1 Model checking and equivalence checking

Masahiro Fujita

1.1 Introduction

Owing to the advances in semiconductor technology, a large and complex system that has a wide variety of functionalities has been integrated on a single chip. It is called *system-on-a-chip* (SoC) or *system LSI*, since all of the components in an electronics system are built on a single chip. Designs of SoCs are highly complicated and require many manpower-consuming processes. As a result, it has become increasingly difficult to identify all the design bugs in such a large and complex system *before* the chips are fabricated. In current designs, the verification time to check whether or not a design is correct can take 80 percent or more of the overall design time. Therefore, the development of verification techniques in each level of abstraction is indispensable.

Logic simulation is a widely used technique for the verification of a design. It simulates the output values for given input patterns. However, because the quality of simulation results deeply depends on given input patterns, there is a possibility that there exist design bugs that cannot be identified during logic simulation. Because the number of required input patterns is exponentially increased when the size of a design is increased, it is clearly impossible to verify the overall design completely by logic simulation. To solve this problem, the development of formal verification techniques is essential. In formal verification, specification and design are translated into mathematical models. Formal verification techniques verify a design by proving its correctness with mathematical reasoning, and, therefore, they can verify the overall design exhaustively. Since formal verification is a mathematical reasoning process and logic circuits compute Boolean functions, it is realized on top of basic Boolean reasoning techniques, such as binary decision diagrams (BDDs), Boolean satisfiability checking methods (so-called SAT methods), and automatic test-pattern generation techniques (ATPG) for manufacturing test fields. The performance of formal verification methods relies heavily on the performance of these techniques. Figure 1.1 shows an overview of a formal verification flow. In formal verification, both specification and design descriptions are translated into mathematical models using front-end tools. Finite state machines, temporal logic, Boolean functions, and so on, are used as mathematical models. After mathematical models are obtained, they are analyzed

Practical Design Verification, eds. Dhiraj K. Pradhan and Ian G. Harris. Published by Cambridge University Press.

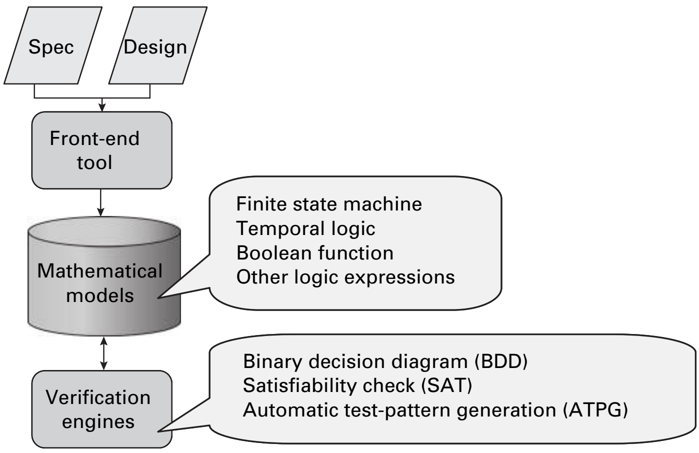

Figure 1.1 Formal verification of design descriptions

using BDD and SAT methods. Formal verification is equivalent to simulating all the cases in logic simulation. If there exists a design bug, formal verification techniques produce a counter-example to support debugging processes.

There are basically two problems in the verification of designs: model checking and equivalence checking. Model checking (or property checking) verifies whether a design satisfies the properties given as its specification. The performance of model checking has drastically improved in recent years, mainly owing to the significant progress of SAT-based efficient implementations. Equivalence checking verifies whether two given designs are equivalent or not. Equivalence checking can be applied to two designs in the same design level or in two different design levels. Depending on the types of equivalence definitions, equivalence checking can be made only on combinational parts of the circuits or on both combinational and sequential parts of the designs. In particular, the former type of equivalence checking has become very practical, and very large designs, such as those with more than 10 million gates, can be formally verified in a couple of hours.

In the actual design flow from highly abstracted design stages down to implementation levels, model checking is applied to each design level to ensure correct functionality, and equivalence checking is applied to any two different design levels so that correctness of the designs can be established. In this chapter, I first briefly review the Boolean reasoning techniques, BDD, SAT, and ATPG methods, in Section 1.2. Property checking and equivalence checking techniques are presented in Sections 1.3 and 1.4 respectively. In Section 1.5, formal verification techniques used in design levels higher than RTL are discussed.

1.2 Techniques for Boolean reasoning

In this section, I introduce three Boolean reasoning techniques, BDD, SAT, and ATPG techniques, which are the bases of formal verification methods. The performance of

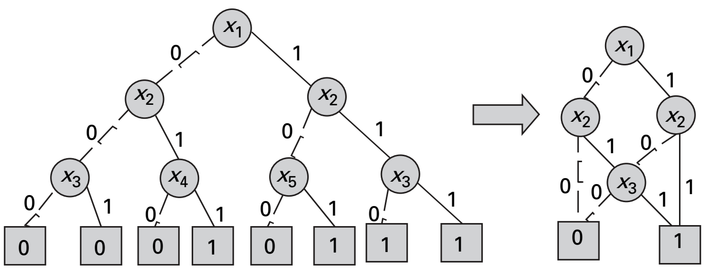

Figure 1.2 A binary decision tree representation of a Boolean function and its corresponding binary decision diagram (BDD)

formal verification methods fully relies on the performance of these techniques. In recent years SAT and ATPG methods, especially their program implementations, have been drastically improved, which make it feasible to verify real-life designs formally within reasonable time.

1.2.1 Binary decision diagrams (BDDs)

Reduced ordered binary decision diagrams (ROBDDs), simply called BDDs, are a canonical representation for Boolean functions. For many Boolean functions of practical interest in VLSI designs, BDDs provide a substantially more compact representation than other traditional alternatives, such as truth tables, sum-of-products (SOP) forms, or conjunctive normal form representations. Further, there exist efficient algorithms to manipulate BDDs. Thus, BDDs and their variants have become widely used in various areas of digital system design, including logic synthesis and formal verification of systems that can be represented in finite state machines. Binary decision diagrams represent the Boolean function as a directed acyclic graph. Let us first consider binary decision trees, an example of which appears on the left-hand side of Fig. 1.2, for the majority function, $f(x_1,x_2,x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3)$. The binary decision tree is a rooted directed tree with two kinds of node, terminal nodes and non-terminal nodes. Each non-terminal node v is labeled with a variable *var(v)* and has two successors, *hi(v)* and *lo(v)*, corresponding to the cases when *var(v)* is set to 1 and 0, respectively. The edge connecting v and *hi(v)*, shown as a solid line (*lo(v)* is shown as a dashed line), is labeled with 1 (0). Each terminal node (leaf node of the tree) is labeled by the Boolean value 0 or 1. Each truth assignment to the variables of the function has a one-to-one correspondence to a path in the tree from the root to a terminal node. This path can be traversed by starting with the root node and taking the edge corresponding to the truth value of the variable labeling the current node. The value labeling the terminal node is the value of the function under this truth assignment. This representation is, however, fairly redundant. For example, the sub-trees corresponding to the assignment ($x_1 = 0$, $x_2 = 1$) and ($x_1 = 1$, $x_2 = 0$) are isomorphic, and the vertex that corresponds to ($x_1 = 0$, $x_2 = 0$) is redundant, since both assignments to x_3 at this point have the same consequence.

A BDD could be obtained for a given Boolean function by essentially placing two restrictions on its binary decision tree representation. The first restriction imposed is a total order $<$ on the variables labeling the vertices, such that for any vertex u in the diagram, if u has a non-terminal successor v, then $var(u) < var(v)$. The second set of restrictions involves merging isomorphic sub-trees and removing redundant vertices by repeatedly applying the following three reduction rules until no further application is possible.

1. *Remove duplicate terminals* Eliminate all but one terminal vertex with a given label and redirect all arcs going to the eliminated vertices into the remaining vertex.
2. *Remove duplicate non-terminals* If two non-terminal vertices u and v have $var(u) = var(v)$, $lo(u) = lo(v)$, and $hi(u) = hi(v)$, then eliminate one of u or v and redirect all incoming arcs to the eliminated vertex to the one that remains.
3. *Remove redundant tests* If a non-terminal vertex v has $hi(v) = lo(v)$, then eliminate v and redirect all its incoming arcs to $hi(v)$.

The resulting representation is a BDD. Figure 1.2 shows an example. The graph on the right-hand side is a BDD corresponding to the binary decision tree of the majority function, shown on the left-hand side in the figure.

Binary decision diagram representations are canonical – that is, two BDDs for a given Boolean function under a given variable ordering are isomorphic. [1] Because of this the equivalence of two Boolean functions can be simply checked by a graph isomorphism check on their respective BDD representations. A function is a tautology if and only if it is isomorphic to the trivial BDD corresponding to a single terminal 1 vertex and satisfiable if and only if it is not isomorphic to the trivial 0 BDD represented by a single 0 terminal vertex. A function is independent of a variable x if and only if there is no vertex labeled with x in its BDD.

The size of a BDD representation is critically dependent on its variable order. Figure 1.3 shows two different BDD representations for the comparator function. The one on the left side uses the ordering $a_1 < a_2 < b_1 < b_2$, while the one on the right uses the order $a_1 < b_1 < a_2 < b_2$. More generally, for an n-bit comparator, the ordering $a_1 < \ldots < a_n < b_1 < \ldots < b_n$ yields a BDD with $3 \cdot 2^n - 1$ vertices, while the ordering $a_1 < b_1 < \ldots < a_n < b_n$ gives a BDD of size $3n + 2$. Thus, the size characteristics of the

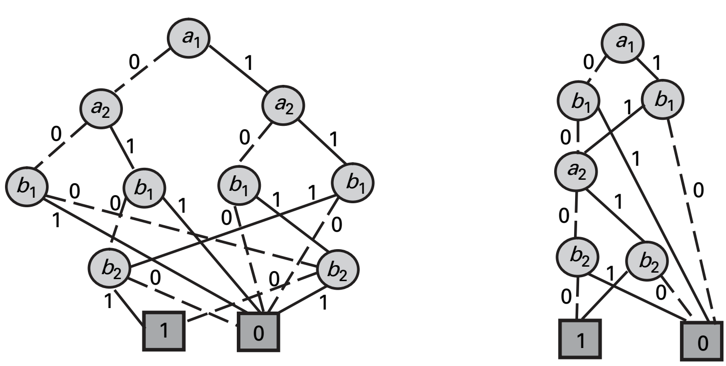

Figure 1.3 An example of how variable ordering can affect the size of an ROBDD

BDD can change from linear asymptotic growth to exponential asymptotic growth by altering the variable ordering strategy. In general, finding the optimal BDD variable order for a given function is a hard problem. Specifically, checking that a given variable order is optimal for a given function is an NP-complete problem. [2] Some classes of Boolean function are particularly difficult cases for BDDs, since any variable order results in a BDD with exponential complexity. The Boolean functions for the middle two outputs of an n-bit integer multiplier are one such example. [3]

The optimal variable order is, however, typically not necessary in order to effectively use BDDs. In practice, we need a variable order that keeps the BDD representations within reasonable limits so that suitable algorithms can manipulate them using the available computer power. In fact, many functions encountered in practical applications do have reasonably compact BDD representations. Moreover, efficient heuristics for BDD variable ordering have been developed that keep BDD sizes in check. One class of variable-ordering heuristics uses domain-specific knowledge to effect a good ordering. For example, if the Boolean function represents a logic gate network, then a depth-first traversal on the network graph can provide a good ordering. [4,5] Another technique, called *dynamic reordering* or *sifting*, [6] is an orthogonal approach, which is used when a domain-specific or constructive ordering algorithm is not available for the functions being manipulated. The technique simply performs a sequence of local reordering moves with the aim of reducing BDD size. It does this on a periodic basis to keep BDD sizes smaller and has often proved to be quite effective in practice.

One operation that is central to the construction, representation, and manipulation of BDDs is the *restriction* or *co-factoring* operation. A restriction or co-factor of f is the function that results when some variable x of f is set to a constant value k (0 or 1), denoted as $f_{x=k}$ or alternatively as f_x for $x = 1$ and $f_{\bar{x}}$ for $x = 0$. Given the two co-factors of a function, it can be expressed using the following identity, known as *Shannon's expansion*: $f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}}$.

The manipulation of BDDs – that is, performing logical operations on functions represented as BDDs – is done using a single universal operation called the *ite (if-then-else)* operator (which internally makes use of the restriction operation). [7] The *ite* operator is a ternary operator, akin in functionality to a multiplexor (mux) in hardware or the *if-then-else* construct available in programming languages. It realizes the function expressed as $ite(f, g, h) = f \cdot g + \overline{f} \cdot h$, where f, g, and h are Boolean functions (possibly non-unique) represented as BDDs. In particular, *ite* can be used to implement any two-variable logic function, such as $f \oplus g = ite(f, \overline{g}, g)$ and $f \geq g = ite(f, 1, \overline{g})$.

Figure 1.4 shows the algorithm used to implement the *ite* operator for BDDs. It is a recursive algorithm where the leaves (terminal cases) of the recursion are degenerate cases of the *ite* operator for which precomputed and stored solutions are substituted, such as $ite(1,f,g) = ite(0,g,f)$ and f $ite(f,g,g) = g$. During the course of the algorithm, the BDD being generated may not remain fully reduced and canonical owing to the addition of new nodes, R. The *reduce*() function in the figure refers to the application of the reduction rules discussed earlier. In practical BDD packages, the need for this

```
ite(f,g,h) {
  if (terminal case) {
    return computed-result;
  } else { // general case
    let v be the top variable of (f,g,h);
    f̃ ← ite(f_v, g_v, h_v)
    f̃ ← ite(f_v, g_v, h_v)
    R = new node labeled by v
    R.hi ← f̃
    R.low ← g̃
    reduce(R)
    return R;
```

Figure 1.4 Algorithm to implement the *ite* operator

reduce() operation is obviated by maintaining hash tables of both unique BDD nodes and previous *ite* calls. New *ite* calls, as well as new BDD nodes (R) created through them, are looked up against these hash tables before initiating new ones, thereby dynamically maintaining and growing a reduced-ordered BDD.

1.2.2 Boolean satisfiability checker

The Boolean satisfiability (SAT) problem is a well-known constraint satisfaction problem, with many applications in the fields of VLSI computer-aided designs and artificial intelligence fields. Given a propositional formula φ, the Boolean satisfiability problem posed on φ is to determine whether there exists a variable assignment under which φ evaluates to *true*. Such an assignment, if one exists, is called a *satisfying assignment* for φ, and φ is called *satisfiable*. Otherwise, φ is said to be *unsatisfiable*. The SAT problem is known to be NP-complete. [8] However, in recent years, there have been tremendous advancements in SAT technology, making SAT solvers a viable option for solving many real-world problems.

Most SAT solvers use a *conjunctive normal form (CNF)* representation of the propositional formula. A CNF formula consists of a conjunction of clauses, each of which is a disjunction of literals, and a literal is a variable or its negation. For example $(a + b + \overline{c})(\overline{a} + c)(a + \overline{b} + c)$ is a propositional formula in CNF over the variables a, b, and c. It is composed of a conjunction of three clauses. The clause $(a + \overline{b} + c)$ is one of the clauses, a disjunction of literals a, $\bar{b}$, and c. Note that for a CNF formula to be satisfied, each of its clauses must be satisfied – that is, evaluate to *true*. There exist polynomial algorithms to transform an arbitrary propositional formula into a satisfiability equivalent CNF formula, which is satisfiable if and only if the original formula is satisfiable.

Most modern SAT solvers are based on the *Davis–Putnam–Logemann–Loveland (DPLL) procedure*. [9,10] The DPLL algorithm essentially performs a

```
sat-solve()
  if preprocess() = CONFLICT then
    return UNSAT
  while TRUE do
    if not decide-next-branch() then
      return SAT;
    while deduce() = CONFLICT do
      blevel ⇐ analyze-conflict();
      if blevel = 0 then
        return UNSAT;
      backtrack (blevel);
    done;
  done;
```

Figure 1.5 A generalized DPLL algorithm

branch-and-bound search over the space of possible Boolean assignments of the variables of the given propositional formula. It is a sound and complete algorithm – that is, it finds a satisfying assignment if and only if the given formula is satisfiable. Figure 1.5 shows the basic processing flow of the DPLL algorithm. This form provides a suitable framework for illustrating the advanced features of modern DPLL-based SAT solvers.

The first operation in the algorithm is a set of preprocessing steps (*preprocess*()) during which it may be discovered that the formula is unsatisfiable. If this is not the case, the algorithm enters the outermost loop, which consists of choosing an unassigned variable and assigning to it a value that has not been explored earlier (*decide-next-branch*()). If no such variable exists, the current partial assignment is a satisfying assignment for the formula. Otherwise, the variable assignments deducible from the current assignments are applied (*deduce*()) using a procedure known as *Boolean constraint propagation (BCP)*. This consists of an iterated application of the *unit clause rule*, which is applied on unit clauses – that is, clauses with all but one literal assigned to false and the last literal unassigned. The unit clause rule asserts the last unassigned literal of each unit clause as true, since the other assignment represents a search path that cannot lead to a satisfying assignment. A conflict occurs when a variable is asserted as true as well as false. If BCP does not lead to a conflict, the *decide-next-branch*() loop is repeated by choosing further unassigned variables and values. However, in the event of a conflict, the search backtracks (*backtrack*()) by undoing a certain number of decisions and their BCP implied assignments, based on an analysis of the conflict by *analyze-conflict*(). If all decisions need to be undone (i.e., the backtrack-level *blevel* is 0), the formula is deemed unsatisfiable, since the entire search space has been exhausted.

The original DPLL algorithm used chronological backtracking – that is, it would backtrack up to the most recent decision, for which the other value of the variable had not been tried. However, modern SAT solvers use *conflict analysis* techniques (shown as (*analyze-conflict*) in the figure) to analyze the reasons for a conflict. Conflict analysis is used to perform *conflict-driven learning* and *conflict-driven backtracking*, which were incorporated independently in the GRASP [11] and rel-sat [12] SAT solvers. Conflict-driven learning consists of adding *conflict clauses* to the formula, to avoid the same conflict in the future. Conflict-driven backtracking allows non-chronological backtracking – that is, up to the closest decision that caused the conflict. These techniques greatly improve the performance of the SAT solver on structured problems. The conflict analysis is realized using *implication graphs*, [11,13] which capture the current state of the SAT solver.

Many other advances have been made in developing the basic components that comprise the DPLL-based SAT solver: the decision engine (heuristics for choosing decision variables and values); the deduction engine (data structures and heuristics for performing BCP and detecting conflicts); and the diagnosis engine (heuristics for conflict-driven learning). [14] An interesting property of CNF representations was first exploited by Zhang in the SATO SAT solver [15] to improve the performance of BCP. It proposed the use of head and tail pointers to point to non-false literals in the list representation of a clause, and maintained the *strong invariant* that all literals before the head pointer, and all literals after the tail pointer, are false. Clearly, detection of a unit clause during BCP becomes easy – that is, when the head and tail pointers coincide on an unassigned literal. The main advantage is that the clause status is updated only when either of the head or tail literals is assigned a false value during BCP. In particular, this eliminates an update when any of the other literals in the clause is assigned a value. When the head or tail literal is assigned a false value during BCP, the associated pointer needs to be moved to another non-false literal, if it exists. This is facilitated by the strong invariant. However, during backtracking, the head or tail pointers may need to be moved back again, to maintain the strong invariant.

A different trade-off was proposed in the Chaff SAT solver. [16] Its BCP scheme, known as *two literal watching with lazy update*, is also based on tracking only two literals per clause during BCP. However, Chaff maintains a *weak invariant*, whereby the two watched literals are required to be non-false, but there is no ordering requirement with respect to other false literals. Again, detection of a unit clause during BCP is easily performed by checking whether both watched pointers coincide, and whether clause updates on assignment to other literals are eliminated.

Most of the modern-day SAT solvers incorporate the advanced techniques for conflict-based learning, branching heuristics, and efficient BCP described above as well as efficient data structures and extremely well-tuned implementations to exploit their algorithmic power fully. With these advancements, SAT solvers can now analyze formulas of up to a million variables and three to four million clauses in a few hours of runtime. Of course, these figures hold for only fairly structured SAT instances derived from certain classes of real-world problems.

1.2.3 Automatic test-pattern generation (ATPG) techniques

Automatic test-pattern generation (ATPG) is the process of generating a suite of test vectors that can be used for the purposes of testing a manufactured circuit for *manufacturing faults*. Manufacturing faults are physical defects introduced into the integrated circuit (IC), during the manufacturing process, which result in its incorrect

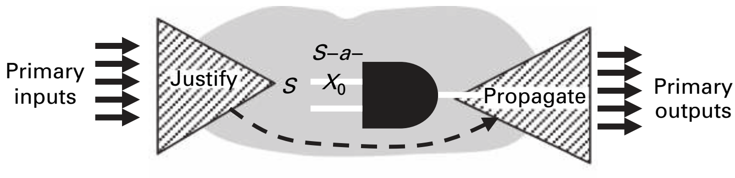

Figure 1.6 ATPG process for a single stuck-at-0 fault

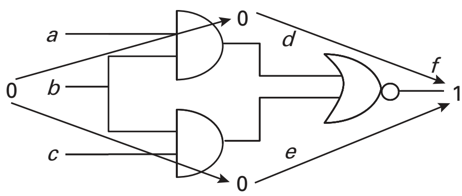

Figure 1.7 An example of implication and learning from circuits

operation. The fault we consider here is one that causes a signal to be permanently stuck at a logical value 0 or 1 (or a defect that can, for all practical purposes, be modeled as such). Such a fault is called a *stuck-at* (0 or 1) *fault*. Very efficient ATPG algorithms for stuck-at faults have been developed, which can be applied to Boolean function reasoning. Therefore, powerful formal verification techniques may be established using ATPG techniques. Thus, the purpose here is to show basic concepts and developments in ATPG so that the link of ATPG to formal verification algorithms becomes evident.

Figure 1.6 illustrates the steps involved in trying to generate a test pattern for a single stuck-at fault. In this example, the signal s is assumed to be under stuck-at-0 fault. To generate a test for s stuck-at-0, we need to find a vector of primary inputs that sets signal s to 1 (justification step) such that some primary output differs between the good circuit and the faulty circuit (propagation step).

As can be seen from the figure, the ATPG problem is basically a sort of SAT problem. We need to reason about the values of signals based on the constraints shown in the figure. Automatic test-pattern generation techniques have, however, their own historical developments rather independent from SAT method. Their algorithms and heuristics are mostly based on logic-circuit structures and properties of logic gates. This means that techniques used in ATPG methods can be used in SAT methods and vice versa.

One of the most important techniques in ATPG to speed up the test pattern generation processes is called "learning." [17,18] As seen in the previous sections, the concept of learning is also utilized in SAT methods to make them much more efficient. Similar efficiency can be achieved in ATPG processes by learning implications of values of signals from the target circuits. Figure 1.7 shows an implication example.

Suppose that input b is 0. Owing to the nature of the AND gate, d and e also become 0. This implies that f is 1. In summary, we have an implication of values that $b = 0$ implies $f = 1$. Please note that this implication process utilizes the functionality of AND and NOR gates. More learning can be made from this by using the law of contraposition, that is, we can also conclude that $f = 0$ implies $b = 1$. As can be seen from the figure, this learned implication is not so obvious. From $f = 1$ we cannot have fixed values for d and e, since what is required from $f = 1$ on d and e is that at least one of d and e must be 1, that is $d = 1$ and $e =$* (don't care) or $d =$* and $e = 1$. There are two possible values for d and e, which means that further reasoning on values of signals is not straightforward. As can be seen from the example, by using the law of contraposition, many more implications can be obtained, which will further enhance the ATPG processes.

As the discussions above on the circuits shown in Fig. 1.7, if $f = 0$, there are several possible cases of values on d and e. As a result, no further simple implication of the values of signals can be made. On the other hand, in both ATPG and SAT methods, reasoning is based on case splitting and backtracking, and knowledge about necessary assignments computed from learning processes is crucial for the number of backtracks which must be performed. Backtracks occur if wrong decisions have been made, i.e., decisions considered wrong if they violate necessary assignments. Hence, it is important to realize that if *all* necessary assignments are known at every stage of the test-pattern generation process (or in general in all Boolean reasoning processes) backtracks can be avoided. Simple learning methods [17,18] cannot identify *all* necessary assignments, based, as they are, on polynomial time-complexity algorithms. The problem of identifying *all* necessary assignments is NP-complete and a method that guarantees identifying all necessary assignments must be exponential in time complexity. One such technique, which can identify all necessary assignments, is "recursive learning". [19] It involves applying learning methods in a recursive way so that even if multiple cases happen when computing implications, all such cases are exhaustively analyzed. For example, let us consider the case of $f = 0$ in the circuit of Fig. 1.7. In this case, there are two cases of values for d and e, i.e., $d = 1$ and $e =$* or $d =$* and $e = 1$. Recursive learning procedures analyze one case at a time and proceed the necessary assignment analysis in a recursive way. The two cases are shown in Figs. 1.8 (a) and (b), respectively. In (a), $d = 1$ implies $a = 1$ and $b = 1$. In (b), $e = 1$ implies $b =1$ and $c = 1$. The important point here is that in both cases $b = 1$. That is, b is always 1. So we can conclude that $f = 0$ implies $b = 1$ without using the law of contraposition. In this simple example, the same implication can also be obtained by applying the law of contraposition to the implication obtained in Fig. 1.7. In general, however, much more learning can be obtained with recursive learning techniques, especially if there are more recursions. The level of recursion is defined as the number

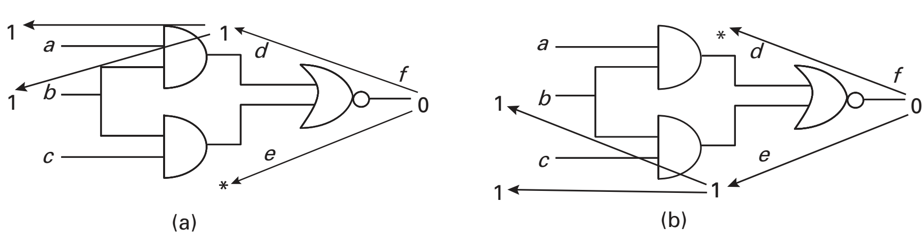

Figure 1.8 Two cases for $f = 0$ in the circuit of Fig. 1.7