *id* for each $\bar{b} \in \{0,1\}^{n-k}$, the insertions and lookups in *id* also can be performed in $O(\log(2^{n-k+1})) = O(n-k+1)$ time. Thus the total time complexity of the algorithm is of order

$$\sum_{k=1}^{n} \binom{n}{k} k 2^{n-k}(n-k+1)$$

which is $O(n^2 3^n)$. We show this as follows:

$$\begin{aligned}
\sum_{k=1}^{n} \binom{n}{k} k 2^{n-k}(n-k+1) &= \sum_{k=1}^{n} \binom{n}{k} k 2^{n-k}(n-k) + \sum_{k=1}^{n} \binom{n}{k} k 2^{n-k} \\
&= \sum_{k=1}^{n} n(n-1)\binom{n-2}{k-1} 2^n \left(\frac{1}{2}\right)^k + \sum_{k=1}^{n} n \binom{n-1}{k-1} 2^n \left(\frac{1}{2}\right)^k \\
&= n(n-1)2^{n-1} \sum_{k=0}^{n-1} \binom{n-2}{k} \left(\frac{1}{2}\right)^k \\
&\quad + n2^{n-1} \sum_{k=0}^{n-1} \binom{n-1}{k} \left(\frac{1}{2}\right)^k
\end{aligned}$$

$$= n(n-1)2^{n-1}\left(1+\frac{1}{2}\right)^{n-2} + n2^{n-1}\left(1+\frac{1}{2}\right)^{n-1}$$

$$= 2(n^2-n)3^{n-2} + n3^{n-1} = O(n^2 3^n)$$

To analyze the space complexity, first note that at iteration k of the main loop the only $TABLE$ lookups are in $TABLE$s computed at the previous (i.e. the $(k-1)$st) iteration. Hence, at any time, we need only store the $TABLE$s from two consecutive iterations (actually, since we employ the order notation, we need consider only the storage required at one iteration). The storage required then, is

$$\max_{0 \le k \le n} \binom{n}{k} 2^{n-k} .$$

We now show that the maximum is attained at $k = \lfloor n/3 \rfloor$, yielding an $O(3^n/\sqrt{n})$ bound.

Letting $f(k) = \binom{n}{k}2^{n-k}$, the function to be maximized, we first show that for all $1 \le k \le n-1$, we have:

$$\frac{f(k+1)}{f(k)} < \frac{f(k)}{f(k-1)} \iff f(k+1)f(k-1) < f(k)f(k)$$

$$\iff \frac{n!\,2^{n-k-1}}{(k+1)!\,(n-k-1)!} \cdot \frac{n!\,2^{n-k+1}}{(k-1)!\,(n-k+1)!}$$

$$< \frac{n!\,2^{n-k}}{k!\,(n-k)!} \cdot \frac{n!\,2^{n-k}}{k!\,(n-k)!}$$

$$\iff k!\,(n-k)!\,k!\,(n-k)!$$

$$< (k+1)!\,(n-k-1)!\,(k-1)!\,(n-k+1)!$$

$$\iff (n-k)\cdot k < (k+1)\cdot(n-k+1)$$

$$\iff 0 < n+1$$

which is true. That is, as k increases, the ratio between consecutive $f(k)$ decreases. Furthermore, we have $f(1)/f(0) = \frac{n}{2}$, while $f(n)/f(n-1) = \frac{1}{2n}$; so as k goes from 1 to n, $f(k)/f(k-1)$ starts out greater than 1, is strictly decreasing, and ends up less than 1. That is, $f(k)$ increases, attains a maximum when $\frac{f(k)}{f(k-1)} > 1 \geq \frac{f(k+1)}{f(k)}$, and finally decreases. Thus, we are looking for the greatest k such that $f(k) > f(k-1)$, that is:

$$\frac{n!\,2^{n-k}}{k!\,(n-k)!} > \frac{n!\,2^{n-k+1}}{(k-1)!\,(n-k+1)!}$$

i.e.

$$\frac{1}{k} > \frac{2}{n-k+1}$$

or simply

$$n \geq 3k \ .$$

Thus, the maximum is at $k = \lfloor \frac{n}{3} \rfloor$.

Finally, we compute $f(\lfloor \frac{n}{3} \rfloor)$. We assume, for simplicity, that n is a multiple of three; otherwise, we may easily add one or two dummy variables without increasing the asymptotic complexity. Thus,

$$\begin{aligned}
f\left(\frac{n}{3}\right) &= \binom{n}{n/3} 2^{2n/3} \\
&= \frac{n!\,2^{2n/3}}{(n/3)!\,(2n/3)!} \\
&= O\left(\frac{\sqrt{2\pi n}\,(n/e)^n\,2^{2n/3}}{\left(\sqrt{2\pi n/3}(\frac{n}{3e})^{n/3}\right)\left(\sqrt{4\pi n/3}(\frac{2n}{3e})^{2n/3}\right)}\right) \text{ (using Stirling's formula)} \\
&= O\left(\frac{\sqrt{n}\cdot n^n \cdot (1/e)^n \cdot 2^{2n/3}}{\left(\sqrt{n}\,n^{n/3}\,(\frac{1}{3})^{n/3}\,(\frac{1}{e})^{n/3}\right)\left(\sqrt{n}\,n^{2n/3}\,(\frac{1}{3})^{2n/3}\,(\frac{1}{e})^{2n/3}\,2^{2n/3}\right)}\right) \\
&= O\left(\frac{\sqrt{n}\cdot n^n \cdot (1/e)^n \cdot 2^{2n/3}}{n \cdot n^n \cdot (1/3)^n \cdot (1/e)^n \cdot 2^{2n/3}}\right) \\
&= O\left(3^n/\sqrt{n}\right)
\end{aligned}$$

3.5 Remarks

Perhaps the most practical way to implement the set *id* is by storing its distinct values in a hash table. The time required would then be $O(n3^n)$ (expected case) and space requirements would be less but asymptotically the same. Furthermore, this would be far easier to implement than a balanced tree scheme.

Multi-valued logic can be represented by decision diagrams whose terminals have values from $\{0, 1, \ldots, k\}$ where k may be an arbitrary integer. Our algorithm generalizes in a very straightforward way to handle this case.

Chapter 4: Combinational Logic Verification

4.1 The Projective Binary Decision Diagram (pBDD)

Using ordered BDDs, we are able to represent compactly, and quickly perform operations upon, a broad class of boolean functions in common use. In this chapter, our goal is to broaden the class of boolean functions that can be represented compactly. We do this by relaxing our notion of variable ordering. In doing this, we also need to preserve those properties of ordered BDDs which make them useful. In particular, we want unary operations to take time proportional to the size of the graph, and binary operations to take time proportional to the product of the sizes. Both of these properties apply in the structure we will be discussing; however, testing satisfiability cannot be done in unit time, as it can with ordered BDDs.

The paradigm of previous chapters was: given a single variable ordering, each path between root and terminal in each BDD must conform to that ordering in order for BDD operations to be well defined. In our new paradigm, we allow a different variable ordering along each path of a given BDD, but require that, given any bit-vector $\bar{b}$, the path traversed to evaluate $f(\bar{b})$ have the same variable ordering in all BDDs. This idea is formalized below.

Consider the binary decision tree of Figure 5 (top), which is identical to Figure 1 except that there are no truth values at the leaves. By filling in the leaves with truth values, we can represent any boolean function. If we are performing a sequence of BDD operations using the variable ordering $\langle 2,1,4,3\rangle$, each BDD we compute can also be computed by filling in this tree with the appropriate truth values, then collapsing it. Thus we may say that this tree is the *template* for BDDs with the variable ordering $\langle 2,1,4,3\rangle$. A template, then, is merely a binary tree with decision variables at the interior nodes, ***but without truth values at the leaves.*** If we fill in the leaves of a template with truth values, it becomes a (not necessarily ordered) BDD representation of a particular boolean function. Thus the template determines the variable ordering along each path, just as π (in Chapter 2) determined the variable

ordering along all paths. The analogy can be taken further: if we drop collapsing rule (i) from the definition of an ordered BDD (Section 2.1), we can consider it to be a pBDD with the template consisting of one node for each variable, with both links pointing to the next node in the variable ordering. This is illustrated in Figure 9.

Thus, in Chapter 2, we dealt with a highly restricted class of templates: those with the same variable ordering along each path from the root, and no decision variable repeated along any path. In this chapter, we will deal with a broader class of templates, the only restriction being that each variable must appear at least once on each path from the root. (Without this restriction, there would be templates which could not represent some boolean functions.) The idea is that we will use a single template throughout a sequence of BDD operations, thus giving us a framework for efficient operations on boolean functions.

Because a template (as defined) is bigger than a truth table, we shall instead use a "collapsed" version of the template, which we define to be any DAG labelled analogously to the tree, for which any truth assignment to the variables will involve traversing nodes in the DAG in the same order as in the tree. This DAG will, we hope, be much smaller than the full tree. Figure 5 (bottom) shows an uncollapsed, unordered template, and Figure 10 (left) shows the corresponding collapsed template (which will be called simply a *template* from now on).

Again, the purpose of a template is to define, for a class of BDDs, the ordering of decision variables along each of the paths from the root to the terminals. In a logic verification application, one typically deals with a known subset of the possible boolean functions. Thus, one would choose a template such that the particular boolean functions used would have compact representations. We develop such a template (for multipliers) in Section 4.3.

When we represent a boolean function using a BDD whose variable ordering along each path conforms to a given template, we say that the boolean function is *projected* onto the template, resulting in a *projective binary decision diagram*, or

pBDD.

4.2 Operations on pBDDs

In the previous section, we noted the correspondence between ordered BDDs and pBDDs illustrated in Figure 9. This correspondence implies that a pBDD can be exponentially larger than its template (since an ordered BDD can have size proportional to 2^n, where n is the number of variables). We can avoid blowups in practice if (i) a template is chosen that is appropriate to the function, (ii) the boolean expression (or circuit) is not much larger than it needs to be (i.e. wasted computation can cause the pBDD to blow up), and (iii) we collapse the nodes intelligently. We will address this last point later in the chapter.

In general, the projection of a boolean expression onto a template will have a many-to-one mapping of all its nodes onto nodes of the template that is a homomorphism under the functions label, 0-link, and 1-link. It will generally be bigger than the template, because each node in the template can be reached by many paths from the root, and just as with ordered BDDs, each node of a pBDD corresponds to a unique boolean function.

As with ordered BDDs, we compute a pBDD representation for the output bit of a circuit by computing representations for all the inputs, then applying the gate operations from left to right.

4.2.1 Projecting a variable onto a template

Here is one way to project a template onto the boolean expression v, for any variable v in the template:

1. Make three copies of the template. Label all the terminals of the first copy 0 (the *0-copy*), and label all the terminals of the second copy 1 (the *1-copy*). Call the third copy the *root-copy*.

2. For each node in the template with decision variable v (call the node n, and its copies n_0, n_1, and n_r), direct the 0-link of n_r to the node

pointed to by the 0-link of n_0, and the 1-link of n_r to the node pointed to by the 1-link of n_1. [Direct the 0-link of n_1 and the 1-link of n_0 to a dummy node called *don't-care*.]

3. Discard any nodes not reachable from the root of the root-copy.

The method (not including the bracketed instruction in step 2) is illustrated in Figure 10. Perhaps the easiest way to follow the figure is to look at a cluster of three nodes, and to follow the three 0-links or 1-links simultaneously. If the nodes are labelled v, then the links all point to the same node; otherwise, they point to the three nodes of another cluster.

Note that we may depart from the pBDD model slightly by introducing don't-care nodes. These are placed at the ends of *inconsistent* paths, which we define to be any path for which there exists a decision variable v such that traversing the path requires taking the 0-link of a node labelled v and the 1-link of another node labelled v. This modification has advantages which will become apparent later on.

4.2.2 Applying a binary operation to two pBDDs

Applying a binary operator to two pBDDs requires a bit more bookkeeping than for ordered BDDs. Given two pBDDs B_1 and B_2, and a binary operator $\odot$, we may compute $B_1 \odot B_2$ by applying to their roots a recursive algorithm analogous to that for ordered BDDs:

(1) Call the two nodes n_1 and n_2. If they are both terminals, then return the terminal labelled $n_1 \odot n_2$. Otherwise, if at least one is a don't-care node, return a don't-care node.

(2) Otherwise, n_1 and n_2 are both internal nodes corresponding to the same node of the template. Thus they have the same decision variable (call it v). Return a node labelled v with the i-link (for $i = 0, 1$) pointing to the node that results from recursively applying this algorithm to the i-link of n_1 and the i-link of n_2.

After the algorithm is applied, we are left (as with the ordered BDD algorithm) with a pBDD whose size is proportional to the product of the sizes of the two pBDDs B_1 and B_2. It remains to collapse the diagram.

4.2.3 Collapsing a pBDD

For this purpose, we use a slightly different definition of "equivalent" from the one we used in Chapter 2. In order for nodes n_1 and n_2 to be equivalent, the following must be true:

(1) Nodes n_1 and n_2 must correspond to the same node in the template, and

(2) Either n_1 or n_2 is a don't-care, or their 0-links are equivalent and their 1-links are equivalent.

Since, by this definition, there are several ways to form equivalence classes from each set of nodes satisfying (1), it is clear that collapsing *must* proceed from the terminals to the root. Of course, we may apply an arbitrary amount of lookahead (analogous to the heuristics in [NB], see section 3.1) to improve our final result at the expense of increased running time.

Since the pBDD representation generally is not canonical, if we are careless we may end up with pBDDs that are not merely exponential in the number of variables, but (using the multiplier template as an example) exponential in the square of the number of variables (the "depth" of the template, as we will see in Section 4.3).

Fortunately, there is no real need to find the exact optimum. Unlike the problem addressed in Chapter 3, the collapsed pBDD will typically be an intermediate representation to which other pBDD operations will be applied. Furthermore, we have presumably chosen a template such that the final pBDD is expected to be compact.

Given the above definition of equivalence, the element of nondeterminism in the collapsing process is deciding with which of (possibly) many nodes to collapse

the don't-cares. In our actual implementation, we adopted the simplest possible heuristic: namely, that we never collapse a don't-care node with a "care" node. This heuristic has two advantages: its time-complexity is linear in the size of the uncollapsed pBDD (expected case), and the final pBDD will have the property that all inconsistent paths are marked don't-care. This latter property speeds tests for satisfiability, as will be seen in Section 4.2.4. However, the heuristic has the disadvantage that the resulting pBDDs are large (though for multipliers, they still grow more slowly than ordered BDDs, cf Section 4.4).

We now present an alternate heuristic, a "greedy" heuristic which results in smaller pBDDs but takes longer to compute for a given size of uncollapsed pBDD. We believe that this heuristic would result in faster design verification on problem sizes significantly larger than in our trials.

By greedy heuristic, we mean that at each step we will collapse in such a way as to build nodes with the highest indegree possible. The intuition is that this will increase the likelihood of being able to collapse nodes closer to the root. An analogy: If you roll two dice, each with three red sides and three white sides, the probability of rolling the same color on both dice is $\frac{1}{2}$. However, if the dice each have five red sides and one white side, the probability rises to $\frac{13}{18}$.

So, given an uncollapsed pBDD, for each node of the template proceeding from the terminals to the root, we collapse the pBDD nodes corresponding to that template node as follows.

1. If the template node is a terminal, we collapse all the TRUEs into one node and all the FALSEs into one node. The don't-cares will be collapsed later. Stop.

2. Otherwise, the template node is an internal node. We divide the set of corresponding pBDD nodes into four types.

   2.1. Don't-cares. These will be collapsed when their parents are collapsed.

   2.2. Nodes whose 0-links are don't-cares. Collapse together all sets of nodes with identical 1-links, and sort the sets by indegree.

   2.3. Nodes whose 1-links are don't-cares. Collapse together all sets of nodes with identical 0-links, and sort the sets by indegree.

   2.4. Nodes for which neither link is a don't-care. Collapse together all sets of nodes with identical 0-links and identical 1-links. Sort the sets by their indegree, plus the indegree of the corresponding set in 2.2, plus the indegree of the corresponding set in 2.3.

3. Now repeat 3.1 and 3.2 until all don't-care links are gone.

   3.1. If the sum of the largest keys in 2.2 and 2.3 is greater than the largest key in 2.4, then collapse the first node in 2.2 with the first node in 2.3, and subtract their indegrees from the keys in 2.4 to which they contribute.

   3.2. Otherwise, collapse the first node in 2.4 with the corresponding nodes in 2.2 and 2.3 and subtract the indegrees of the nodes in 2.2 and 2.3 from other keys in 2.4 to which they contribute.

4. If there are nodes of type 2.2 or 2.3 left (there cannot be both), replace all their don't-care links with links to the node of greatest indegree that corresponds to the same template node as the don't-care.

"Greedy" collapsing heuristic for pBDDs

We illustrate Step 3 of the collapsing heuristic in Figure 11. Assume that after Step 2, the nodes corresponding to node a of a template have been broken up into the five sets shown above the line. We can collapse a_1 with a_3, for a total indegree of 12, or we can collapse a_2, a_3 and a_5 for a total indegree of 13, or a_4 and a_5 for a total indegree of 12. Since the second option yields the largest indegree, we collapse a_2, a_3, and a_5 together to become a'_1. This leaves a_1 and a_4. Since their 0-links are different, no further collapsing is possible.

Actually, the efficacy of this heuristic may be improved by performing step 2 on each node of a level of the template, and then performing step three on all the nodes (of the level) together.

One way to collapse pBDDs even further is to find and eliminate nodes to which all paths from the root are inconsistent. Unfortunately, we have made little progress in this area.

4.2.4 Testing Satisfiability

A pBDD is unsatisfiable iff all paths leading to a terminal valued TRUE are inconsistent. As noted previously, the simple collapsing heuristic guaranteed that all inconsistent paths were marked don't-care. Thus, if we always collapse according to this heuristic, then a pBDD is satisfiable iff some of its terminals are TRUE. If we use some other collapsing heuristic, testing satisfiability becomes more time-consuming. We present one such method, and comment on its performance in practice.

Our method merely involves transforming the pBDD into an ordered BDD, and checking whether the ordered BDD is identical to the terminal FALSE. We perform this transformation as follows:

1. $B \leftarrow$ the original pBDD.

2. For each variable v in turn:

   a. $B_0 \leftarrow$ a copy of B with each node labelled v replaced by its 0-link.

   b. $B_1 \leftarrow$ a copy of B with each node labelled v replaced by its 1-link.

c. $B \leftarrow$ the BDD with B_0 and B_1 the respective 0-link and 1-link of the root, which is labelled v.

d. Collapse B according to the collapsing rules for ordered BDDs, with the additional rule that if nodes n_1 and n_2 are labelled with the same variable, and n_2 is at the i-link of n_1, then redirect the i-link of n_1 to point to the i-link of n_2.

At each iteration of step 2, we pick the variable which appears at the nodes of B with greatest frequency.

Since we presumably are using pBDDs because we happen to be dealing with functions that have unwieldy ordered BDD representations, we would expect that this method would produce large intermediate BDDs. However, our experience with multipliers contradicts this: in our tests, B stays about the same size as the precollapsed pBDD for about $\frac{2}{3}$ of the iterations, then shrinks quickly.

4.3 The Multiplier Template

In Chapter 2, we noted that any ordered BDD representation for the nth bit of the output of an $m \times m$ multipler must be of size proportional to 2^n. In this section, we construct a class of templates which can lead to pBDD representations of size proportional to n^3.

Briefly, our method is to construct a BDD which represents the given output bit, and throw away the values at the terminals. A more detailed explanation follows.

Assume we are trying to multiply vectors $(a_{m-1}a_{m-2}\dots a_0)$ and $(b_{m-1}b_{m-2}\dots b_0)$ (the method easily generalizes to $m_1 \times m_2$ multipliers, for $m_1 \neq m_2$). We consider a_0 and b_0 to be the least significant bits of the multiplicands. Also, for $0 \leq i \leq 2m$, let c_i be defined as follows:

$$c_i = \sum_{j=0}^{n} a_j b_{i-j}$$

where we define $a_k, b_{i-k} = 0$ if $k < 0$ or $k > n$ (similarly for $i-k$). To compute the nth bit of the multiplication (again, bit 0 is the least significant bit) for $n \geq 1$, we

could compute c_1, divide by two and truncate, add c_2, divide by two and truncate, add c_3, and so forth up to c_n. From the result of this computation we would then take the least significant bit for the final answer.

Following this method, we build the template in n stages. At stage i, $1 \leq i \leq n$, we build a tally diagram to count the number of product terms in c_i that are ones, then we pair up the links on the bottom to feed into the next stage. Implementing this tally-and-halve computation in a BDD is illustrated in Figure 6. The bottom stage, instead of counting, need only compute parity (XOR). At this point, the template appears as in Figure 6a.

Finally, we expand each decision node $a_j b_k$ into a pair of nodes that performs the AND computation. It is easy to do this in such a way as to satisfy the constraint that all variables must appear on each path from the root to a terminal. One method is illustrated in Figure 7.

Since the final template has $O(n^2)$ rows, with $O(n)$ nodes per row, the space complexity of the template is indeed $O(n^3)$. More precisely, a simple counting argument shows that the number of nodes (including the two terminals) is n^3+3n+6 if $m \geq n$, and fewer otherwise.

Note that in this particular template, all the nodes at a level are labelled with the same variable. Thus, this template is suitable for implementation as a DCVS tree (as described in Section 3.1 and [NB] et al). This also means that we can get away with one node per level, and a space complexity of $O(n^2)$ for the template (see Figure 8). However, representing a multiplier bit with this template still requires space proportional to n^3. We prefer the $O(n^3)$ template because it will make verification of a multiplier circuit easier, as will be explained in section 4.4.

4.4 Combinational Logic Verification in Practice

In general, to compare two pBDDs for equivalence, we would compute the exclusive-or of the pBDDs and test the result for satisfiability.

To test that a given circuit correctly performs multiplication, we first build the appropriate template as per Section 4.3. Then we use the methods of Section 4.2 to build up a pBDD which performs the same computation as the circuit: we project each input line onto the template, then for each gate, we apply the gate operation to its inputs and somehow collapse the resulting pBDD. Finally, we test the pBDD corresponding to the output line.

As noted in Section 4.3, we constructed the multiplier template by building a multiplier (unordered) BDD and throwing away the values at the two terminals. Thus, to verify our pBDD, all we need do is negate the values of all nodes corresponding to the 1-terminal in the template, and test the result for satisfiability; if it is unsatisfiable, then the circuit is correct.

The running time of a verification algorithm based on pBDDs is generally proportional to (i) a polynomial in the size of the template, (ii) the number of gates, and (iii) the largest pBDD generated. A "bare-bones" approach, in which little care is taken to keep the pBDDs small, tends to keep term (i) small, while term (iii) blows up quickly. Conversely, efforts to stem the blowup of term (iii) tend to increase the degree of term (i).

We implemented the bare-bones approach, and used it to verify outputs c_3 (depends on 8 inputs) through c_8 (depends on 18 inputs) of a multiplier circuit. The circuit itself was a 16×16 multiplier built by first computing the sixteen 16×1 subproducts, then summing them using a "tree" of fourteen carry-save adders, with a carry-propagate adder at the end. The design (which we found in [HB]) is shown schematically in Figure 14(a).

The results of our experiments are as follows:

out-bit	#inputs	pBDD size	growth	CPU seconds pBDDs	DBAs	ratio(%)
3	8	226	---	1.57	1.35	86.0
4	10	915	4.049	5.16	3.72	72.1
5	12	3387	3.702	22.46	13.48	60.0
6	14	10588	3.126	86.19	58.32	67.7
7	16	32941	3.111	335.63	238.46	71.0
8	18	97174	2.950	4110.05	3895.66	94.8

Column 3 lists the actual size of the largest pBDD in each trial (term (iii) above). Column 4 shows the rate of blowup; that is, the ratio of the maximum pBDD size in the current trial to that of the previous trial. We argue that even when we do not make an effort to keep the pBDDs small, they grow significantly more slowly than 2^n in the number of inputs (though admittedly much worse than the ideal of n^3).

In Columns 5–7, we compare the running times for our implementation of verification with pBDDs to our implementation of the Differential Boolean Analyzer of [SB]. We argue that the slower growth of the term (iii) in our method compensates for the high-degree of term (i), and we expect that our method would surpass the performance of the DBA on the next larger problem size. More strenuous efforts to collapse the pBDDs would result in slower running times in these instances, but faster running times on larger problem sizes. Of course, replacing the BDD model with Typed Shannon's canonical form ([MB] and Chapter 2) would result in faster running times on all problem sizes (at the expense of extra coding).

All trials were conducted using C on a VAX 8650 except for the last, which was done on a VAX 785 due to memory constraints. The 785 runs about 3.5 times more slowly than the 8650 on this problem.

Chapter 5: Sequential Logic Verification

5.1 Overview

The problem that we address is the following: given are two single-output circuits constructed from logic gates and latches, each having no race conditions, i.e. all loops contain at least one latch. The given circuits need not have the same number of latches (in fact, one may be purely combinational and the other sequential). We are also given an integer n indicating the number of "conceptual" inputs for the two circuits. Often, the two circuits each have n "real" inputs; but it may be that one or the other or both of the circuits has fewer than n real inputs because it is regarded as a serial implementation of some function. Furthermore, for each of the circuits we are given a number indicating the number of clock cycles between the arrival of its new inputs; these two numbers need not be equal. The problem is to decide whether the two circuits are functionally equivalent and, if they are not, to produce a sequence of inputs for which their outputs differ.

This problem arises in several ways, among them:

(1) a design may be replaced by a newer design which perhaps is faster, cheaper or which utilizes a different technology or parts library. In this case one would like to verify that the new design is indeed functionally equivalent to the old,

(2) one can construct a verification system that checks whether a given higher (say, register-transfer) level description of a circuit is equivalent to a given logic level description, as follows: first synthesize a simple (unoptimized) logic level description from the behavioral descriptions, and then check it for equivalence with the given logic level description.

Our solution to this problem consists of deriving from each circuit an n-DFA (which will be described in section 5.3). We then check whether these two n-DFAs accept the same language, which happens if and only if the two circuits are functionally equivalent.

One drawback of our method is that its space and time complexity are both polynomial in the number of states, and hence exponential in the number of latches. Note, however, that this is also characteristic of the temporal logic approach to this problem [BC]. On the other hand, our algorithm is typically polynomial in the number of circuit inputs for typical circuits (this claim is supported by the claims of [B2] and by our limited computational experience thus far). Hence we regard our method as useful for circuits with a rather small amount of memory, and any reasonable number of inputs.

Section 5.2 contains a more formal statement of the problem, including a precise definition of functional equivalence. Section 5.3 contains a defintion of the n-ary Deterministic Finite Automaton, or n-DFA, which we make use of in our algorithm. Our algorithm is presented in Section 5.4, along with certain key implementation details in Section 5.5.

5.2 The problem

Each circuit C considered here is constructed from D-flipflops (latches) and logic gates such as ANDs, ORs and NOTs, and is subject to the following design constraints:

(i) C has exactly one output, which is also the output of a latch.

(ii) Each feedback loop in C contains a latch — that is, there can be no race conditions.

(iii) Logic gates are assumed to have no delay, and latches to have unit delay (i.e. one clock cycle).

(iv) All primary inputs arrive over $\Delta(C)$ cycles, for some integer $\Delta(C)$. In other words, the computation being performed by the circuit may change only at time $k\Delta(C)$ for $k = 1, 2, \ldots$ Thus the circuit is when-determinate [Ul].

Our verification problem is as follows: we are given circuits C_1 and C_2 and a start state (i.e. an initial assignment to the latches) q_1 and q_2 for C_1 and C_2,

respectively. We also given an integer n which is the number of primary inputs of both C_1 and C_2.

Now if C is an n-input circuit, w an (infinite) binary sequence, z an integer, and q_0 a state of C, then $C(w, z, q_0)$ denotes the output of C after z Δ cycles, starting from state q_0, where the first n bits of w are input to C initially, followed by the second n bits of w, and in general bits $(r-1)n+1$ through rn of w constitute the rth set of inputs to C. Our task is to decide whether C_1 and C_2 are *equivalent*, by which we mean that for all infinite binary sequences w, for all integers $r \geq 1$,

$$C_1(w,\ r,\ q_1) = C_2(w,\ r,\ q_2)$$

Conceptually, for $i = 1, 2$, C_i is regarded as implementing an n-input function, and is fed these inputs in a well-defined manner, requiring a total of $\Delta(C_i)$ clock cycles. Thus the equation above means that C_1 and C_2 have the same output after each has processed the same (arbitrarily long) sequence of sets of conceptual input.

As an example, C_1 may be a purely combinational 16-bit adder (where the output represents the high order bit of the sum), and C_2 may be a serial implementation of the same circuit, which accepts 2 bits (one from each operand) at a time. Then $n = 32$, $\Delta(C_1) = 1$ and $\Delta(C_2)$ would most likely equal 16.

5.3 The n-ary Deterministic Finite Automaton (n-DFA)

We now present a compact way to represent the state diagram of a single- output sequential circuit.

We define an n-DFA to be a 5-tuple (V, E, s, β, F) where (V, E) is a directed graph, $s \in V$ and $F \subseteq V$ are distinguished vertices (s may be thought of as the start vertex and F as the final vertices), and β is a function labelling each $e \in E$ with a boolean expression over the variables $X_m = \{x_1, x_2, ..., x_m\}$. Furthermore, β is subject to the constraint that for each $v \in V$, if $e_1, e_2, ..., e_r$ are the edges originating

at v, then

$$\beta(e_i) \wedge \beta(e_j) \equiv 0 \ \forall i \neq j, \quad \text{and} \quad \sum_{i=1}^{r} \beta(e_i) \equiv 1 \ .$$

In other words, each assignment to the variables in X_m satisfies exactly one expression in $\{\beta(e_i) : 1 \leq i \leq r\}$.

The n-DFA may be regarded as a concise notation for the traditional DFA (see [HU]) in the case in which Σ (the input symbols) is the set of binary strings of length n. The n-DFA is potentially more concise in that the members of the alphabet are not enumerated and the transition function is not explicitly tabulated for each state-input pair. Thus, if the expression $\beta(e)$ happens to be a full minterm expansion for each $e \in E$, then the description of the n-DFA is just as long as that of its corresponding traditional DFA.

The language $L(A)$ accepted by an n-DFA A is simply the language accepted by its corresponding traditional DFA.

5.4 Our solution

Our strategy is as follows: First, for each circuit, we construct an equivalent circuit that performs its entire computation at every clock cycle. In this way, we make the circuits more directly comparable. Then, we transform these new circuits into n-DFAs. Finally, we check whether the automaton for circuit C_1 accepts the same language as that for C_2.

5.4.1 Constructing C'

The circuit C contains the following information:

I': set of conceptual inputs (the I' for both circuits must be identical)

Δ: positive integer, speed of circuit

I: set of input lines

L: set of latches

G: set of gates

function OP: $G \mapsto \{\text{and}, \text{or}, \text{not}, \text{xor}, \text{nand}, \text{etc.}\}$

function GSIG: $G \mapsto 2^{(G \cup L \cup I)}$ [signals entering a gate]

function LSIG: $L \mapsto G \cup L \cup I$ [signal entering a latch]

function F: $I \times \{1, 2, \ldots, \Delta\} \mapsto I'$ [mapping input lines to conceptual inputs]

out: $\langle out.latch, out.tick \rangle$ for $out.latch \in L, 1 \leq out.tick \leq \Delta$ [the output]

From these data, we construct C' as follows:

$\Delta' = 1$

$L' = L \cup \{\text{out}\}$

$G' = G \times \{1, 2, \ldots, \Delta\}$ [new gate identifiers are ordered pairs]

$\text{OP}'(\langle g, n \rangle) = \text{OP}(g)$ for all g,n [we make Δ copies of each gate]

function cascade: $(G \cup L \cup I) \times \{1, 2, \ldots, \Delta\} \mapsto G' \cup L' \cup I'$. Defined as follows:

$$\text{cascade}(g, n) = \langle g, n \rangle \text{ for } g \in G, 1 \leq n \leq \Delta$$

$$\text{cascade}(i, n) = \text{F}(i, n) \text{ for } i \in I, 1 \leq n \leq \Delta$$

$$\text{cascade}(l, 1) = l \text{ for } l \in L$$

$$\text{cascade}(l, n) = \text{cascade}(\text{LSIG}(l), n-1) \text{ for } l \in L, 2 \leq n \leq \Delta$$

GSIG$'$: $\text{GSIG}(g) = \{s_1, \ldots, s_j\} \Rightarrow \text{GSIG}'(g) = \{\text{cascade}(s_1), \ldots, \text{cascade}(s_j)\}$

LSIG$'$: $\text{LSIG}'(l) = \text{cascade}(\text{LSIG}(l), \Delta)$ for $l \in L$

$$\text{LSIG}'(\text{out}) = \text{cascade}(out.latch, out.tick)$$

This transformation is analogous to flattening out a loop in a program, and is illustrated in Figure 12. The circuit is a 4×4 multiplier which performs its computation in four clock cycles. The bits of one multiplicand persist throughout the computation, and the other multiplicand is fed in one bit per clock cycle. One output bit and four carry bits are generated at each clock cycle, and the carry bits

are fed back into the circuit. After the transformation, there is a copy of the circuit for each of the four clock cycles, and the latched signals from one copy feed into the next.

5.4.2 Constructing $A^{(n)}$

Given an n-input circuit C' with start state q_0, we construct the n-DFA $A^{(n)}(C') = (V, E, \beta, s, F)$ such that:

(1) the members of V are in one-to-one correspondence with the (up to) 2^k states (i.e. assignments to the k latches) of C' which can be reached from q_0 via some sequence of inputs. Hereafter we shall refer to members of V as states, in implicit reference to this correspondence.

(2) s is the member of V corresponding to q_0.

(3) F is the set of states in which the output latch has value "1".

(4) There is an edge $(v, w) \in E$ whenever state w is reachable from state v in the circuit C' in exactly one clock cycle. In this case, the boolean expression $\beta(v, w)$ is satisfied by precisely those input sets to C' causing this transition.

The edges E and the labels β are computed as follows. Number the latches 1 to k. Express the next value of the ith latch, for $1 \leq i \leq k$, as $NEXT_i(x_1, \ldots, x_n, y_1, \ldots, y_k)$ where $x_1, \ldots, x_n$ are the primary inputs and $y_1, \ldots, y_k$ are the current latch values. A boolean expression for $NEXT_i$ is computed directly from the circuit using the methods of Chapters 2 and 4 — this of course is made possible by the absence of combinational loops. Now a way to compute E and β is to do the following for each pair of states $v, w \in V$ (using some form of linear search from q_0): Let $v_1, v_2, \ldots, v_k$ $(w_1, w_2, \ldots, w_k)$ be the values of the latches corresponding to state v (resp. w). Then set

$$\beta(v, w) \leftarrow \prod_{i \in I^+} NEXT_i(x_1, \ldots, x_m, v_1, \ldots, v_k) \wedge \prod_{i \in I^-} \overline{NEXT_i(x_1, \ldots, x_m, v_1, \ldots, v_k)}$$

where $I^+ = \{i : w_i = 1\}$ and $I^- = \{i : w_i = 0\}$.

After some simplification of the $\beta(v, w)$, we check whether it is satisfiable; we include the edge (v, w) in E if and only if it is.

5.4.3 Checking the equivalence of C_1 and C_2

Now given circuits C_1 and C_2 along with $\Delta(C_1)$, $\Delta(C_2)$, q_1, q_2 and n as described in Section 2 above, we compute $A_1 = A^{(n)}(C_1)$ and $A_2 = A^{(n)}(C_2)$ and then decide whether $L(A_1) = L(A_2)$. We can do this in a way which is a generalization of a method used to test traditional DFA's for equivalence [HU]. In particular, if $A_1 = (V_1, E_1, s_1, \beta_1, F_1)$ and $A_2 = (V_2, E_2, s_2, \beta_2, F_2)$, then we construct the n-DFA

$$A' = (\ V_1 \times V_2,\ E',\ [s_1, s_2],\ \beta',\ F'\ )\ ,$$

where

(i) for all $v_1, w_1 \in V_1$, $v_2, w_2 \in V_2$,

$$\beta'([v_1,\ v_2],\ [w_1,\ w_2]) = \beta_1(v_1,\ w_1) \wedge \beta_2(v_2,\ w_2)\ ,$$

(ii) E' is, as usual, the set of pairs $(v, w) \in (V_1 \times V_2)^2$ such that $\beta'(v, w)$ is satisfiable, and

(iii) F' is $(F_1 \times (V_2 - F_2)) \cup ((V_1 - F_1) \times F_2)\ )$.

Thus A' accepts the language

$$(L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2))\ .$$

We then check whether this language is empty by determining whether there is a directed path in the graph $(V_1 \times V_2,\ E')$ from the start vertex of A' to one of its final vertices. A simple inductive proof shows that indeed C_1 and C_2 are equivalent if and only if $L(A^{(n)}(C_1)) = L(A^{(n)}(C_2))$.

Incidentally, a minor modification can be made to this method that would allow the designer to specify Don't-Care conditions on the input sequences. One way to

handle this is to allow the designer to enter a third sequential circuit C_3 accepting precisely those input sequences for which it doesn't matter whether the circuits C_1 and C_2 are functionally equivalent. In particular, C_1 and C_2 are equivalent — relative to the don't-care condition — if

$$\overline{L(A^{(n)}(C_3))} \cap L(A')$$

is empty.

5.5 Some implementation notes

Throughout the implementation, boolean functions are represented as binary decision diagrams (BDD's) as in [B2] (see also [Ak], [BF]). As discussed previously, this representation is canonical and has been found to be compact and quickly manipulable on a wide variety of boolean expressions derived from circuits. In particular, using this representation, checking a boolean expression for satisfiability (a key step in our algorithm) is usually quite fast for functions arising from real-world circuits [B2]. For our tests, we deliberately chose circuits with compact ordered BDD representations.

For variable ordering, we chose to put all the latches near the root, and all the inputs near the terminals. This choice makes it quick to restrict a BDD to a given latch state, but results in larger BDDs than are necessary. It is unclear whether the trade-off is worthwhile.

When constructing $A^{(n)}$, we first derive expressions for $NEXT_i$ for each latch i. We represent these as BDD's where the nodes are labelled with latch and input identifiers. Figure 13 schematically shows the $NEXT_i$ in Batman form.

In addition to the $NEXT_i$, we compute a BDD representing the XOR of the two circuits' outputs, again in terms of latches and inputs. Given a latch state for each circuit, we can determine the existence of a set of inputs for which the outputs don't match, simply by plugging the latch states into the BDD and checking whether the 0-terminal was reached. If the latch state pair is reachable from the starting state

pair, and if there exists, from that latch state pair, a set of inputs that cause the outputs to disagree, then the circuits are not functionally equivalent.

To compute the transitions out of a state w, we need only restrict the latch decision nodes to the latch values of w, and then take the cross-product of the $NEXT_i$. This will result in a BDD where the terminals are latch states instead of zero and one. We define the cross-product $f \times g$ of two functions f and g as follows: for all x_1 in the domain of f and x_2 in the domain of g, $(f \times g)(x_1, x_2)$ is the ordered pair $\langle x_1, x_2 \rangle$.

So, beginning with a queue containing only the start-state pair, we:

(1) If the queue is empty, return success and halt.

(2) Otherwise, pop a state pair off the queue, and compute all transitions out of the state pair as per the above paragraph.

(3) Test that all reachable state pairs' outputs match. If some don't match, return failure.

(4) Otherwise, push all reachable (but as yet unreached) state pairs on the queue, and go back to (1).

Again, this is more efficient than brute-force enumeration of cubes, because we keep track of sets of BDD nodes that have already been visited throughout the algorithm, and we don't go past them again.

We implemented this algorithm in C on a VAX 8650, and tested it by comparing two different implementations of an accumulator (shown in schematic in Figure 14(b)). Testing the fifth least significant bit of the sum, which depended on a total of 14 latches in our circuits, took about 6.2 seconds. The most directly comparable benchmark we found was [DM]. There, the authors state that they have compared two 128-state DFAs in 97 seconds (also on an 8650). In [DC], the authors required "several minutes" to build up a 128-state diagram and about 10 seconds to test the diagram against a statement in temporal logic (on a LISP machine).

Chapter 6: Conclusions and Future Work

The BDD is a powerful data structure. Here, we have extended its usefulness in several ways. We have presented an algorithm for minimizing ordered BDDs which reduces the time-complexity of this operation from factorial to exponential. We have generalized the notion of variable ordering in BDDs so that multipliers may be represented in polynomial space, and we have shown how to perform combinational logic verification using this more general structure. Also, we have adapted the DFA for use in sequential logic verification.

In Chapter 3, our algorithm produced the smallest possible ordered BDD from a given truth table. If, instead of a truth table, we were to choose a more compact representation for the input (such as a circuit diagram), we could speed up the algorithm dramatically in those cases where there exists a small BDD. Our only requirement is that the folding operation be easy to apply to the representation we choose.

Perhaps the most effective way to collapse pBDDs would be to detect and remove unreachable nodes. Since the practicality of pBDD algorithms depends heavily on the pBDDs not blowing up exponentially, we believe that this would be the most valuable extension to the work of Chapter 4.

Another area for future work would be to determine whether the state minimization algorithm of [Ho] could be adapted for use on n-DFAs. This would improve the performance of the sequential logic verification algorithm of Chapter 5.

REFERENCES

[AH] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Ak] S. B. Akers, "Binary Decision Diagrams," *IEEE Trans. Comput.*, Vol. C-27, No. 6 (June 1978), pp. 509–516.

[AR] M. S. Abadir and H. K. Reghbati, "Functional Test Generation for Digital Circuits Described Using Binary Decision Diagrams," *IEEE Trans. on Computers*, Vol. C-35, No. 4 (April 1986), pp. 375–379.

[Ba] Barrow, H. G., "Proving the Correctness of Digital Hardware Designs," *VLSI Design* (July 1984), pp. 64-77.

[BC] Browne, M., E. Clarke, D. Dill and B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic," Dept. of Computer Science, Carnegie-Mellon University, Tech. Rep. No. CMU-CS-85-100, December 1984.

[B1] R. E. Bryant, "Symbolic Verification of MOS Circuits," in Henry Fuchs (ed.), *1985 Chapel Hill Conf. on Very Large Scale Integration,* Computer Science Press, pp. 419–438.

[B2] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, Vol. C-35, No. 8 (August 1986), pp. 677–691.

[Da] Darringer, J., "The Application of Program Verification Techniques to Hardware Verification," *Proc. 16th Design Automation Conf.*, (June 1979), pp. 375-381.

[DC] Dill, D., and E. Clarke, "Automatic Verification of Asynchronous Circuits Using Temporal Logic," in Henry Fuchs (ed.), *1985 Chapel Hill Conf. on Very Large Scale Integration,* Computer Science Press, pp. 127–143.

[FS] Friedman, S. J. and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," *Proc. 24rd Design Automation Conf.* (June 1987), pp. 348–356.

[Ge] German, S. M., "Functional Verification of Generic Hardware Designs in the Language Zeus," manuscript.

[HB] Hwang, Kai and Fayé A. Briggs, *Computer Architecture and Parallel Processing* (McGraw-Hill, 1984), pp. 170–171.

[HG] L. G. Heller, W. R. Griffin, J. W. Davis and N. G. Thoma, "Cascode Voltage Switch Logic—a Differential Logic Family," *31st Int. Solid State Circuits Conf.—Digest of Technical Papers* (1984), pp. 16–17.

[Ho] Hopcroft, J. E., "An *nlogn* algorithm for minimizing the number of states in a finite automaton," in (Z. Kohavi, ed.) *The Theory of Machines and Computations*, pp. 189-196, Academic Press, New York, 1971.

[HU] Hopcroft, J. E. and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.

[KN] Kapur, D. and P. Narendran, "An Equational Approach to Theorem Proving in First-Order Predicate Calculus," *Software Engineering Notes*, Vol. 10 No. 4 (August 1985), pp. 63–66.

[Le] C. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell Syst. Tech. Journal,* No. 38 (July 1959), pp. 985–999.

[MB] Madre, J-C. and J-P. Billon, "Proving Circuit Correctness using Formal

Comparison Between Expected and Extracted Behaviour," *Proc. 25rd Design Automation Conf.* (June 1988), pp. 205–210.

[Mc] McFarland, M., "Mathematical Models for Formal Verification in a Design Automation System," Ph.D. Thesis, Carnegie-Mellon University, Tech. Rep. DRC-02-06-81, July, 1981.

[Mo] B. M. E. Moret, "Decision Trees and Diagrams," *ACM Computing Surveys,* Vol. 14 (Dec. 1982), pp. 593–623.

[NB] R. Nair and D. Brand, "Construction of Optimal DCVS Trees," IBM Research Report RC-11863, Thomas J. Watson Research Center, March 1986.

[PS] Pitchumani, V. and E. P. Stabler, "An Inductive Assertion Method for Register Transfer Level Design Verification," *IEEE Trans. Comp.*, Vol. C-32, No. 12 (Dec. 1983), pp. 1073-1080.

[SB] Smith, G. L., R. J. Bahnsen and H. Halliwell, "Boolean Comparison of Hardware and Flowcharts," *IBM J. Res. Develop.*, Vol. 26, No. 1 (Jan. 1982), pp. 106-116.

[SF] K. J. Supowit and S. J. Friedman, "A New Method for Verifying Sequential Circuits," *Proc. 23rd Design Automation Conf.* (June 1986), pp. 200–207.

[Ul] Ullman, J. D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

[Wo] Wojcik, A. J., "Formal Design Verification of Digital Systems," *Proc. 20th Design Automation Conf.* (June 1983), pp. 228-234.

[YH] E. J. Yoffa and P. S. Hauge, "ACORN: A Local Customization Approach to DCVS Physical Design," *Proc. 22nd Design Automation Conf.* (June 1985), pp. 32–38.

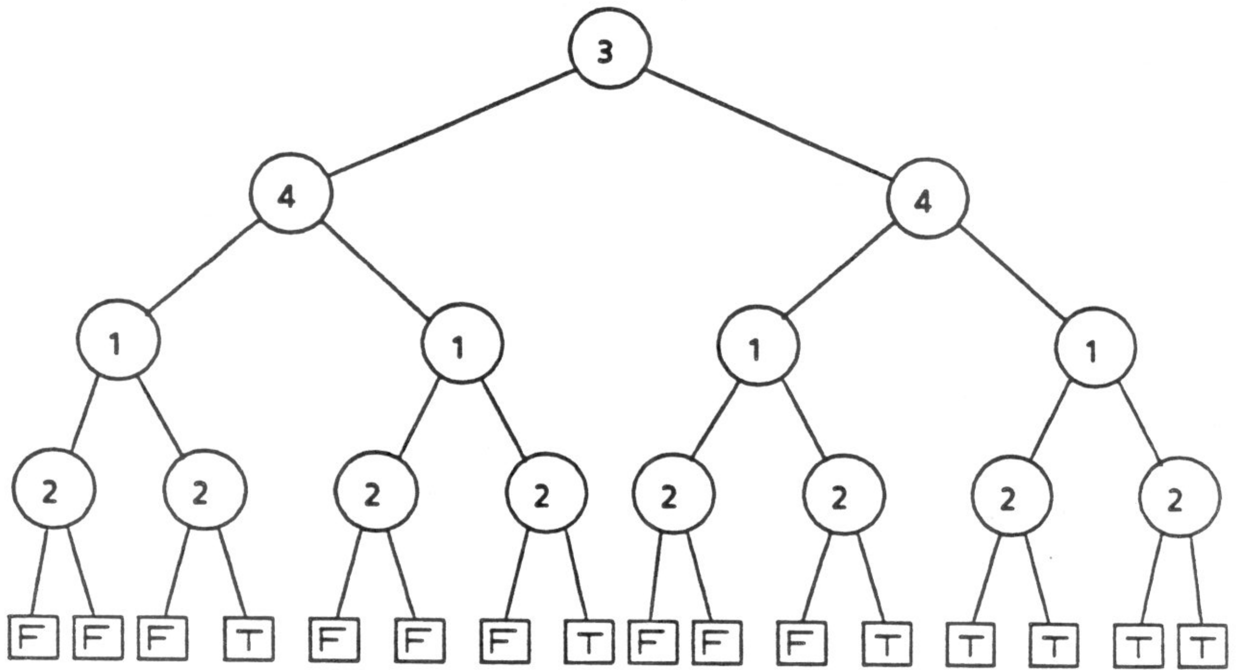


Figure 1: Decision tree representation of $x_1x_2 + x_3x_4$

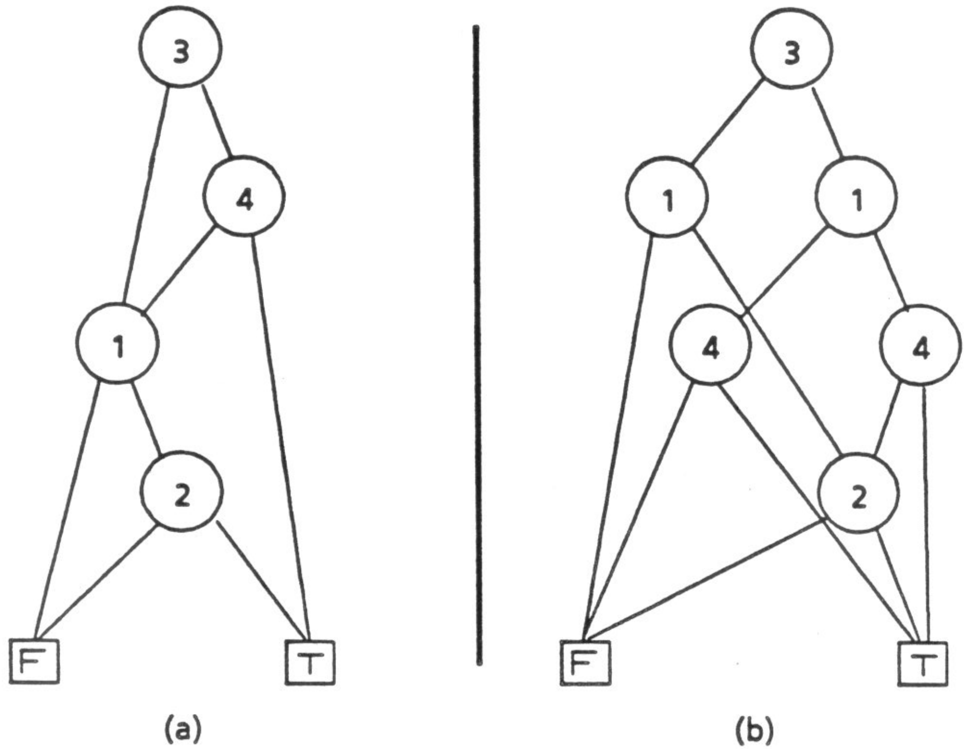


Figure 2: Two possible ordered BDD representations of $x_1x_2 + x_3x_4$

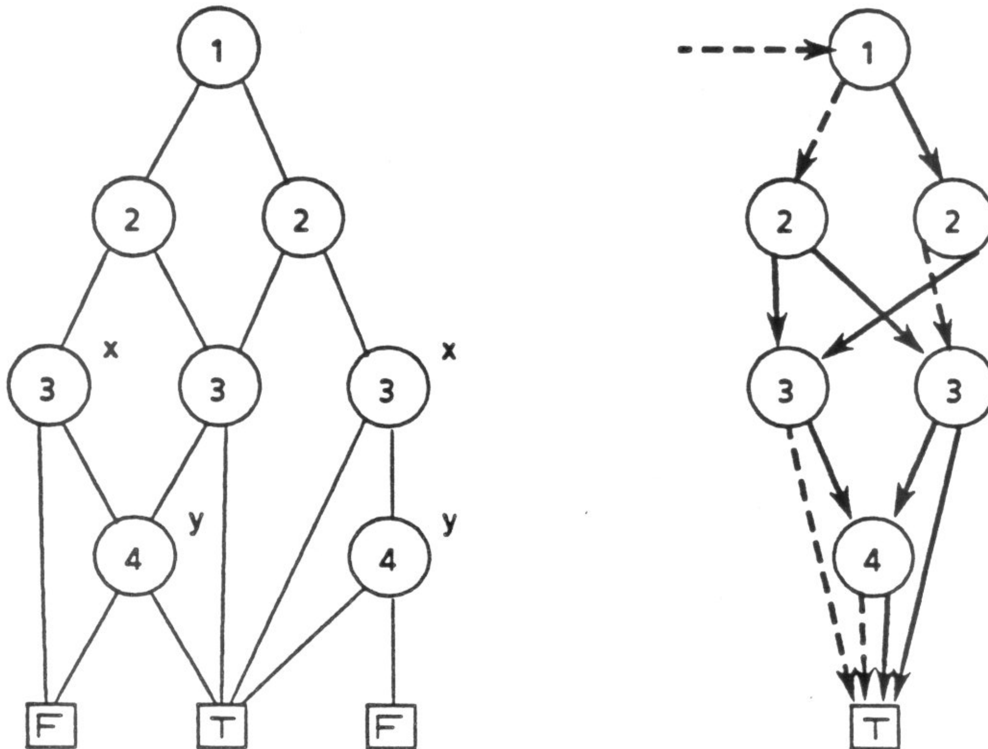


Figure 3: An ordered BDD and its corresponding typed Shannon's canonical form: The dotted arrows point to "negative" functions. Compression results from collapsing together the nodes labelled x and the nodes labelled y.

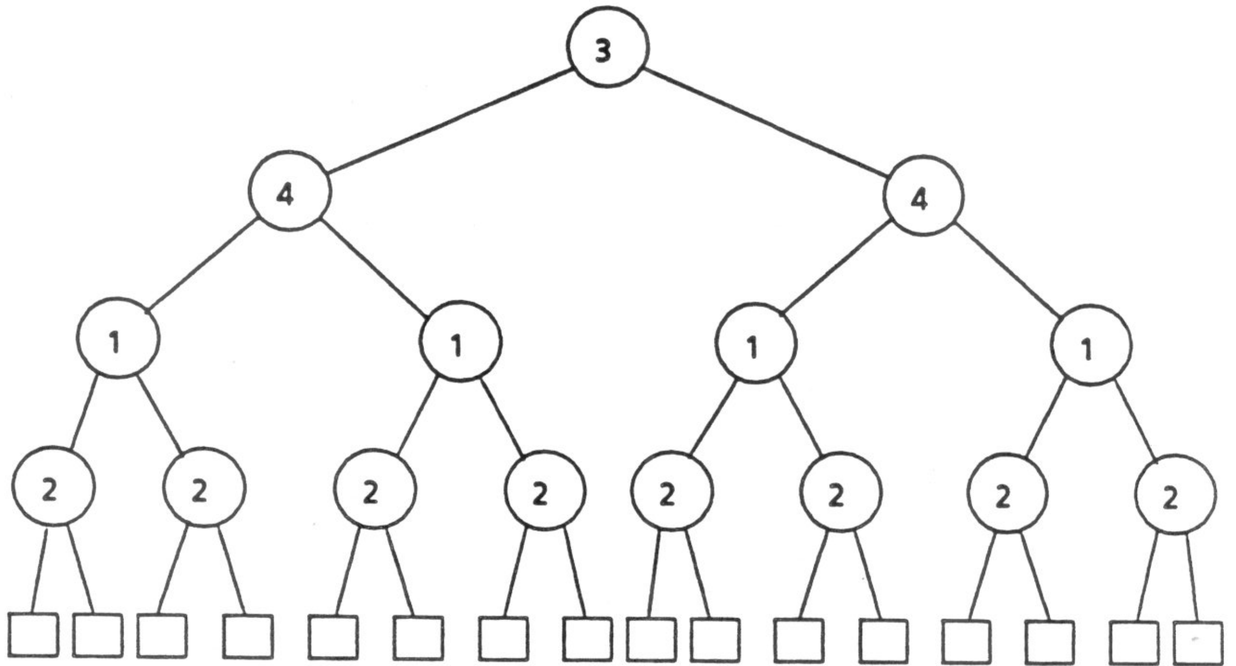
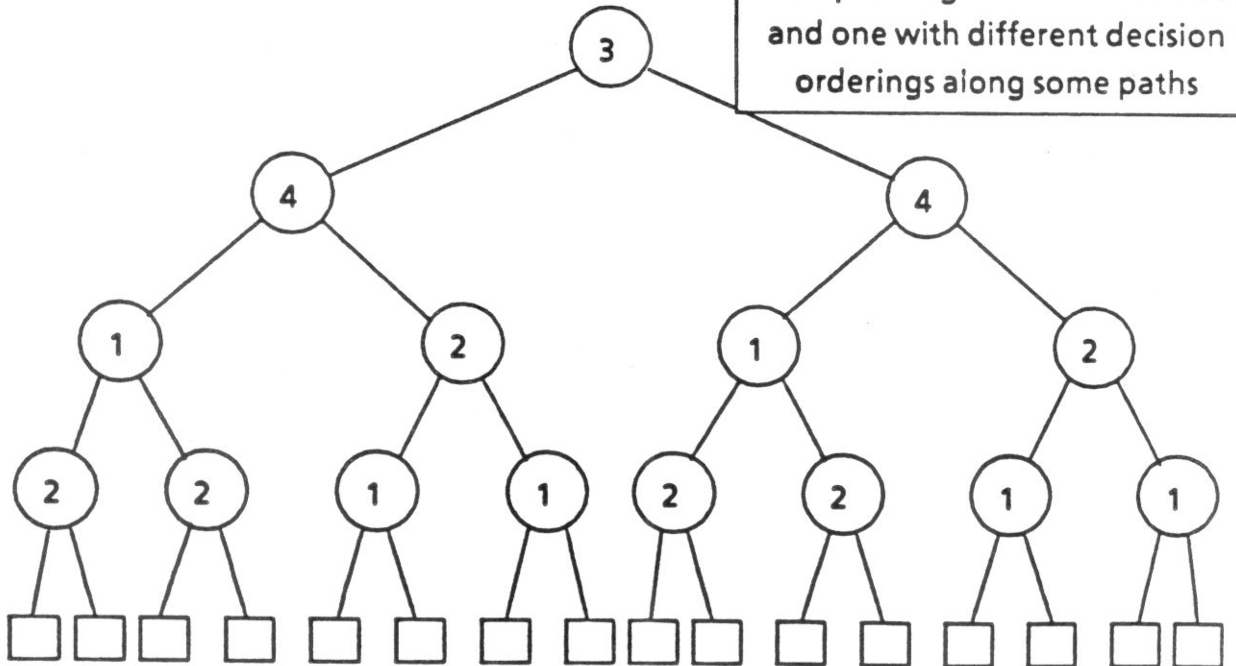


Figure 5: A template corresponding to an ordered BDD, and one with different decision orderings along some paths



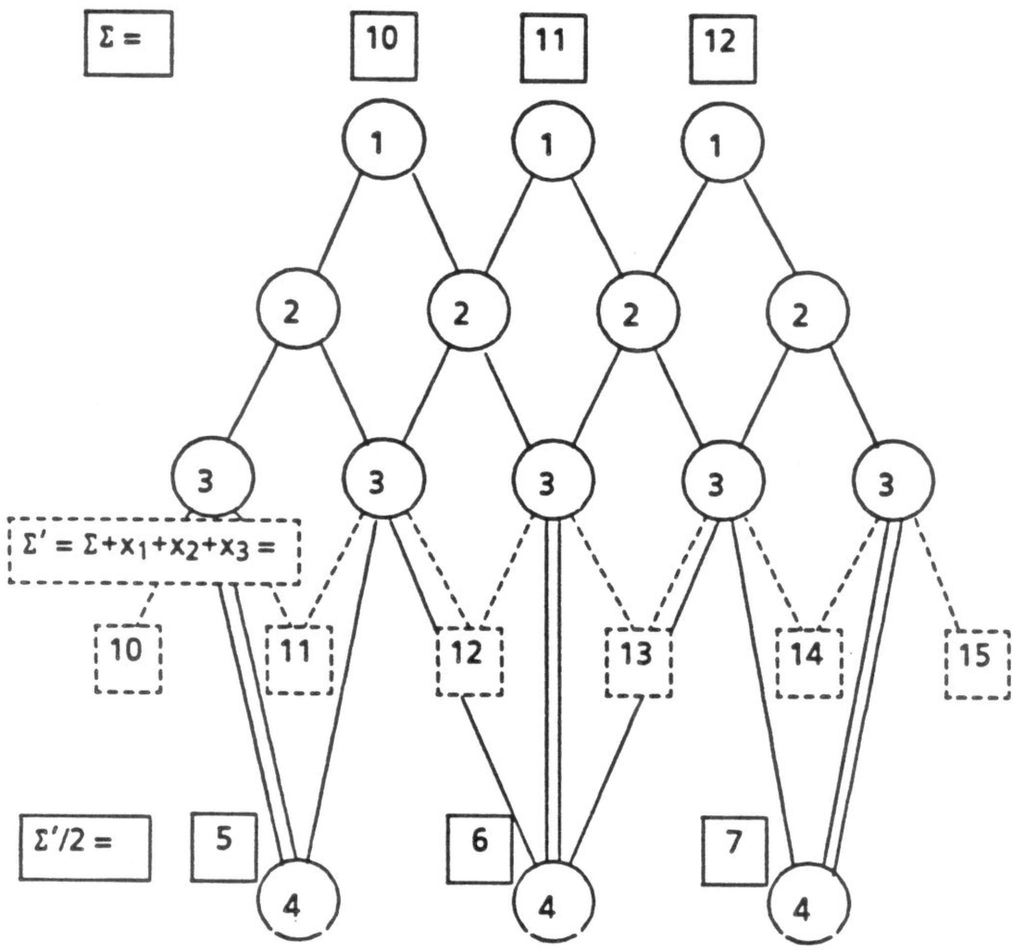


Figure 6: Generalized BDD implementation of a tally, and division by two between stages.

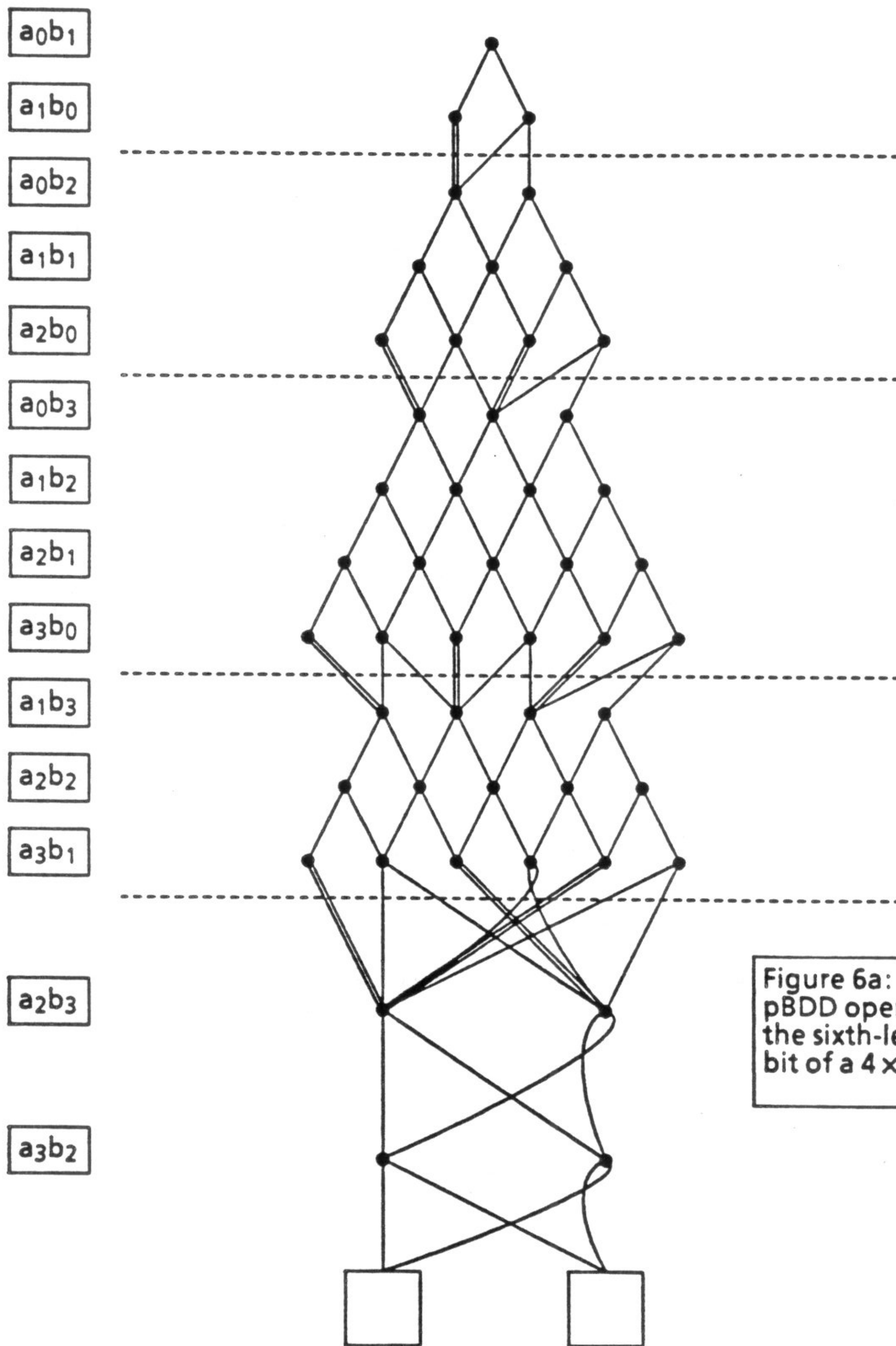
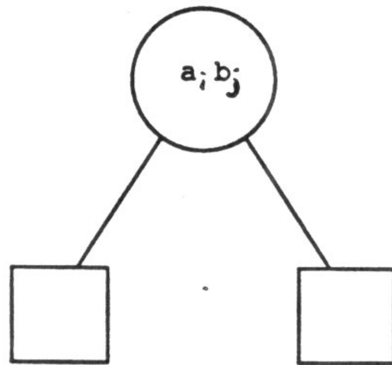
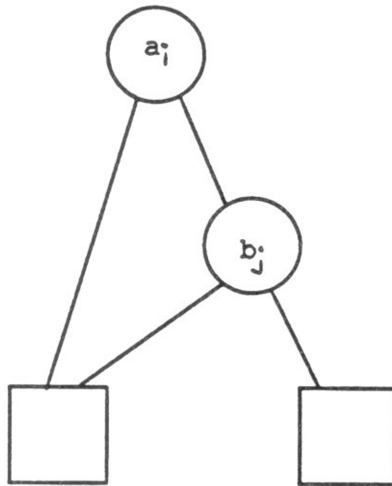


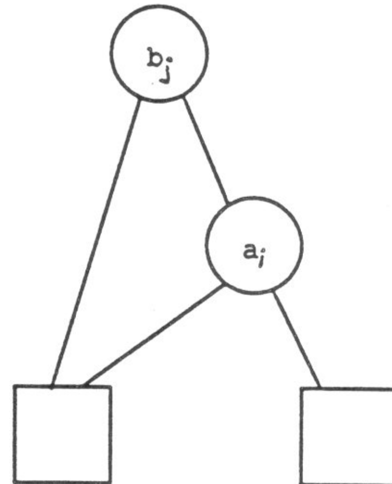
Figure 6a: Template for pBDD operations involving the sixth-least-significant bit of a 4×4-bit multiplier



becomes...



...if i>j (i<j during stage 2)



...otherwise

Figure 7: Expanding product decisions into single-variable decisions while preserving the property of all variables on each path.

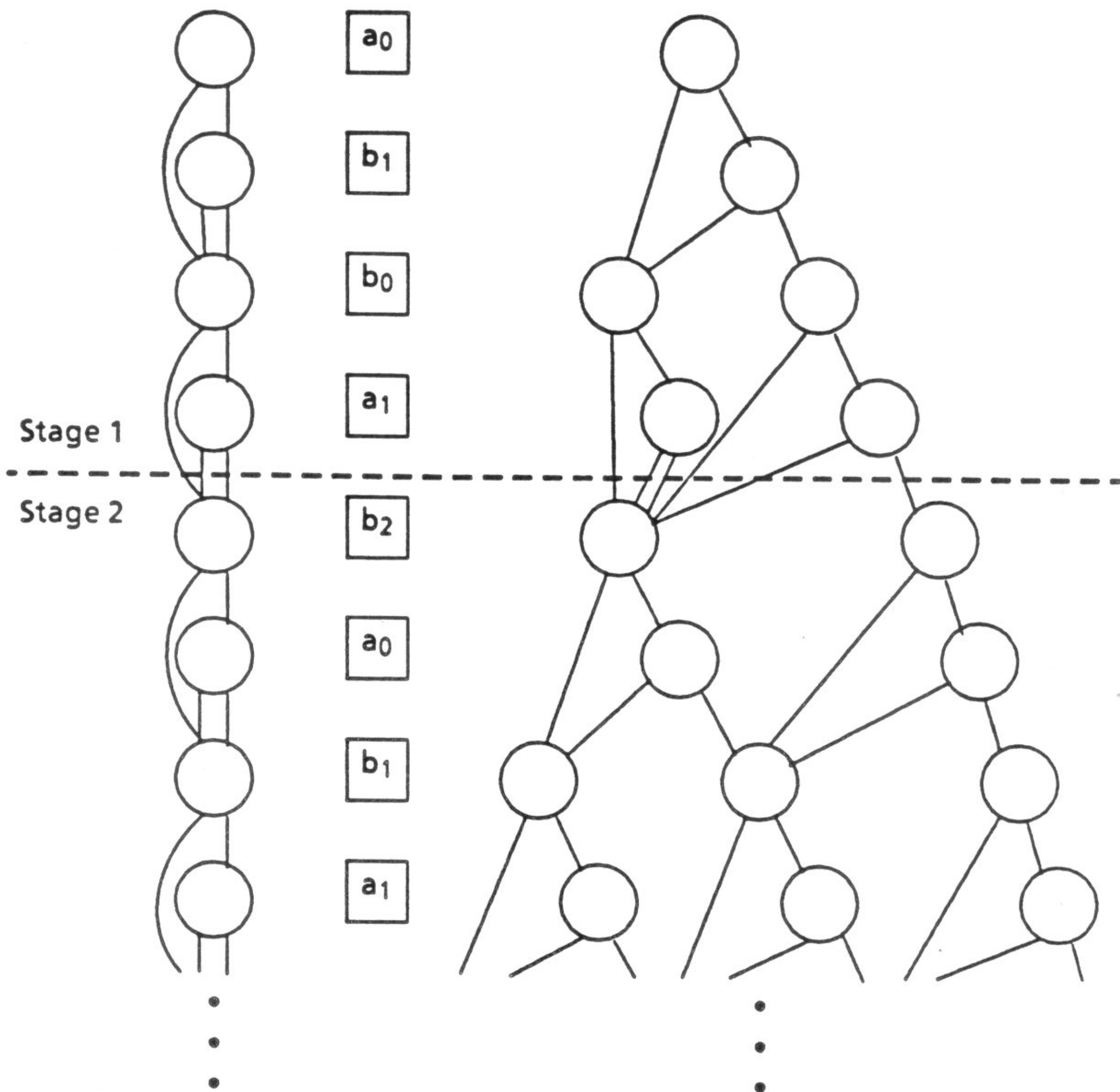


Figure 8: $O(n^2)$ multiplier template vs. $O(n^3)$ template.

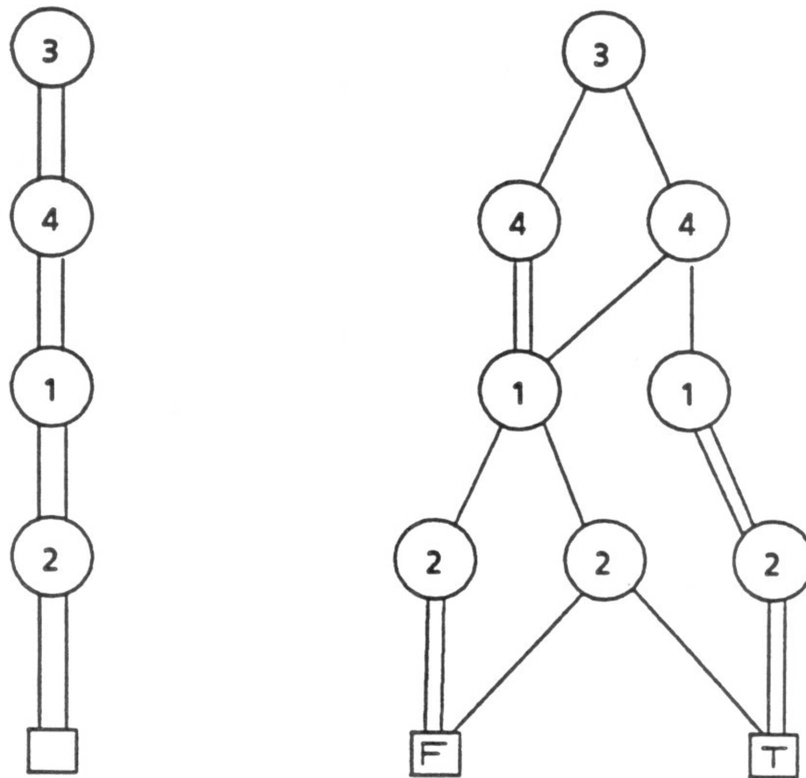


Figure 9: "Ordered BDD" template, and pBDD representing $x_1x_2 + x_3x_4$. Compare with Figure 2(a).

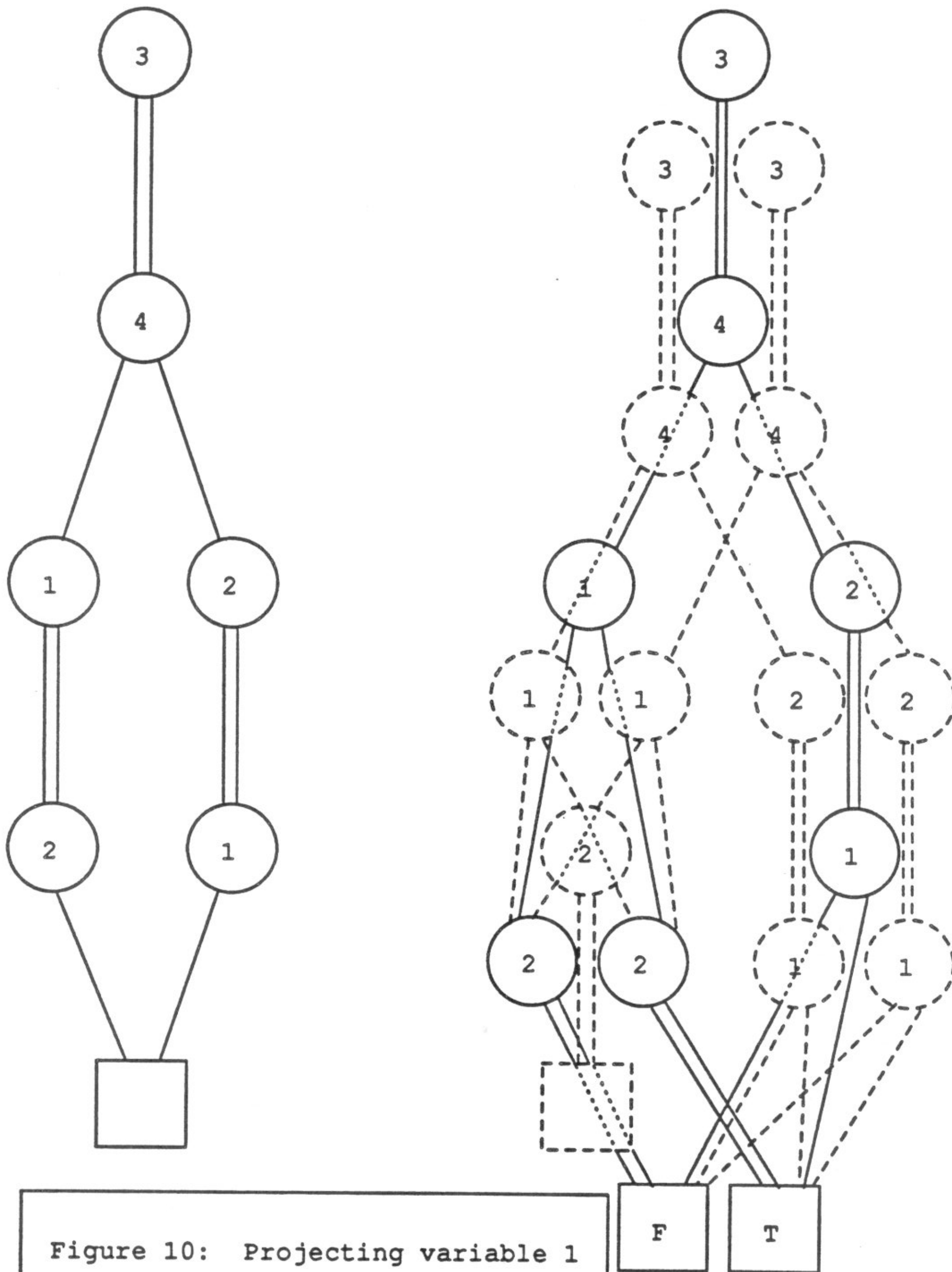
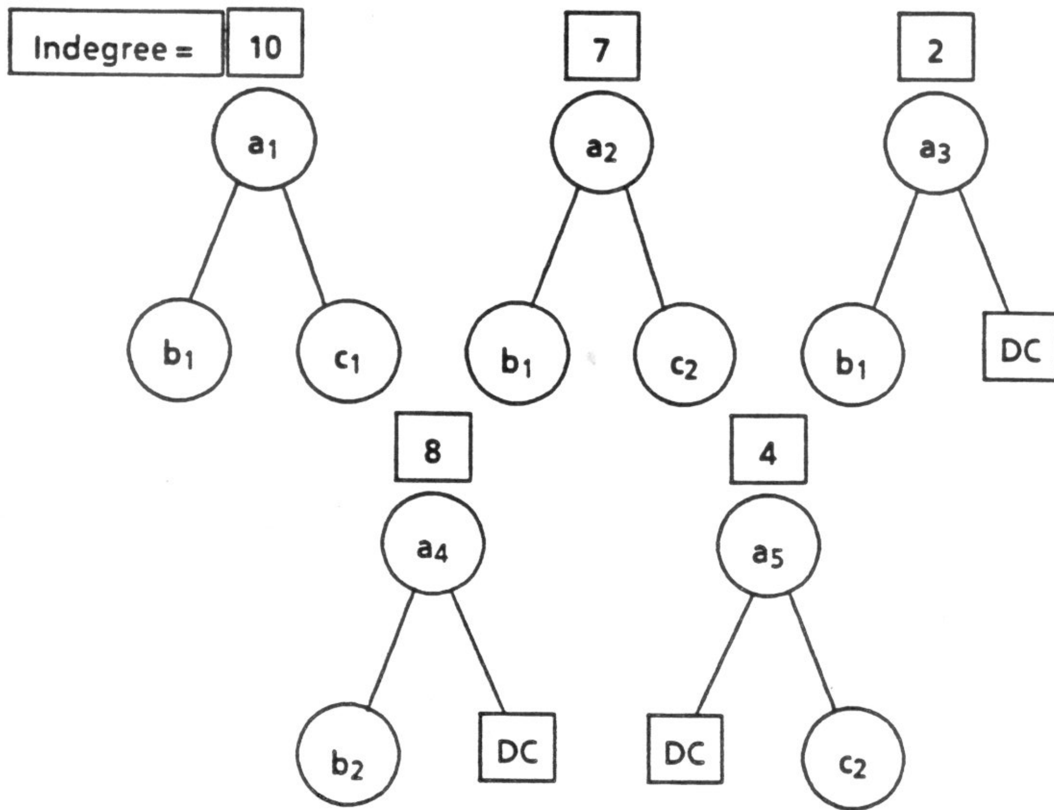


Figure 10: Projecting variable 1 onto the template at left. Dashed nodes and edges are discarded in step 3 of the algorithm in 4.2.1.



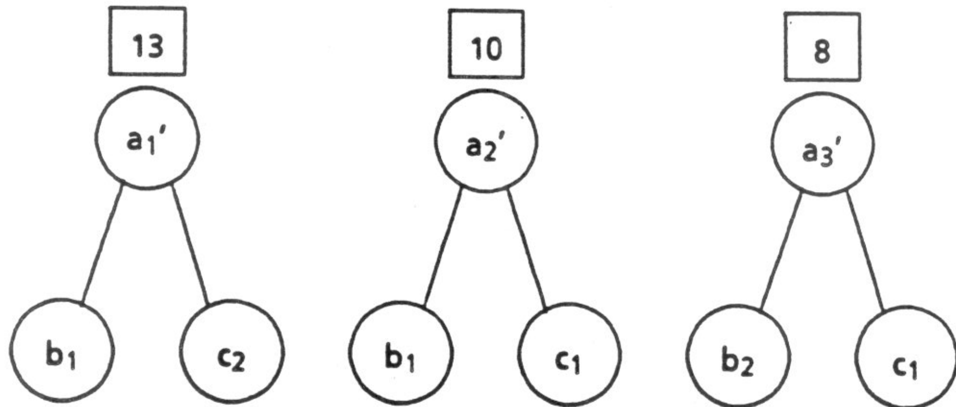
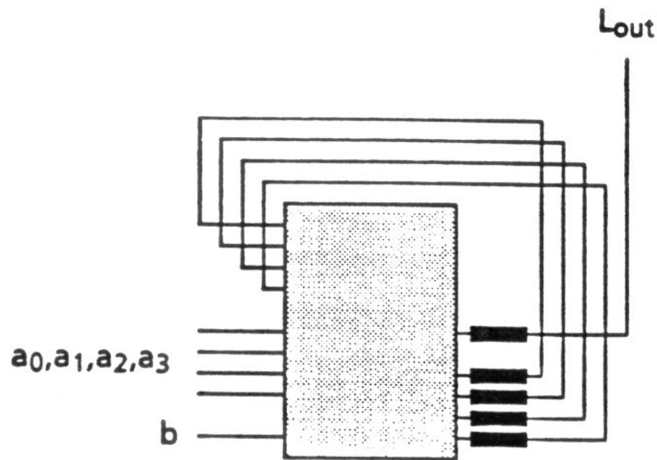


Figure 11: Example of the collapsing heuristic for pBDD nodes.



$I = \{a_0, a_1, a_2, a_3, b\}$

$\Delta = 4$

$I' = \{A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3\}$

$F(a_i, j) = A_i$ for $i = 0,1,2,3,$ $j = 1,2,3,4$

$F(b, j) = B_{j-1}$ for $j = 1,2,3,4$

$out = \langle L_{out}, 2 \rangle$

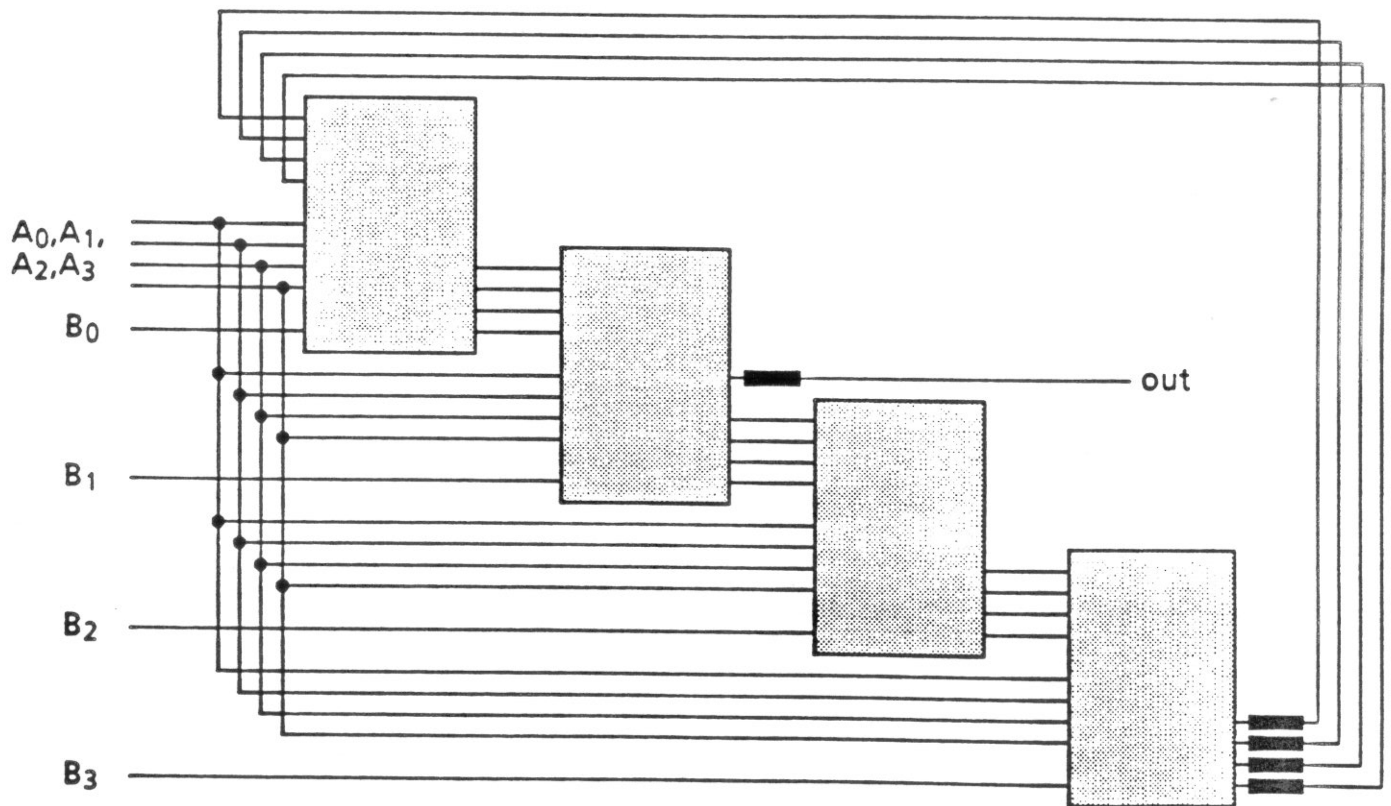


Figure 12: Cascading a 4x4 parallel-serial multiplier so that its entire computation is performed at every clock cycle.

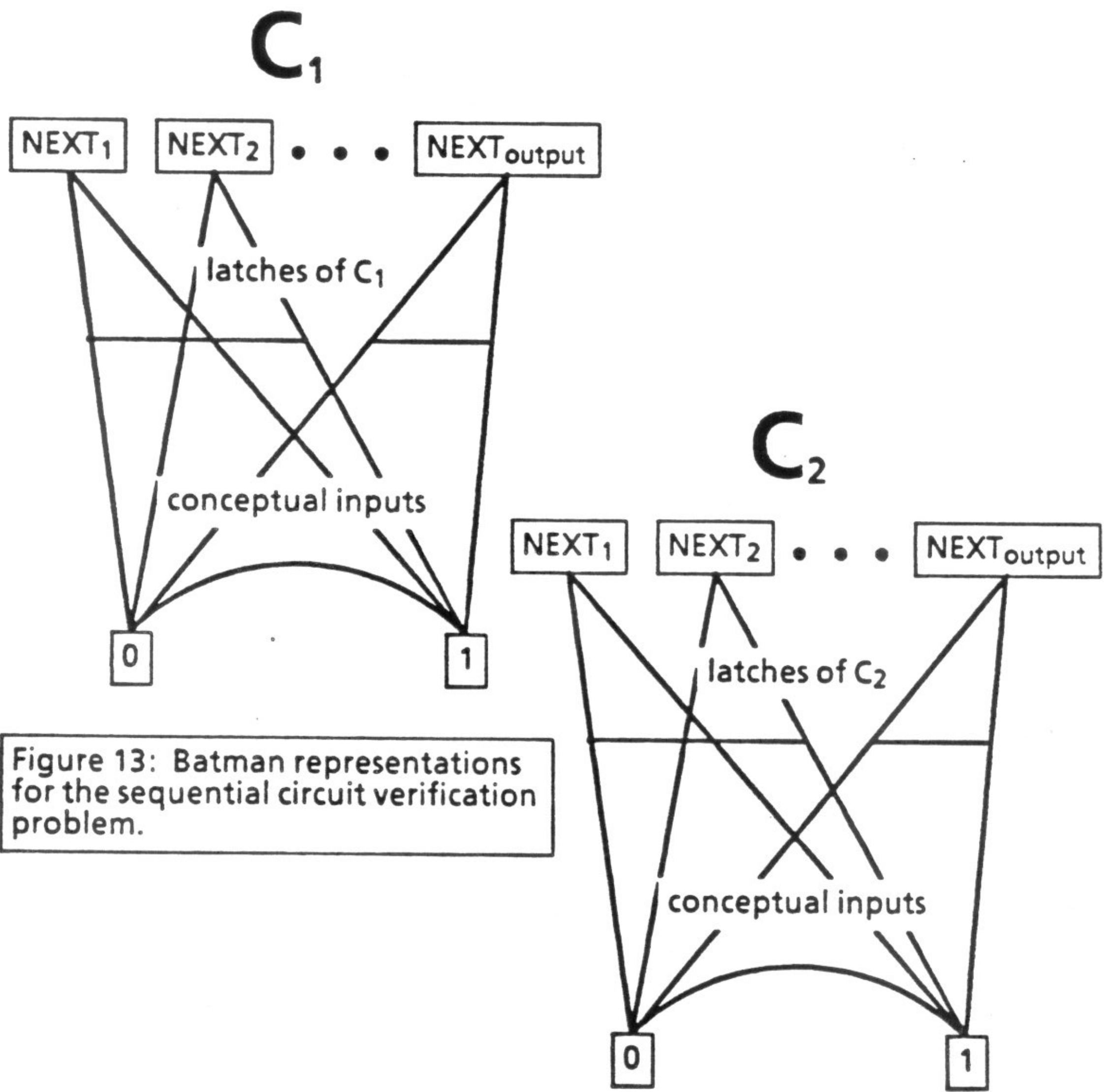
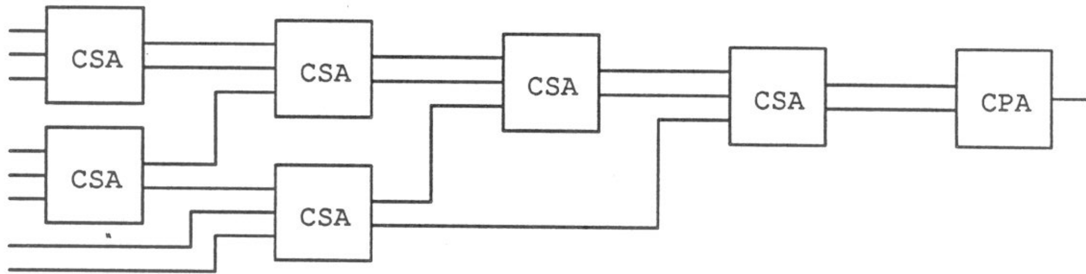
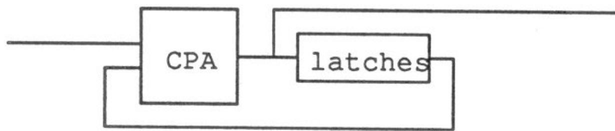
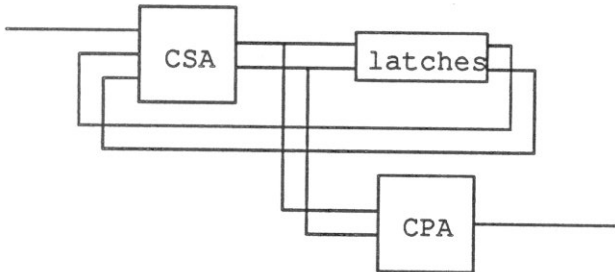


Figure 13: Batman representations for the sequential circuit verification problem.



(a) Combining eight 8x1 multipliers into one 8x8 multiplier





(b) Two accumulator designs

Figure 14: Circuit designs used to test the algorithms