1

Preliminaries

1.1 Boolean Circuits

A Boolean variable is a variable whose value can only be either 0 (false) or 1 (true) or unknown. Every Boolean variable has two literals. They are the normal form and the negation/complement of the variable. The negation of a variable always evaluates to the opposite value of the variable. Suppose v is a Boolean variable; then its negation is $\bar{v}$. When v is 1, $\bar{v}$ is 0; when v is 0, $\bar{v}$ is 1. The literals of variable v are then v and $\bar{v}$.

A function consisting of Boolean variables is known as a Boolean function. It is a mapping between Boolean spaces. For example, the function $f : B^m \rightarrow B^n$ is a mapping between the input space of m Boolean variables and the output space of n Boolean variables. We use $f(x_1, x_2, \ldots, x_{m-1}, x_m)$ to indicate the input variables or input values of the Boolean function f.

The mapping between Boolean spaces is achieved by Boolean operators. The basic Boolean operators (operations) AND, OR, NOT, XOR, and XNOR are denoted as $\cdot, +, \sim, \oplus$, and $\bar{\oplus}$, respectively, in this book. We may omit the symbol $\cdot$ for clarity. The behavior of the basic operators is listed in Table 1.1. Complex Boolean operators can be derived from these basic operators. In fact, only AND and NOT, or only OR and NOT, are sufficient to derive all other Boolean operations.

An example of Boolean function is $f(a, b) = a \cdot b$, which computes the logical conjunction of variables a and b. A Boolean function may contain literals. The Boolean function $f(a, b) = a \cdot \bar{b}$ is such an example that computes the logical conjunction of variable a and the negation of variable b. It may be surprising for readers who are not familiar with Boolean algebra to see a function $f(a, b, c) = a \cdot b$. This function is actually nothing special but is normal and valid. It just means that, among the three variables, the value of variable c is "don't care." That is to say, the value of c can be either 0 or 1, and $f(a, b, c) = ab = ab\bar{c} + abc$. For another example, the function $f(a, b, c) = (a + b)$ can be expanded into $f(a, b, c) = (a + b) = (a + b)\bar{c} + (a + b)c$.

Observability don't cares (ODCs) (Damiani and De Micheli 1990) of a Boolean variable are the conditions under which the variable is not affecting any of the primary outputs. For example, if an input i of an AND gate has the controlling value 0, its set of other inputs J are unobservable no matter what values they have. The ODC of J is $\bar{i}$. Satisfiability don't cares

Boolean Circuit Rewiring: Bridging Logical and Physical Designs, First Edition.
Tak-Kei Lam, Wai-Chung Tang, Xing Wei, Yi Diao and David Yu-Liang Wu.

Table 1.1 Behavior of the basic Boolean operators

Operator	When will it returns true?
AND $\cdot$	All of its operands are true
OR $+$	Any one of its operands is true
NOT $\sim$	Its operand is false
XOR $\oplus$	Both of its operands have different values
XNOR $\bar{\oplus}$	Both of its operands have the same values

(SDCs) of a circuit node represent the local input patterns at the node that cannot be generated by the node's fanins. As a trivial example of SDC, if we connect all inputs of a two-input AND gate to a common signal, the values of its inputs can never be $\{1, 0\}$ or $\{0, 1\}$.

Many rules in ordinary algebra, such as commutative addition and multiplication, associative addition and multiplication, and variable distribution, can be applied into Boolean algebra. Therefore, function $f(a, b, c) = (a + b) = (a + b)\bar{c} + (a + b)c = a\bar{c} + b\bar{c} + ac + bc$. For each of the conjunction term, it can be expanded by connecting it with all combinations of the literals of the missing variables by conjunction. Some additional important rules that are obeyed in Boolean algebra only include $a \cdot a = a$ and $a + a = a$. Regarding our example, it can be expanded as follows:

$$\begin{aligned} f(a, b, c) &= (a + b) \\ &= (a + b)\bar{c} + (a + b)c \\ &= (a\bar{c} + b\bar{c}) + (ac + bc) \\ &= (ab\bar{c} + a\bar{b}\bar{c} + ab\bar{c} + \bar{a}b\bar{c}) + (abc + a\bar{b}c + abc + \bar{a}bc) \\ &= ab\bar{c} + a\bar{b}\bar{c} + \bar{a}b\bar{c} + abc + a\bar{b}c + \bar{a}bc \end{aligned}$$

Other rules can be derived from the basic rules easily. Since Boolean algebra is a vast area of study, even the elementary topics can cover a whole book. In this book, we shall not cover every detail.

Boolean functions can be realized in hardware using logic gates. A Boolean circuit or Boolean network is composed of gates and implements some Boolean functions. We simply use circuits or networks to represent Boolean circuits when the meaning is clear in the context. Figure 1.1 illustrates the logic gates implementing the basic Boolean functions. An example circuit composed of AND, OR, and NOT gates is shown in Figure 1.2(b). For gate e, its inputs are a and b, so it is implementing the function $a + b$. Regarding gate f, its function is $a \cdot \bar{b}$.

A less famous Boolean operator is the cofactor. The cofactor of a Boolean function $f(x_1, x_2, \ldots, x_{n-1}, x_n)$ with respect to a variable x_k is $f|_{x_k} = f(x_1, x_2, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_{n-1}, x_n)$. Suppose $f = ab + c$, $f|_a = b + c$, and $f|_c = ab + 1 = 1$. Similarly, the cofactor of a Boolean function $f(x_1, x_2, \ldots, x_{n-1}, x_n)$ with respect to the complement of a variable x_k is $f|_{\overline{x_k}} = f(x_1, x_2, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_{n-1}, x_n)$. Suppose $f = ab + c$, $f|_{\bar{a}} = 0 \cdot b + c = c$. Every Boolean function can be expressed using Shannon's expansion. For example, $f(x) = x \cdot f|_x + \bar{x} \cdot f|_{\bar{x}}$. An example of a function with multiple inputs is $f(x, y, z, w) = xyf|_{xy} + \bar{x}yf|_{\bar{x}y} + x\bar{y}f|_{x\bar{y}} + \bar{x}\bar{y}f|_{\bar{x}\bar{y}}$

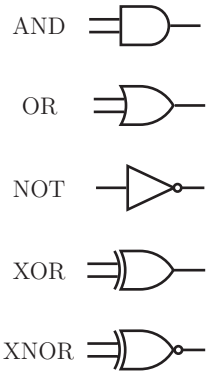

Figure 1.1 Basic logic gates

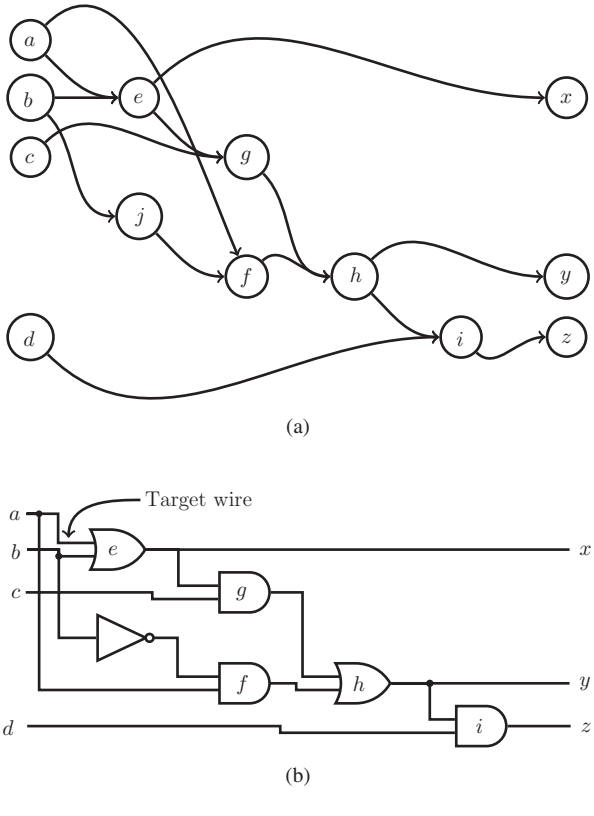

Figure 1.2 Directed acyclic graph representation of a circuit. (a) A directed acyclic graph; (b) a Boolean circuit

In general, Boolean circuits can be represented by directed acyclic graphs (DAGs). A DAG is a kind of graph in which two vertices may be connected by an edge pointing to either one of the vertices, such that there are no loops in the graph. With regard to a circuit, the vertices of its corresponding graph represent its gates, and the edges of the graph represent its wires. An example of a DAG is illustrated in Figure 1.2(a). It is in fact the corresponding graph representation for the circuit in Figure 1.2(b). For instance, the node e in the graph represents the OR gate e, and the edge from a to e corresponds to the target wire indicated in the figure.

Each node in a DAG is therefore associated with a Boolean function. Edges pointing toward a node are known as the fanins of the node, and edges leaving the node are known as the fanouts of the node. The source of a wire connecting s to d has a fanout d, and the destination of the wire d has a fanin s. The number of fanins of a node n is called the in-degree/fanin number of the node, which is denoted by $d^+(n)$. Similarly, the number of fanouts of node n is called the out-degree/fanout number and is denoted by $d^-(n)$. A special case of a DAG is the and-inverter graph (AIG) in which every node represents an AND gate. The values of the edges may be complemented to implement all Boolean functions.

In a circuit, the nodes whose fanin numbers are 0 or have no fanins are known as primary inputs (PIs), and the nodes whose fanout numbers are 0 or have no fanouts are known as primary outputs (POs).

Node a is a transitive fanin (TFI) of node b if there is a path from node a to node b. Node b is said to be a transitive fanout (TFO) of node a if is reachable from node a. All TFIs of a node form the TFI cone of the node, and all TFOs of a node form the TFO cone of the node.

The most famous problem regarding Boolean circuits is the Boolean satisfiability problem. It is always known as the SAT problem. This problem is to determine whether there is any assignment of values to the variables of a given Boolean function such that the function can be evaluated to 1. For the Boolean function $f(a, b, c) = a \cdot b + c$, the value assignments $\{a = 1, b = 1, c = 0\}$ allows f to evaluate to 1. It is a solution that satisfies this SAT problem instance.

1.2 Redundancy and Stuck-at Faults

Central to rewiring techniques is the concept of logic redundancy. Logic redundancy means that, in a circuit, the logic value of a connection or a component has no effect on the output values of the circuit. The occurrence of logic redundancy is usually due to some design errors or faults such as manual design errors, shorting of two wires, or fabrication defects in the circuit. In practice, this kind of redundancy is always targeted for removal so as to minimize the chip area and to improve the yield of the chip. However, there are other kinds of redundancy that are added into circuits intentionally, for example, for fault tolerance.

Engineers have developed the stuck-at fault model to manipulate logic redundancy. As a matter of fact, many types of faults in Boolean circuits can be modeled as stuck-at faults (Jha and Kundu 1990). In this model, the value of a faulty wire is stuck at a constant, either 0 ($sa0$) or 1 ($sa1$). The input vectors that can test a fault are the test vectors of the fault. If there is no test vector for a fault, the fault is regarded as untestable. Testing an untestable fault always results in value assignment conflicts at some locations. Regarding an AND (OR) gate input wire, it is sufficient to certify the untestability of its stuck-at-1 (stuck-at-0) fault to ensure that it is redundant. Its stuck-at-1 (stuck-at-0) is called a noncontrolling-value stuck-at fault, otherwise called a controlling-value stuck-at fault. The names come from the fact that the

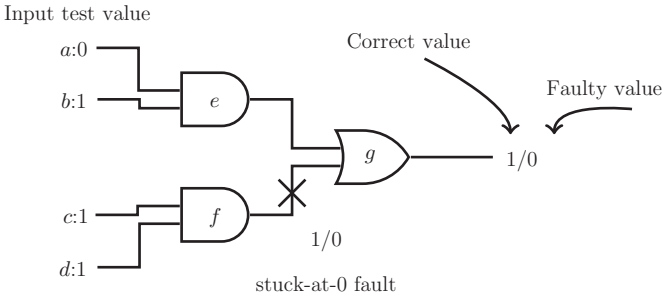

Figure 1.3 Stuck-at fault

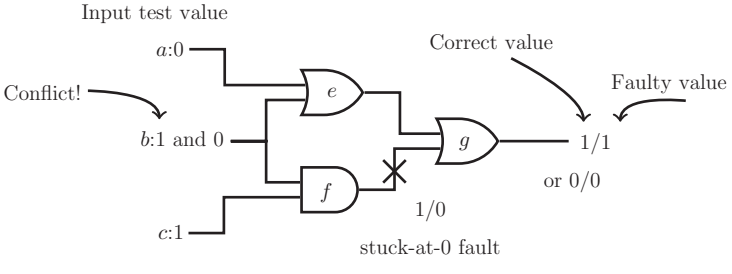

Figure 1.4 Untestable stuck-at fault

controlling value of an AND (OR) gate is 0 (1). In general, a gate's input wire is redundant if its noncontrolling-value stuck-at fault cannot be tested. For an input of a gate that has no noncontrolling value (e.g., XOR), it is redundant only if both its noncontrolling-value and controlling-value stuck-at faults are untestable.

In this book, the notation $sa0(s \to d)$ $(sa1(s \to d))$ will be used to denote the stuck-at-0 (stuck-at-1) of the wire $s \to d$ whose source node (source) is s and destination node (sink) is d. The D-notation is also adopted to denote stuck-at-0 as $1/0(D)$ and stuck-at-1 as $0/1(\bar{D})$.

Figure 1.3 shows a stuck-at fault. The notation $1/0$ on the wire means that the normal or correct value of the wire is 1 and is 0 when a fault occurs. To observe such a fault, we need to find a path in the circuit so that the faulty value can be propagated to at least one primary output and the difference between the normal and faulty values can be observed by the users. The value assignment $\{a = 0, b = 1, c = 1, d = 1\}$ is called a test vector for propagating and observing the $1/0$ fault in the figure. If such a path does not exist for a stuck-at fault, then we can say that there is no test vector for the fault and the fault is undetectable. The wire is thus redundant and can be removed without changing the behavior of the circuit.

An untestable stuck-at fault is shown in Figure 1.4. The value of the input b has to be set to 1 because we want to assign 1 as the correct value of the stuck-at-0 fault. On the other hand, it has to be set to 0 simultaneously so as to make the fault observable. This is obviously impossible. If b is either 1 or 0, the correct and faulty values at the circuit's output are either $1/1$ or $0/0$. This means the stuck-at-0 fault cannot be tested at all. From this analysis, we know that the corresponding wire $f \to g$ is redundant.

1.3 Automatic Test Pattern Generation (ATPG)

Automatic Test Pattern Generation (ATPG), as its name implies, is a circuit testing method and is commonly applied to prove the testability of the stuck-at faults in a given circuit. With regard to a stuck-at fault, it works by feeding the circuit with the essential input test vectors and comparing the output values obtained with those of a fault-free circuit. If there is any difference, the fault is observable.

There are two steps in ATPG, namely (i) fault excitation (activation) and (ii) fault propagation. Fault excitation for a wire's stuck-at fault is to derive a test vector such that the wire can have distinguishable correct and faulty values. Regarding the stuck-at-0 fault in Figure 1.3, the output of gate f has to be 1 to differentiate the correct and faulty values at wire $f \rightarrow g$. Then, the values of both c and d have to be 1 because an AND gate can only output 1 when all of its inputs are 1. The partial test vector is $\{c = 1, d = 1\}$.

Fault propagation means propagating a fault to some outputs of the circuit. If the upper input of the OR gate g (wire $e \rightarrow g$) is assigned with 1, the circuit's output (g's output) will be $1/1$ and the fault is not observable. Hence, wire $e \rightarrow g$ should be assigned with 0 instead. The value of wire $e \rightarrow g$ will be 0 when one of the inputs of gate e is 0. Therefore, all the test vectors for the stuck-at-0 fault in this example are as follows: (i) $\{a = 0, b = 0, c = 1, d = 1\}$ (ii) $\{a = 0, b = 1, c = 1, d = 1\}$ (iii) $\{a = 1, b = 0, c = 1, d = 1\}$. Alternatively, the test vectors can be represented by (i) $\{a = *, b = 0, c = 1, d = 1\}$ (ii) $\{a = 0, b = *, c = 1, d = 1\}$, where $*$ means that the value can be either 0 or 1.

1.4 Dominators

A dominator (Kirkland and Mercer 1987) of a wire w is a gate through which all paths from w to any primary outputs must go. For instance, in Figure 1.4, the set of dominators of b is g because the paths from b to all the circuit outputs (i) $b \rightarrow e \rightarrow g \rightarrow$ (ii) $b \rightarrow f \rightarrow g \rightarrow$ all pass through g. Side inputs of a dominator with respect to a wire w are its inputs that are not lying in w's fault propagation paths. In Figure 1.4, the upper input ($e \rightarrow g$) of the dominator g with respect to the stuck-at fault $sa0(f \rightarrow g)$ is a side input. The gates e and f are not the dominators of $sa0(f \rightarrow g)$.

Algorithm 1.1: Procedure *Cal_Dominators*

```
   input : a circuit C
1  begin
2      N ← primary inputs of C;
3      foreach node n ∈ N do
4          D_n ← n ;          /* D_n is the set of dominators of n */
5          S ← ∅;
6          foreach fanout f_n of n do
7              S ← S ∩ Cal_Dominators(f_n);
8          D_n ← D_n ∪ S;
9  end
```

By definition, a gate is a dominator of itself because it must pass through itself. Every fanout of the gate has its own set of dominators. The common set of the dominators of the gate's fanouts represents the gates through which all fanouts must pass. Hence, the dominators of all the gates in a circuit can be calculated recursively. The recursive algorithm is listed in Algorithm 1.1.

For a given fault, determining all test vectors to detect the fault is proven to be extremely complex. The notion of dominators is always applied in fault propagation so that the process is more systematic and efficient. In fault propagation, the side inputs of each of the dominators with respect to a fault are assigned with the noncontrolling value of the dominator. This is how $e \rightarrow g$ is assigned with 0 in the example illustrated in Figure 1.4. Further logic implications are then carried out based on the assigned values.

1.5 Mandatory Assignments and Recursive Learning

The fundamental component of ATPG is the logic implication procedure during fault excitation and propagation. Given a Boolean network and a set of preassigned values on some of the gates, logic implication is to derive the values of unassigned gates such that they are the same for any given input to the network that results in the preassigned values. Referring to the example in Figure 1.3, gate e is the first gate whose output is preassigned with a value (1). The unassigned gates are then determined by evaluating the truth tables of the gates with assigned values and the connectivity of the circuit.

The simplest form of logic implication is called direct implication, which is simply to evaluate the truth tables of the gates with assigned values to determine unknown values. However, it is not always possible to derive unknown values by truth table evaluation. For example, when the output of an AND gate is 0, which of its inputs should have the value 0? The problem of logic implication is actually much more complex than one would imagine.

The set of values on the unassigned gates found by the implication procedure is called the mandatory assignments (MAs) (or necessary assignments). A mandatory assignment is a

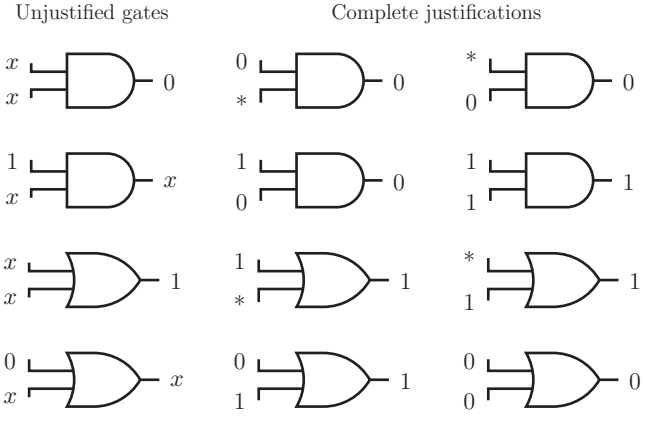

Figure 1.5 Unjustified AND/OR gates and their complete justifications

"forced mandatory assignment" (FMA) if it turns an originally testable fault to be untestable when its opposite value is used instead. With reference to Figure 1.3, the mandatory assignments for testing stuck-at-0 fault on wire $f \rightarrow g$ are $\{a = 0, b = 1, c = 1, d = 1, e = 0, f = 1\}$. Among these mandatory assignments, the mandatory assignment $b = 1$ is not an FMA because the fault $sa0(f \rightarrow g)$ is still testable even if $b = 0$ is used.

Recursive learning (Kunz and Pradhan 1994) or indirect implication is a method to derive all mandatory assignments for a given configuration of value assignments on the Boolean network. It is based on propagating the values found by direct implication according to the structural information of the circuit. The following gives several definitions that are needed in the recursive learning method:

1. Given a gate g that has at least one specified input or output signal, g is unjustified if there are one or more unspecified input or output signals of g for which it is possible to find a combination of value assignments that yield a conflict at g. Otherwise, g is justified. Figure 1.5 shows all possible assignments on a two-input AND gate and a two-input OR gate that will result in the gate becoming unjustified. In the figure, x represents unknown values and $*$ represents either 0 or 1.
2. A set of signal assignments $J = \{f_1 = V_1, f_2 = V_2, \ldots, f_n = V_n\}$, where $f_1, \ldots, f_n$ are the unspecified input or output signals of an unjustified gate g, is called the justifications for g if the combination of value assignments in J makes g justified.
3. Let C^* be the set of all justifications for gate g. The set of complete justification C for g is defined to be $C = \{J_i | J_i \subseteq J^*, \forall J^* \in C^*\}$.

An example of recursive learning in action is illustrated in Figure 1.6. Suppose the output of gate $g3$ is assigned with 1 (Figure 1.6(a)). After propagation of this value and justification of the assigned values, there can be three outcomes as shown in Figure 1.6(b)–(d). The common value assignment among the three configurations is $\{b = 1\}$. As a result, we can conclude that $g3 = 1$ implies $b = 1$ (Figure 1.6(e)). If only direct implication is applied and without recursive learning, this result cannot be obtained.

The level of logic implication is deeper through the use of recursive learning. As we shall see in later chapters, a deeper logic implication level is beneficial to rewiring techniques in that it allows them to identify more alternative wires. This has been proven by experimental results. The tradeoff of adopting recursive learning in rewiring algorithms is an increase in the runtime.

1.6 Graph Theory and Boolean Circuits

Definition 1.1 *Given a Boolean network, a cut-set (or cut for brevity) of a given node set O_v is a set of nodes not in O_v and are directly connected to the boundary nodes of O_v.*

It can be understood that any path passing through O_v to any PO must also go through some nodes in the cut-set. Basically, a cut of a given node v can be considered a "cutting boundary" of nodes that partitions the fanout cone of v from the node v. A cut-set is *K-feasible* if the cardinality of the cut-set is no more than K.

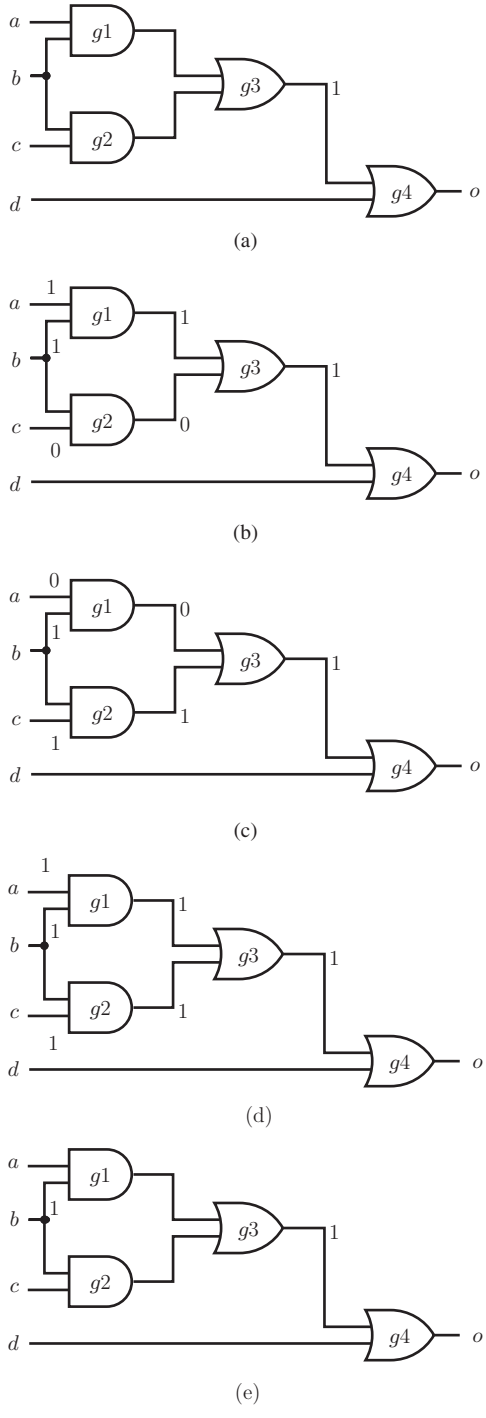

Figure 1.6 Example of recursive learning. (a) Gate with preassigned value: $g3$; (b) justifications (I) for $g1$, $g2$, and $g3$; (c) justifications (II) for $g1$, $g2$, and $g3$; (d) justifications (III) for $g1$, $g2$, and $g3$; (e) values implied from $g3 = 1$

Based on the above definitions, a *blocking cut* (B-cut) is defined as follows:

Definition 1.2 *Given a network, a cut-set S is a blocking cut (B-cut) of a node set if every path from the node set to the primary outputs must go through one and only one node in S.*

References

M. Damiani and G. De Micheli. Observability don't care sets and Boolean relations. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers, 1990 IEEE International Conference on*, pages 502–505, November 1990. doi: 10.1109/ICCAD.1990.129965.

N. K. Jha and S. Kundu. *Testing and Reliable Design of CMOS Circuits*. *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1990. URL http://books.google.com.hk/books?id=T0lTAAAAMAAJ.

T. Kirkland and M. R. Mercer. A topological search algorithm for ATPG. In *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 502–508, New York, 1987. ACM. ISBN: 0-8186-0781-5. doi: 10.1145/37888.37963.

W. Kunz and D. K. Pradhan. Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE Transactions Computer-Aided Design*, 13:1143–1158, 1994.