CHAPTER

Logic synthesis in a nutshell 6

Jie-Hong (Roland) Jiang
National Taiwan University, Taipei, Taiwan

Srinivas Devadas
Massachusetts Institute of Technology, Cambridge Massachusetts

ABOUT THIS CHAPTER

What is **logic synthesis?** As the name itself suggests, logic synthesis is the process of automatic production of logic components, in particular digital circuits. It is a subject about how to abstract and represent logic circuits, how to manipulate and transform them, and how to analyze and optimize them. *Why* does logic synthesis matter? Not only does it play a crucial role in the electronic design automation flow, its techniques also find broader and broader applications in formal verification, software synthesis, and other fields. *How* is logic synthesis done? Read on!

This chapter covers classic elements of logic synthesis for combinational circuits. After introducing basic data structures for Boolean function representation and reasoning, we will study technology-independent logic minimization, technology-dependent circuit optimization, timing analysis, and timing optimization. Some advanced subjects and important trends are presented as well for further exploration.

6.1 INTRODUCTION

Since Jack Kilby's invention of the first ***integrated circuit*** (IC) in 1958, there have been unprecedented technological advances. Intel co-founder Gordon E. Moore in 1965 predicted an important miniaturization trend for the semiconductor industry, known as Moore's Law, which says that the number of available transistors being economically packed into a single IC grows exponentially, doubling approximately every two years. This trend has continued for more than four decades, and perhaps will continue for another decade or even longer. At this time of 2008, the number of transistors in a single IC can be as many as several billion. This continual increase in design complexity under stringent time-to-market constraints is the primary driving force for changes in design tools and methodologies. To manage the ever-increasing complexity, people seek to maximally automate the design process and deploy techniques such as abstraction and hierarchy. Divide-and-conquer approaches are typical in the ***electronic design automation*** (EDA) flow and lead to different abstraction levels, such as the behavior level, ***register-transfer level*** (RTL), gate level, transistor level, and layout level from abstract to concrete.

Logic synthesis is the process that takes place in the transition from the register-transfer level to the transistor level. It is a highly automated procedure bridging the gap between high-level synthesis and physical design automation. Given a digital design at the register-transfer level, logic synthesis transforms it into a gate-level or transistor-level implementation. The highly engineered process explores different ways of implementing a logic function optimal with respect to some desired design constraints. The physical positions and interconnections of the gate layouts are then further determined at the time of physical design.

The main mathematical foundation of logic synthesis is the intersection of logic and algebra. The "algebra of logic" created by George Boole in 1847, a.k.a. *Boolean algebra*, is at the core of logic synthesis. (In our discussion we focus on two-element Boolean algebra [Brown 2003].) One of the most influential works connecting Boolean algebra and circuit design is Claude E. Shannon's M.S. thesis*, A Symbolic Analysis of Relay and Switching Circuits*, completed at the Massachusetts Institute of Technology in 1937. He showed that the design and analysis of switching circuits can be formalized using Boolean algebra, and that switching circuits can be used to solve Boolean algebra problems. Modern electronic systems based on digital (in contrast to analog) and two-valued (in contrast to multi-valued) principles can be more or less attributed to Shannon. The minimization theory of Boolean formulas in the two-level ***sum-of-products*** (SOP) form was established by Willard V. Quine in the 1950s. The minimization of SOP formulas found its wide application in IC design in the 1970s when ***programmable logic arrays*** (PLAs) were a popular design style for control logic implementation. It was the earliest stage of logic design minimization. When multilevel logic implementation became viable in the 1980s, the minimization theory and practice were broadened to the multi-level case.

Switching circuits in their original telephony application were strictly *combinational*, containing no memory elements. Purely combinational circuits however are not of great utility. For pervasive use in computation a combinational circuit needs to be augmented by memory elements that retain some of the state of a circuit. Such a circuit is *sequential* and implements a ***finite state machine*** (FSM). FSMs are closely related to finite automata, introduced in the theory of computation. Finite automata and finite state machines as well as their state minimization were extensively studied in the 1950s. Even though FSMs have limited computation power, any realistic electronic system as a whole can be seen as a large FSM because, after all, no system can have infinite memory resources. FSM state encoding for the two-level and multilevel logic implementations was studied extensively in the 1980s.

In addition to two-level and multilevel logic minimization, important algorithmic developments in logic synthesis in the 1980s include retiming of synchronous sequential circuits, algorithmic technology mapping, reduced ordered binary decision diagrams, and symbolic sequential equivalence checking using characteristic functions, just to name a few. Major logic synthesis tools of this period include, for example, ESPRESSO [Rudell 1987] and later MIS [Brayton 1987], developed at the University of California at Berkeley. They soon turned out to be the core engines of commercial logic synthesis tools.

In the 1990s, the subject of logic synthesis was much diversified in response to various IC design issues: power consumption, interconnect delay, testability, new implementation styles such as ***field programmable gate array*** (FPGA), etc. Important algorithmic breakthroughs over this period include, for instance, sequential circuit synthesis with retiming and resynthesis, don't care computation, image computation, timing analysis, Boolean reasoning techniques, and so on. Major academic software developed in this period include, *e.g.,* SIS [Sentovich 1992], the descendant of MIS.

In the 2000s, the directions of logic synthesis are driven by design challenges such as scalability, verifiability, design closure issues between logic synthesis and physical design, manufacture process variations, etc. Important developments include, for instance, effective satisfiability solving procedures, scalable logic synthesis and verification algorithms, statistical static timing analysis, statistical optimization techniques, and so on. Major academic software developed in this period include, *e.g.,* MVSIS [Gao 2002] and the ABC package [ABC 2005], with first release in 2005.

The advances of logic synthesis have in turn led to blossoming of EDA companies and the growth of the EDA industry. One of the first applications of logic optimization in a commercial use was to remap a netlist to a different standard cell library (in the first product, *remapper*, developed by Synopsys, an EDA company founded in 1986). It allowed an IC designer migrate a design from one library to another. Logic optimization could be used to optimize a gate-level netlist and map it into a target library. While logic optimization was finding its first commercial use for remapping, designers at major corporations, such as IBM, had already been demonstrating the viability of a top-down design methodology based on logic synthesis. At these corporations, internal simulation languages were coupled with synthesis systems that translated the simulation model into a gate-level netlist. Designers at IBM had demonstrated the utility of this synthesis-based design methodology on thousands of real industrial ICs. Entering a simulation model expressed using a ***hardware description language*** (HDL) makes logic synthesis and optimization move from a minor tool in a gate-level schematic based design methodology to the cornerstone of a highly productive IC design methodology. Commercial logic synthesis tools evolve and continue to incorporate developments addressing new design challenges.

The scope of logic synthesis can be identified as follows. An IC may consist of digital and analog components; logic synthesis is concerned with the digital part. For a digital system with sequential behavior, its state transition can be implemented in a synchronous or an asynchronous way depending on the existence of synchronizing clock signals. (Note that even a combinational circuit can be considered as a single-state sequential system.) Most logic synthesis algorithms focus on the synchronous implementation, and a few on the asynchronous one.

A digital system can often be divided into two portions: datapath and control logic. The former is concerned with data computation and storage, and often consists of arithmetic logic units, buses, registers/register files, etc.; the latter is concerned with the control of these data processing units. Unlike control logic, datapath circuits are often composed of regular structures. They are typically laid out manually by IC designers with full custom design to ensure that design constraints are satisfied, especially for high performance applications. Hence datapath design involves less logic synthesis efforts. In contrast, control logic is typically designed using logic synthesis. As the strengths of logic synthesis are its capabilities in logic minimization, it simplifies control logic. Consequently logic synthesis is particularly good for control-dominating applications, such as protocol processing, but not for arithmetic-intensive applications, such as signal processing.

Aside from the design issues related to circuit components, market-oriented decisions influence the design style chosen in implementing a product. The amount of design automation and logic synthesis efforts depends heavily on such decisions. Design styles based on full custom design, standard cells, and FPGAs represent typical trade-offs. In full custom design, logic synthesis is of limited use, mainly only in synthesizing performance non-critical controllers. For standard cell and FPGA based designs, a great portion of a design may be processed through logic synthesis. It is not surprising that logic synthesis is widely applied in ***application specific ICs*** (ASICs) and FPGA-based designs.

6.2 DATA STRUCTURES FOR BOOLEAN REPRESENTATION AND REASONING

The basic mathematical objects to be dealt with in this chapter are Boolean functions. How to compactly represent Boolean functions (the subject of logic minimization) and how to efficiently solve Boolean constraints (the subject of

Boolean reasoning) are closely related questions that play central roles in logic synthesis. There are several data structures for Boolean function representation and manipulation. For *Boolean representation*, we introduce some of the most commonly used ones, in particular, ***sum-of-products*** (SOP), ***product-of-sums*** (POS), ***binary decision diagrams*** (BDDs), ***and-inverter graphs*** (AIGs), and **Boolean networks**, among many others. For Boolean reasoning, we discuss how BDD, SAT, and AIG packages can serve as the core engines for Boolean function manipulation and for automatic reasoning of Boolean function properties. The efficiency of a data structure is mainly determined by its succinctness in representing Boolean functions and its capability of supporting Boolean manipulation. Each data structure has its own strengths and weaknesses; there is not a single data structure that is universally good for all applications. Therefore, conversion among different data types is a necessity in logic synthesis, where various circuit transformation and verification techniques are applied.

6.2.1 **Quantifier-free and quantified Boolean formulas**

We introduce (quantifier-free) Boolean formulas for Boolean function representation and ***quantified Boolean formulas*** (QBFs) for Boolean reasoning.

A **Boolean variable** is a variable that takes on binary values $\mathbb{B} = \{\text{false}, \text{true}\}$, or $\{0, 1\}$, under a truth assignment; a **literal** is a Boolean variable or its complement. In the **n-dimensional Boolean space** or **Boolean n-space** $\mathbb{B}^n$, an atomic element (or vertex) $a \in \mathbb{B}^n$ is called a **minterm**, which corresponds to a truth assignment on a vector of n Boolean variables.

An n-ary **completely specified Boolean function** $f : \mathbb{B}^n \rightarrow \mathbb{B}$ maps every possible truth assignment on the n input variables to either true or false. Let symbol "–", "X", or "2" denote the don't care value. We augment $\mathbb{B}$ to $\mathbb{B}_+ = \mathbb{B} \cup \{-\}$ and define an **incompletely specified Boolean function** $f : \mathbb{B}^n \rightarrow \mathbb{B}_+$, which maps every possible truth assignment on the n input variables to true, false, or don't care. For some $a \in \mathbb{B}^n$, $f(a) = -$ means the function value of f under the truth assignment a does not matter. That is, a is a don't care condition for f. Unless otherwise stated, we shall assume that a Boolean function is completely specified.

The mapping induced by a set of Boolean functions can be described by a **functional vector** or a **multiple-output function $\boldsymbol{f}$**, which combines $m > 1$ Boolean functions into a mapping $\boldsymbol{f} : \mathbb{B}^n \rightarrow \mathbb{B}^m$ if $\boldsymbol{f}$ is completely specified, or a mapping $\boldsymbol{f} : \mathbb{B}^n \rightarrow {\mathbb{B}_+}^m$ if $\boldsymbol{f}$ is incompletely specified.

For a completely specified function f, we define its **onset** $f^{\text{on}} = \{a \in \mathbb{B}^n \mid f(a) = 1\}$ and **offset** $f^{\text{off}} = \{a \in \mathbb{B}^n \mid f(a) = 0\}$. For an incompletely specified function f, in addition to the onset and offset, we have the **dcset** $f^{\text{dc}} = \{a \in \mathbb{B}^n \mid f(a) = -\}$. Although the onset, offset, and dcset are named sets rather than functions, we will see later that sets and functions can be unified through the use of the so-called **characteristic functions**.

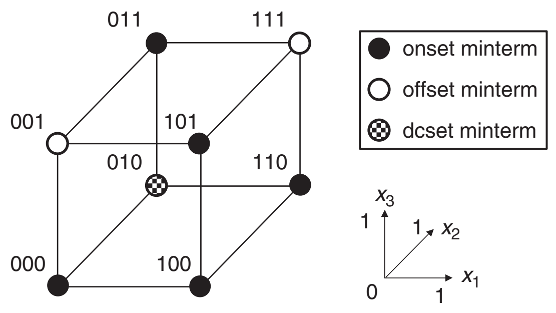


FIGURE 6.1

Boolean 3-space and a 3-ary Boolean function.

Example 6.1 The Boolean 3-space spanned by the variable vector (x_1, x_2, x_3) can be viewed as a combinatorial cube as shown in Figure 6.1, where the labeled vertices represent the minterms and two minterms are connected by an edge if their Hamming distance is one (that is, their binary codes differ in one position). The onset $f^{\text{on}} = \{000, 011, 100, 101, 110\}$, offset $f^{\text{off}} = \{001, 111\}$, and dcset $f^{\text{dc}} = \{010\}$ of some function f are embedded in the combinatorial cube.

A completely specified Boolean function *f* is a **tautology**, written as $f \equiv 1$ or $f \Leftrightarrow 1$, if its onset equals the universal set, *i.e.*, the entire Boolean space. In other words, the output of *f* equals 1 under every truth assignment on the input variables.

Any Boolean function can be expressed in a **Boolean formula**. Table 6.1 shows the building elements (excluding the last two symbols, $\exists$ and $\forall$) of a Boolean formula. Symbols $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ are Boolean connectives. A Boolean formula φ can be built recursively through the following formation rules:

$$\varphi ::= 0|1|A|\neg\varphi_1|\varphi_1 \wedge \varphi_2|\varphi_1 \vee \varphi_2|\varphi_1 \Rightarrow \varphi_2|\varphi_1 \Leftrightarrow \varphi_2 \quad (6.1)$$

where the symbol "::=" is read as "can be" and symbol "|" as "or". That is, a Boolean formula φ can be a constant 0, a constant 1, an atomic Boolean variable from a variable set *A*, $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, or $\varphi_1 \Leftrightarrow \varphi_2$, built from Boolean formulas φ_1 and φ_2. To save on parentheses and enhance readability, we assume the precedence of the Boolean connectives $\Leftrightarrow$, $\Rightarrow$, $\vee$, $\wedge$, $\neg$ is in an ascending order. Also we often omit expressing the conjunction symbol $\wedge$ in a formula.

Example 6.2 The Boolean formula

$$((x_1 \vee (\neg x_2)) \vee ((\neg x_1) \wedge x_3)) \wedge (x_1 \wedge (\neg x_2))$$

can be shortened to

$$((x_1 \vee \neg x_2) \vee \neg x_1 x_3)(x_1 \neg x_2)$$

Table 6.1 Symbolic Notation and Meaning

Symbol	**Symbol Name**	**English Meaning**
(	left parenthesis	for punctuation
)	right parenthesis	for punctuation
$\neg$, $'$	complement symbol	logical "not"
$\wedge$, .	conjunction symbol	logical "and"
$\vee$, $+$	disjunction symbol	logical "(inclusive) or"
$\Rightarrow$	implication symbol	logical "if . . . , then . . . "
$\Leftrightarrow$, $\equiv$	bi-implication symbol	logical ". . . if and only if . . . "
$\exists$	existential quantifier	"there exists . . . "
$\forall$	universal quantifier	"for all . . . "

Using the associativity of disjunction and conjunction, we can further shorten the formula to

$$(x_1 \vee \neg x_2 \vee \neg x_1 x_3) x_1 \neg x_2$$

but we can no longer trace a unique sequence of rules used to derive this formula.

A set of Boolean operators is called **functionally complete** if they are sufficient to generate any Boolean function. Note that not all of the above Boolean connectives are necessary to form a set of functionally complete operators. For example, the sets $\{\neg, \wedge\}$ and $\{\neg, \Rightarrow\}$ are functionally complete, whereas $\{\wedge, \Rightarrow\}$ is not.

We may consider a Boolean function as the semantics of some Boolean formulas. There are different (syntactical) Boolean formulas representing the same (semantical) Boolean functions. It is this flexibility that makes logic synthesis an art.

Boolean operations over Boolean functions can be defined in terms of set operations, such as union $\cup$, intersection $\cap$, and complement over sets. Boolean function $h = f \wedge g$ has onset $h^{\text{on}} = f^{\text{on}} \cap g^{\text{on}}$ and offset $h^{\text{off}} = f^{\text{off}} \cup g^{\text{off}}$; Boolean function $h = f \vee g$ has onset $h^{\text{on}} = f^{\text{on}} \cup g^{\text{on}}$ and offset $h^{\text{off}} = f^{\text{off}} \cap g^{\text{off}}$; Boolean function $h = \neg f$ (also denoted as $\bar{f}$ or f') has onset $h^{\text{on}} = f^{\text{off}}$ and offset $h^{\text{off}} = f^{\text{on}}$. The dcset of function h can be derived using the fact that the union of the onset, offset, and dcset is equal to the universal set.

Quantified Boolean formulas (QBFs) generalize (quantifier-free) Boolean formulas with the additional universal and existential quantifiers: $\forall$ and $\exists$, respectively. In writing a QBF, we assume that the precedences of the quantifiers are lower than those of the Boolean connectives. In a QBF, variables being quantified are called **bound variables**, whereas those not quantified are called **free variables**.

Example 6.3 Consider the QBF $\forall x_1, \exists x_2.f(x_1, x_2, x_3)$, where f is a Boolean formula. It is read as *"For every* (truth assignment of) x_1, *there exists some* (truth assignment of) x_2, $f(x_1, x_2, x_3)$." In this case, x_1 and x_2 are bound variables, and x_3 is a free variable.

Any QBF can be rewritten as a quantifier-free Boolean formula through **quantifier elimination** by formula expansion (among other methods), *e.g.,*

$$\forall x.f(x,y) = f(0,y) \wedge f(1,y)$$

and

$$\exists x.f(x,y) = f(0,y) \vee f(1,y)$$

where f is a Boolean formula. Consequently, for any QBF φ, there exists an equivalent quantifier-free Boolean formula that *refers only to the free variables* of φ. For a QBF of size n with k bound variables, its quantifier-free Boolean formula derived by formula expansion can be of size $O(2^n \cdot k)$. QBFs are thus of the same expressive power as quantifier-free Boolean formulas, but can be exponentially more succinct.

Example 6.4 The QBF $\forall x_1, \exists x_2.f(x_1, x_2, x_3)$ can be rewritten as

$$\begin{aligned} & \forall x_1.(f(x_1,0,x_3) \vee f(x_1,1,x_3)) \\ = \; & (\exists x_2.f(0,x_2,x_3)) \wedge (\exists x_2.f(1,x_2,x_3)) \\ = \; & (f(0,0,x_3) \vee f(0,1,x_3)) \wedge (f(1,0,x_3) \vee f(1,1,x_3)) \end{aligned}$$

Note that $\forall x_1, \exists x_2.f(x_1, x_2, x_3)$ differs from and is, in fact, weaker than $\exists x_2, \forall x_1.f(x_1, x_2, x_3)$. That is, $(\exists x_2, \forall x_1.f(x_1, x_2, x_3)) \Rightarrow (\forall x_1, \exists x_2.f(x_1, x_2, x_3))$. In contrast, $\forall x_1, \forall x_2.f(x_1, x_2, x_3)$ is equivalent to $\forall x_2, \forall x_1.f(x_1, x_2, x_3)$, and similarly $\exists x_1, \exists x_2.f(x_1, x_2, x_3)$ is equivalent to $\exists x_2, \exists x_1.f(x_1, x_2, x_3)$.

Moreover, it can be verified that the universal quantification $\forall$ commutes with the conjunction $\wedge$, whereas the existential quantification $\exists$ commutes with the disjunction $\vee$. That is, for any QBFs φ_1 and φ_2, we have

$$\forall x.(\varphi_1 \wedge \varphi_2) = \forall x.\varphi_1 \wedge \forall x.\varphi_2$$

whereas

$$\exists x.(\varphi_1 \vee \varphi_2) = \exists x.\varphi_1 \vee \exists x.\varphi_2$$

Nonetheless in general $\forall$ does not commute with $\vee$, whereas $\exists$ does not commute with $\wedge$. That is, in general

$$\forall x.(\varphi_1 \vee \varphi_2) \neq \forall x.\varphi_1 \vee \forall x.\varphi_2$$

and

$$\exists x.(\varphi_1 \wedge \varphi_2) \neq \exists x.\varphi_1 \wedge \exists x.\varphi_2$$

On the other hand, for any QBF φ, we have

$$\neg \forall x.\varphi = \exists x.\neg \varphi \tag{6.2}$$

and

$$\neg\exists x.\varphi = \forall x.\neg\varphi \tag{6.3}$$

Because $\forall$ and $\exists$ can be converted to each other through negation, either quantifier solely is suffcient to represent QBFs.

An important fact about QBFs is that they are equivalent under renaming of bound variables. For example, $\forall x.f(x, y) = \forall z.f(z, y)$ and $\exists x.f(x, y) = \exists z.f(z, y)$. Renaming bound variables is often necessary if we want to rewrite a QBF in a different way. Being able to identify the scope of a quantifier is crucial for such renaming.

Example 6.5 In the QBF

$$Q_1x, Q_2y.(\,f_1(x,y,z) \vee f_2(\,y,z) \wedge Q_3x.f_3(x,y,z))$$

with $Q_i \in \{\forall, \exists\}$, quantifier Q_1 is applied only to the variable x of f_1, quantifier Q_2 is applied to the y variables of all the functions, and quantifier Q_3 is applied only to the variable x of f_3. The QBF can be renamed as

$$Q_1a, Q_2b.(f_1(a,b,z) \vee \neg f_2(b,z) \wedge Q_3x.f_3(x,b,z))$$

In studying QBFs, it is convenient to introduce a uniform representation, the so-called **prenex normal form**, where the quantifiers of a QBF are moved to the left leaving a quantifier-free Boolean formula on the right. That is,

$$Q_1x_1, Q_2x_2, \ldots, Q_nx_n.f(x_1, x_2, \ldots, x_n)$$

where $Q_i \in \{\forall, \exists\}$ and f is a quantifier-free Boolean formula. Such movement is always possible by Equations (6.2) and (6.3) as well as the following equalities: For QBFs φ_1 and φ_2,

$$(\varphi_1 \diamond Qx.\varphi_2) = Qx.(\varphi_1 \diamond \varphi_2) \text{ if } x \text{ is not a free variable in } \varphi_1 \tag{6.4}$$

where $Q \in \{\forall, \exists\}$ and $\diamond \in \{\wedge, \vee\}$,

$$(\varphi_1 \Rightarrow \forall x.\varphi_2) = \forall x.(\varphi_1 \Rightarrow \varphi_2) \text{ if } x \text{ is not a free variable in } \varphi_1 \tag{6.5}$$

$$(\varphi_1 \Rightarrow \exists x.\varphi_2) = \exists x.(\varphi_1 \Rightarrow \varphi_2) \text{ if } x \text{ is not a free variable in } \varphi_1 \tag{6.6}$$

$$((\forall x.\varphi_1) \Rightarrow \varphi_2) = \exists x.(\varphi_1 \Rightarrow \varphi_2) \text{ if } x \text{ is not a free variable in } \varphi_2. \quad \text{and} \tag{6.7}$$

$$((\exists x.\varphi_1) \Rightarrow \varphi_2) = \forall x.(\varphi_1 \Rightarrow \varphi_2) \text{ if } x \text{ is not a free variable in } \varphi_2 \tag{6.8}$$

With the renaming of bound variables, we know that the above conditions, x not a free variable in φ_i, can always be satisfied. Thereby any QBF can be converted into an equivalent formula in prenex normal form.

Prenex normal form is particularly suitable for the study of **computational complexity**. The number of alternations between existential and universal quantifiers in a QBF in prenex normal form directly reflects the difficulty in solving the formula. (In solving a QBF φ, we shall assume that all variables of φ are quantified, *i.e.*, no free variables in φ.) For instance, there are three alternations of quantifiers in the QBF $\forall x_1, \forall x_2, \exists x_3, \forall x_4, \exists x_5.f(x_1, \ldots, x_5)$. The more alternations of quantifiers are in a QBF in prenex normal form, the higher the computational complexity is in solving it. The levels of difficulties induce the **polynomial hierarchy**, a hierarchy of complexity classes, in complexity theory (see, *e.g.*, [Papadimitriou 1993] for comprehensive introduction). The problem of solving QBFs is known as ***quantified satisfiability*** (QSAT); in particular, the problem is known as QSAT$_i$ for QBFs in prenex normal form with i alternations of quantifiers. The entire polynomial hierarchy is contained by the PSPACE complexity class; the problem QSAT (without an a *priori* alternation bound i) is among the hardest in PSAPCE, *i.e.*, PSPACE-complete. A particularly interesting special case is QSAT$_0$ with all variables quantified existentially. It is known as the ***Boolean satisfiability*** (SAT) problem, which is NP-complete [Garey 1979]. Solving QBFs is much harder than solving the satisfiability of Boolean formulas.

In the above discussion of QBF solving, we assumed all variables are not free. For a QBF φ with free variables, we say that it is **satisfiable** (respectively **valid**) if it is true under *some* (respectively *every*) truth assignment on the set of free variables. Hence asking about the *satisfiability* of a Boolean formula $f(\boldsymbol{x})$ is the same as asking about the *validity/satisfiability* of the QBF $\exists \boldsymbol{x}.f(\boldsymbol{x})$; asking about the *validity* of a Boolean formula $f(\boldsymbol{x})$ is the same as asking about the *validity/satisfiability* of the QBF $\forall \boldsymbol{x}.f(\boldsymbol{x})$. Note that the validity and satisfiability of a formula are the same if there are no free variables.

Although QBFs are not directly useful for circuit representation, many computational problems in logic synthesis and verification (such as image computation, don't care computation, Boolean resubstitution, combinational equivalence checking, etc.) can be posed as QBF solving. Once a computational task is written in a QBF, its detailed algorithmic solution is almost apparent and can be derived using Boolean reasoning engines.

6.2.2 Boolean function manipulation

In addition to Boolean AND, OR, NOT operations, **cofactor** is an elementary Boolean operation. For a function $f(x_1, \ldots, x_i, \ldots, x_n)$, the **positive cofactor** and **negative cofactor** of f with respect to x_i are $f(x_1, \ldots, 1, \ldots, x_n)$, denoted as f_{x_i} or $f|_{x_i=1}$, and $f(x_1, \ldots, 0, \ldots, x_n)$, denoted as $f_{\neg x_i}$ or $f|_{x_i=0}$, respectively. We can also cofactor a Boolean function with respect to a **cube**, namely the conjunction of a set of literals, by iteratively cofactoring the function with each literal in the cube.

Example 6.6 Cofactoring the Boolean function $f = x_1x_2\neg x_3 \vee x_4\neg x_5x_6$ with respect to the cube $c = x_1x_2\neg x_5$ yields function $f_c = \neg x_3 \vee x_4x_6$.

Universal and existential quantifications can be expressed in terms of cofactor, with

$$\forall x_i.f = f_{x_i} \wedge f_{\neg x_i} \tag{6.9}$$

and

$$\exists x_i.f = f_{x_i} \vee f_{\neg x_i} \tag{6.10}$$

Moreover, the **Boolean difference** $\frac{\partial f}{\partial x_i}$ of f with respect to variable x_i is defined as

$$\frac{\partial f}{\partial x_i} = \neg(f_{x_i} \equiv f_{\neg x_i}) = f_{x_i} \oplus f_{\neg x_i} \tag{6.11}$$

where $\oplus$ denotes an exclusive-or (XOR) operator. Using the Boolean difference operation, we can tell whether a Boolean function functionally depends on a variable. If $\frac{\partial f}{\partial x_i}$ equals constant 0, then the valuation of f does not depend on x_i, that is, x_i is a redundant variable for f. We call that x_i is a **functional support variable** of f if x_i is not a redundant variable.

By **Shannon expansion**, every Boolean function f can be decomposed with respect to some variable x_i as

$$f = x_i f_{x_i} \vee \neg x_i f_{\neg x_i} \tag{6.12}$$

Note that the variable x_i needs not be a functional support variable of f.

6.2.3 Boolean function representation

Below we discuss different ways of representing Boolean functions.

6.2.3.1 *Truth table*

The mapping of a Boolean function can be exhaustively enumerated with a **truth table**, where every truth assignment has a corresponding function value listed.

Example 6.7 Figure 6.2 shows the truth table of the majority function $f(x_1, x_2, x_3)$, which valuates to true if and only if at least two of the variables $\{x_1, x_2, x_3\}$ valuate to true.

Truth tables are **canonical** representations of Boolean functions. That is, two Boolean functions are equivalent if and only if they have the same truth table.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

FIGURE 6.2

Truth table of the 3-ary majority function.

Canonicity is an important property that may be useful in many applications of logic synthesis and verification.

For practical implementation, a truth table is effective in representing functions with a few input variables (often no more than 5 or 6 variables for modern computers having a word size 32 or 64 bits). By storing a truth table as a computer word, basic Boolean operations over two small functions can be done in constant time by parallel bitwise operation over their truth tables. Truth tables however are impractical to represent functions with many input variables.

6.2.3.2 *SOP*

Sum-of-products (SOP), or ***disjunctive normal form*** (DNF) as it is called in computer science, is a special form of Boolean formulas consisting of disjunctions (sums) of conjunctions of literals (product terms or cubes). It is a flat structure corresponding to a two-level circuit representation (the first level of AND gates and the second level of an OR gate). In two-level logic minimization, the set of product terms (*i.e.*, cubes) of an SOP representation of a Boolean function is called a **cover** of the Boolean function. A Boolean function may have many different covers, and a cover uniquely determines a Boolean function.

Example 6.8 The expression $f = ab\neg c + a\neg bc + \neg abc + \neg a\neg b\neg c$ is in SOP form. The set $\{ab\neg c, a\neg bc, \neg abc, \neg a\neg b\neg c\}$ of cubes forms a cover of function f.

In our discussion, we often do not distinguish a cover and its represented function.

Every Boolean formula can be rewritten in an SOP representation. Unlike the truth table representation, the SOP representation is not canonical. In fact, how to express a Boolean function in the most concise SOP-form is intractable (in fact, NP-complete), and is termed **two-level logic minimization**.

Given SOP as the underlying Boolean representation, we study its usefulness for Boolean manipulation. Consider the conjunction of two cubes. It is computable in time linear in the number of literals because, having defined cubes as sets of literals, we compute the conjunction of cubes c and d, denoted $q = c \cap d$, by actually taking the union of the literal sets in c and d. However if $q = c \cap d$ computed in this fashion contains both a literal l and its complement $\neg l$, then the intersection is empty. Similarly the conjunction of two covers can be obtained by taking the conjunction of each pair of the cubes in the covers. Therefore, the AND operation of two SOP formulas is of quadratic time complexity. On the other hand, the OR operation is of constant time complexity since the disjunction of two SOP formulas is readily in SOP form. The complement operation is of exponential time complexity in the worst case.

Example 6.9 Complementing the function

$$f = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n$$

will result in 2^n product terms in the SOP representation.

In addition to the above basic Boolean operations, SAT and TAUTOLOGY checkings play a central role in Boolean reasoning. Checking whether an SOP formula is satisfiable is of constant time complexity since any (irredundant) SOP formula other than constant 0 must be satisfiable. In contrast, checking whether an SOP formula is tautological is intractable, in fact, coNP-complete. When compared with other data structures to be introduced, SOP is not commonly used as the underlying representation in Boolean reasoning engines, but mainly used in two-level and multilevel logic minimization.

For the purposes of minimizing two-level logic functions, efficient procedures for performing Boolean operations on SOP representations or covers are desirable. A package for performing various Boolean operations such as conjunction, disjunction, and complementation is available as part of the ESPRESSO program [Rudell 1987].

6.2.3.3 *POS*

Product-of-sums (POS), or ***conjunctive normal form*** (CNF) as it is called in computer science, is a special form of Boolean formulas consisting of conjunctions (products) of disjunctions of literals (clauses). It is a flat structure corresponding to a two-level circuit representation (the first level of OR gates and the second level of an AND gate).

Example 6.10 The formula $(a + b + \neg c)(a + \neg b + c)(\neg a + b + c)(\neg a + \neg b + \neg c)$ is in POS form.

Every Boolean formula has an equivalent formula in POS form. Even though POS seems just the dual of SOP, it is not as commonly used in circuit design as SOP partly due to the characteristics of CMOS circuits, where NMOS is preferable to PMOS. Nevertheless it is widely used in Boolean reasoning. Satisfiability (SAT) solving over CNF formulas is one of the most important problems in computer science. In fact, every NP-complete problem can be reformulated in polynomial time as a SAT problem.

Given POS as the underlying data structure, we study its usefulness for Boolean function manipulation. For the AND operation, it is of constant time complexity since the conjunction of two POS formulas is readily in POS. For the OR operation, it is of quadratic time complexity since in the worst case the disjunction of two POS formulas must be converted to a POS formula by the distributive law.

Example 6.11 Given POS formulas $\varphi_1 = (a)\cdot(b)$ and $\varphi_2 = (c)\cdot(d)$, their disjunction $\varphi_1 + \varphi_2$ equals $(a + c)\cdot(a + d)\cdot(b + c)\cdot(b + d)$.

On the other hand, the complement operation is of exponential time complexity since in the worst case a POS formula may need to be complemented with De Morgan's Law followed by the distributive law.

Example 6.12 Complementing the $2n$-input Achilles heel function

$$f = (x_1 + y_1) \cdot (x_2 + y_2) \cdots (x_n + y_n)$$

will result in 2^n clauses in the POS representation.

As for the SAT and TAUTOLOGY checkings of POS formulas, the former is NP-complete, and the latter is of constant time complexity because any (irredundant) POS formula other than constant 1 cannot be a tautology. The POS representation is commonly used as the underlying representation in Boolean reasoning engines, called **SAT solvers**.

6.2.3.4 *BDD*

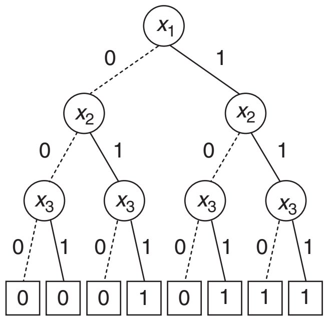
Binary decision diagrams (BDDs) were first proposed by Lee [Lee 1959] and further developed by Akers [Akers 1978]. In their original form, BDDs are not canonical in representing Boolean functions. To canonicalize the representation, Bryant [Bryant 1986, 1992] introduced restrictions on BDD variable ordering and proposed several reduction rules, leading to the well-known ***reduced ordered BDDs*** (ROBDDs). Among various types of decision diagrams, ROBDDs are the most widely used, and will be our focus.

Consider using an n-level binary tree to represent an arbitrary n-input Boolean function $f(x_1, \ldots, x_n)$. The binary tree, called a BDD, contains two types of nodes. A **terminal** node, or leaf, υ has as an attribute a value $value(\upsilon) \in \{0, 1\}$. A **non-terminal** node υ has as attributes an argument level-index $index(\upsilon) \in \{1, \cdots, n\}$ and two children: the 0-child, denoted $else(\upsilon) \in V$, and the 1-child, denoted $then(\upsilon) \in V$. If $index(\upsilon) = i$, then x_i is called the **decision variable** for node υ. Every node υ in a BDD corresponds to a Boolean function $f[\upsilon]$ defined recursively as follows.

1. For a terminal node υ,
   (a) If $value(\upsilon) = 1$, then $f[\upsilon] = 1$.
   (b) If $value(\upsilon) = 0$, then $f[\upsilon] = 0$.

2. For a non-terminal node υ with $index(\upsilon) = i$,
   $f[v](x_1, \ldots, x_n) = \neg x_i \cdot f[else(v)](x_1, \ldots x_n) + x_i \cdot f[then(v)](x_1, \ldots, x_n)$

Recall that, in Shannon expansion, a Boolean function f can be written as $f = x_i f_{x_i} + \neg x_i f_{\neg x_i}$. Suppose a BDD node representing some function f is controlled by variable x_i. Then its 0-child and 1-child represent functions $f_{\neg x_i}$ and f_{x_i}, respectively. Accordingly a BDD in effect represents a recursive Shannon expansion. For a complete binary tree, it is easily seen that we can always find some value assignment to the leaves of a BDD to implement any n-input function $f(x_1, \ldots, x_n)$ because every truth assignment of variables $x_1, \ldots, x_n$ activates exactly one path from the root node to a unique leaf with the right function value. Note that a BDD represents the offset and the onset of a function as disjoint covers, where each cube in the cover corresponds to a path from the root node to some terminal node.



x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

FIGURE 6.3

Binary tree representation of the majority function.

Example 6.13 The binary tree representation of the majority function is shown in Figure 6.3, where a circle (square) represents a non-terminal (terminal) node and a dotted (solid) edge indicates the pointed 0-child (1-child) of its parent node.

Definition 6.1. *A BDD is* **ordered** (*i.e., an OBDD*) *if the nodes on every path from the root node to a terminal node of the BDD follow the same variable ordering.*

Definition 6.2. *Two OBDDs D_1 and D_2 are* isomorphic *if there exists a one-to-one function σ from the nodes of D_1 onto the nodes of D_2 such that for any node υ if $\sigma(\upsilon) = w$, then either both υ and w are terminal nodes with value(υ) = value(w), or both υ and w are non-terminal nodes with index(υ) = index(w), σ(else(υ)) = else(w) and σ(then(υ)) = then(w).*

Since an OBDD only contains one root and the children of any non-terminal node are distinguished, the isomorphic mapping σ between OBDDs D_1 and D_2 is constrained and easily checked for. The root in D_1 must map to the root in D_2, the root's 0-child in D_1 must map to the root's 0-child in D_2, and so on all the way to the terminal nodes. Testing two OBDDs for isomorphism is thus a simple linear-time check.

Definition 6.3. ([Bryant 1986]). *An OBDD D is* **reduced** *if it contains no node υ with else(υ) = then(υ) nor does it contain distinct nodes υ and w such that the subgraphs rooted in υ and w are isomorphic.*

An ***reduced OBDD*** (ROBDD) can be constructed from an OBDD with the following three reduction rules:

1. Two terminal nodes with the same value attribute are merged.
2. Two non-terminal nodes u and υ with the same decision variable, the same 0-child, *i.e.*, *else*(u) = *else*(υ), and the same 1-child, *then*(u) = *then*(υ) are merged.
3. A non-terminal node υ with *else*(υ) = *then*(υ) is removed, and its incident edges are redirected to its child node.

Iterating the reduction steps bottom-up on an OBDD until no further modification can be made, we obtain its unique corresponding ROBDD. These rules ensure that no two nodes of the ROBDD are structurally (also functionally) isomorphic, and that the derived ROBDD has fewest nodes under a given variable ordering. It can be shown that no two nodes of an ROBDD represent the same Boolean function, and thus two ROBDD of the same Boolean function must be isomorphic. That is, ROBDDs are a canonical representation of Boolean functions. Every function has a unique ROBDD for a given variable ordering.

Theorem 6.1 (ROBDD Canonicity [Bryant 1986]). *For any Boolean function f, there is a unique (up to isomorphism) ROBDD denoting f, and any other OBDD denoting f contains more nodes.*

Proof. A sketch of the proof is given using induction on the number of inputs.

Base case: If f has zero inputs, it can be either the unique 0 or 1 ROBDD.

Induction hypothesis: Any function g with a number of inputs $< k$ has a unique ROBDD.

Choose a function f with k inputs. Let D and D' be two ROBDDs for f under the same ordering. Let x_i be the input with the lowest index in the ROBDDs D and D'. Define the functions f_0 and f_1 as f_{x_i} and $f_{\neg x_i}$, respectively. Both f_0 and f_1 have less than k inputs, and by the induction hypothesis these are represented by unique ROBDDs D_0 and D_1.

We can have nodes in common between D_0 and D_1 or have no nodes in common between D_0 and D_1. If there are no nodes in common between D_0 and D_1 in D, and no nodes in common between D_0 and D_1 in D', then clearly D and D' are isomorphic.

Consider the case where there is a node u that is shared by D_0 and D_1 in D. There is a node u' in the D_0 of D' that corresponds to u. If u' is also in D_1 of D', then we have a correspondence between u in D and u' in D'. However, there could be another node u'' in the D_1 of u'' that also corresponds to u. While the existence of this node implies that D and D' are not isomorphic, the existence of u' and u'' in D' is a contradiction to the statement of the theorem, since the two nodes root isomorphic subgraphs corresponding to u. (This would imply that D' is not reduced.) Therefore, u'' cannot exist, and D and D' are isomorphic. □

Example 6.14 Figure 6.4, from 6.4a to 6.4c, shows the derivation of the ROBDD from the binary tree of the majority function.

Example 6.15 Consider the OBDD of Figure 6.5a. By the first reduction rule, we can merge all the terminal nodes with value 0 and all the terminal nodes with value 1. The functions rooted in the two nodes with control variable x_3 are identical, namely x_3. By the second reduction rule, we can delete one of the identical nodes and make the nodes that were pointing to the deleted node (those nodes whose 0- or 1-child correspond to the deleted node) point instead to the other node. This does not change the Boolean function corresponding to the OBDD. The simplified OBDD is shown in Figure 6.5b. In Figure 6.5b there is a node with control variable x_2 whose 0-child and 1-child both point to the same node. This node is redundant because the function f rooted in the node corresponds to function

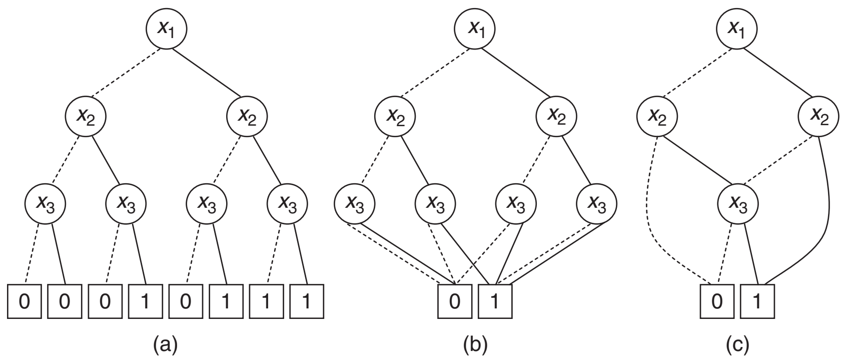


FIGURE 6.4

From binary tree to ROBDD.

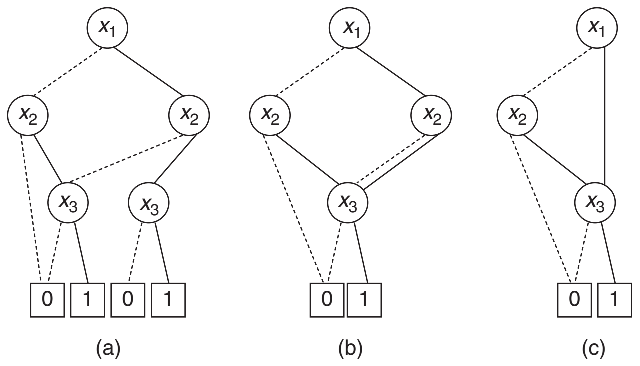


FIGURE 6.5

OBDD and simplified OBDDs.

$$f = x_2 \cdot x_3 + \neg x_2 \cdot x_3 = x_3$$

Thus, by the third reduction rule, all the nodes that point to f can be made to point to its 0- or 1-child without changing the Boolean function corresponding to the OBDD as illustrated in Figure 6.5c.

Example 6.16 Figure 6.6 shows a reduction example using a labeling technique for the ROBDD taken from [Bryant 1986]. We first assign the 0 and 1 terminal nodes a and b labels, respectively, in Figure 6.6a. Next, the right node with control variable x_3 is assigned label c. Upon encountering the other node with node with control variable x_3, we find that the second reduction rule is satisfied and assign this node the label c as well. Proceeding upward we assign the label c to the right node with control variable x_2 since the third reduction rule is satisfied for this node. (The 0-child and the 1-child of this node have the same label.) The left node with control variable x_2 is assigned label d, and the root node is assigned the label e. Note that the nodes are labeled in such a way that each label indicates a unique (sub-)ROBDD. Sorting and deleting redundant nodes results in the ROBDD of Figure 6.6b.

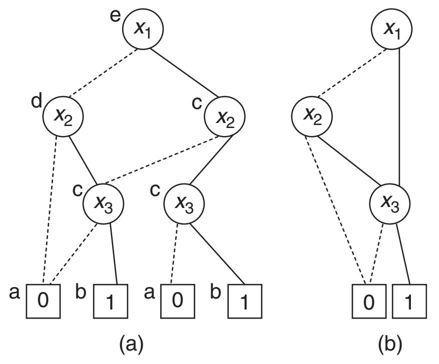


FIGURE 6.6

Reduction example.

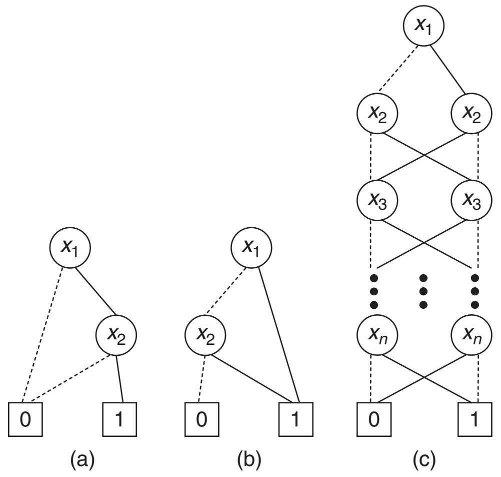


FIGURE 6.7

ROBDD examples: (a) ROBDD of function $f = x_1 \wedge x_2$. (b) ROBDD of function $f = x_1 \vee x_2$. (c) ROBDD of the n-ary odd parity function.

Example 6.17 To see that ROBDDs represent the offset and the onset of a function as disjoint covers, consider the examples of Figure 6.7. The ROBDD in (a) represents the function $f = x_1 \wedge x_2$. There are exactly two paths leading to the 0 terminal node. If x_1 is a 0, then the function represented by the ROBDD evaluates to a 0 since the 0-child of the node with index x_1 is the 0 terminal node. If x_1 is a 1 and x_2 is a 0, the function evaluates to a 0. Thus, the offset is represented as $\{\neg x_1, x_1\neg x_2\}$. The two cubes in the cover are disjoint. If x_1 and x_2 are both 1, the function evaluates to a 1. The onset is the singleton $\{x_1x_2\}$. Note that a cube of these covers corresponds to a single path from the root node to some terminal node. Similar analysis can be applied for the ROBDDs in (b) and (c).

In representing a Boolean function, different variable orderings may result in ROBDDs with very different sizes (in terms of the number of nodes).

Example 6.18 If the variables in the function $f = ab + cd$ are ordered as $index(a) < index(b) < index(c) < index(d)$ (a on top and d at bottom), the resulting ROBDD has only 4 non-terminal nodes. However, if the order $index(a) < index(c) < index(b) < index(d)$ is chosen, there are 7 non-terminal nodes.

Due to the sensitivity of ROBDD sizes to the chosen variable ordering, finding a suitable ordering becomes an important problem to obtain a reasonably sized ROBDD representing a given logic function. Finding the best variable ordering that minimizes the ROBDD size is coNP-complete [Bryant 1986]. However, there are good heuristics. For example, practical experience suggests that symmetric and/or correlated variables should be ordered close to each other. Other heuristics attempt to generate an ordering such that the structure of the ROBDD under this ordering mimics the given circuit structure.

It is not surprising that there exists a family of Boolean functions whose BDD sizes are exponential in their formula sizes under all BDD variable orderings. For instance, it has been shown that ROBDDs of certain functions, such as integer multipliers, have exponential sizes irrespective of the ordering of variables [Bryant 1991]. Fortunately for many practical Boolean functions, there are variable orderings resulting in compact BDDs. This phenomenon can be explained intuitively by the fact that a BDD with n nodes may contain up to 2^n paths, which correspond to all possible truth assignments. ROBDD representations can be considerably more compact than SOP and POS representations.

Example 6.19 The odd parity function of Figure 6.7c is an example of function which requires $2n - 1$ nodes in an ROBDD representation but 2^{n-1} product terms in a minimum SOP representation.

We examine how well ROBDDs support Boolean reasoning. Complementing the function of an ROBDD can be done in constant time by simply interchanging the 0 and 1 terminal nodes.

In cofactoring an ROBDD with respect to a literal x_i (respectively $\neg x_i$), the variable x_i is effectively set to 1 (respectively 0) in the ROBDD. This is accomplished by determining all the nodes whose 0- or 1-child corresponds to any node υ with $\mathit{index}(\upsilon) = i$, and replacing their 0- or 1-child by $\mathit{then}(\upsilon)$ (respectively $\mathit{else}(\upsilon)$).

Example 6.20 Figure 6.8 illustrates a cofactor example, where the given ROBDD of (a) has been cofactored with respect to x_3 yielding the ROBDD of (b). Similarly, an ROBDD can be cofactored with respect to $\neg x_i$ by using $else(\upsilon)$ to replace all nodes υ with $index(\upsilon) = i$.

Binary Boolean operations, such as AND, OR, XOR, and so on, over two ROBDDs (under the same variable ordering) can be realized using the recursive BddApply operation. In the generic BddApply operation, ROBDDs D_1 and D_2 are combined as $D_1 \langle op \rangle D_2$ where $\langle op \rangle$ is a Boolean function of two arguments.

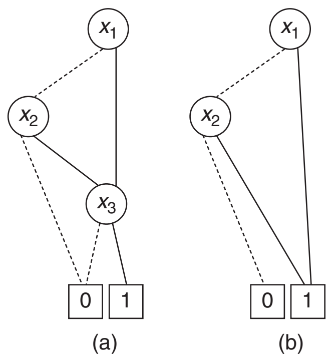


FIGURE 6.8

Cofactor example.

The result of the BddApply operation is another ROBDD. The operation can be customized by replacing $\langle op \rangle$ with a specific operator, *e.g.*, AND, OR, XOR, etc.

The algorithm proceeds from the roots of the two argument graphs downward, creating nodes in the resultant graph. It is based on the following recursion

$$f \langle op \rangle\, g = x_i \cdot (f_{x_i} \langle op \rangle\, g_{x_i}) + \neg x_i \cdot (f_{\neg x_i} \langle op \rangle\, g_{\neg x_i})$$

From an ROBDD perspective we have

$$f[v]\langle op \rangle g[w] = x_i \cdot (f[\mathit{then}(v)]\langle op \rangle g[\mathit{then}(w)]) + \neg x_i \cdot (f[\mathit{else}(v)]\langle op \rangle g[\mathit{else}(w)]) \quad (6.13)$$

where $f[v]$ and $g[w]$ are the Boolean functions rooted in the nodes v and w. There are several cases to consider.

1. If v and w are terminal nodes, we simply generate a terminal node u with $\mathit{value}(u) = \mathit{value}(v)\ \langle op \rangle\ \mathit{value}(w)$.
2. Else, if $\mathit{index}(v) = \mathit{index}(w) = i$, we follow Equation (6.13). Create node u with $\mathit{index}(u) = i$, and apply the algorithm recursively on $\mathit{else}(v)$ and $\mathit{else}(w)$ to generate $\mathit{else}(u)$ and on $\mathit{then}(v)$ and $\mathit{then}(w)$ to generate $\mathit{then}(u)$.
3. If $\mathit{index}(v) = i$ but $\mathit{index}(w) > i$, we create a node u having index i, and apply the algorithm recursively on $\mathit{else}(v)$ and w to generate $\mathit{else}(u)$ and on $\mathit{then}(v)$ and w to generate $\mathit{then}(u)$.
4. If $\mathit{index}(v) > i$ and $\mathit{index}(w) = i$ we create a node u having index i and apply the algorithm recursively on v and $\mathit{else}(w)$ to generate $\mathit{else}(u)$ and on v and $\mathit{then}(w)$ to generate $\mathit{then}(u)$.

Implementing the above algorithm directly results in an algorithm of exponential complexity in the number of input variables, since every call in which one of the arguments is a non-terminal node generates two recursive calls. Two refinements can be applied to reduce this complexity. Firstly, if the algorithm is applied to two nodes where one is a terminal node, then we can return the result based on some Boolean identities. For example, we have $f \vee 1 = 1$ and

$f \vee 0 = f$ for $\langle op \rangle$ = OR, $f \wedge 0 = 0$ and $f \wedge 1 = f$ for $\langle op \rangle$ = AND and $f \oplus 0 = f$ and $f \oplus 1 = \neg f$ for $\langle op \rangle$ = XOR. Secondly, more importantly the algorithm need not evaluate a given pair of nodes more than once. We can maintain a hash table containing entries of the form (v, w, u) indicating that the result of applying the algorithm to subgraphs with roots v and w was u. Before applying the algorithm to a pair of nodes we first check whether the table contains an entry for these two nodes. If so, we can immediately return the result. Otherwise we make the two recursive calls, and upon returning, add a new entry to the table. This refinement drops the time complexity to $O(|D_1| \cdot |D_2|)$, where $|D_1|$ and $|D_2|$ are the number of nodes in the two given graphs.

Example 6.21 We illustrate the BddApply algorithm with an example taken from [Bryant 1986]. The two ROBDDs to be operated on by an OR operator are shown in Figure 6.9a and 6.9b. Each node in the two ROBDDs has been assigned a unique label. This label could correspond to the labels generated during ROBDD reduction. The labels are required to maintain the table entries described immediately above.

The OBDD resulting from the OR of the two ROBDDs is shown in Figure 6.9c. First, we choose the pair of root nodes labeled `a1` and `b1`. We create a node with control variable x_1 and recursively apply the algorithm to the node pairs `a3, b1` and `a2, b1`. Since `a3` corresponds to the 1 terminal node, we can immediately return the 1 terminal node as the result of the OR. We must still compute the OR of the `a2, b1` node pair. This involves the computation of the OR of `a2, b3` and `a2, b2`, and so on. Note that `a3, b3` will appear as a node pair twice during the course of the algorithm.

Reducing the OBDD of Figure 6.9c results in the ROBDD of Figure 6.9d.

On the other hand, SAT and TAUTOLOGY checkings using BDDs are of constant time complexity due to the canonicity of BDDs. More specifically, SAT (respectively TAUTOLOGY) checking corresponds to checking if the BDD is not equal to the 0-terminal (respectively 1-terminal) node. Another application of BDDs is

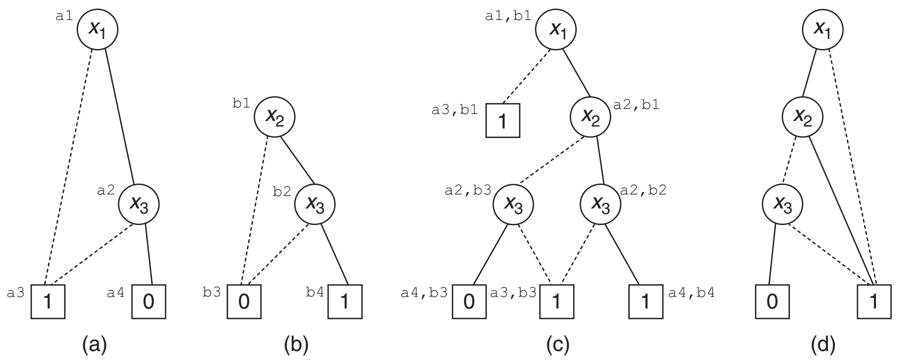


FIGURE 6.9

ROBDD examples for the BddApply operation: (a) ROBDD of function $f_1 = \neg x_1 \vee \neg x_3$. (b) ROBDD of function $f_2 = x_2 \wedge x_3$. (c) Intermediate OBDD after the BddApply operation for $f_1 \vee f_2$. (d) Final ROBDD of $f_1 \vee f_2$.

checking if two functions f_1 and f_2 are equivalent. The problem is of constant time complexity given that f_1 and f_2 are already represented in BDDs under the same variable ordering. Two BDDs (under the same variable ordering) represent the same function if and only if they have the same root node.

As all the above Boolean manipulations are efficiently solvable (*i.e.*, in polynomial time), BDDs are a powerful tool in logic synthesis and verification. We are by no means saying that Boolean reasoning is easy because the BDD size of a function can be exponential in the number of variables. Building the BDD itself risks exponential memory blow-up. Consequently BDD shifts the difficulty from Boolean reasoning to Boolean representation. Nevertheless once BDDs are built, Boolean manipulations can be done efficiently. In contrast, CNF-based SAT solving is memory efficient but risks exponential runtime penalty. Depending on problem instances and applications, the capability and capacity of state-of-the-art BDD packages vary. Just to give a rough idea, BDDs with hundreds of Boolean variables are still manageable in memory but not with thousands of variables. In contrast, state-of-the-art SAT solvers typically may solve in reasonable time the satisfiability problem of CNF formulas with up to tens of thousands of variables.

For the implementation of effective BDD packages, there are several important techniques. Firstly, complemented edges can be used to compactly represent a function as well as its complement [Madre 1988]. A complemented edge indicates that the function rooted in the node that the edge points to has be complemented. Introducing complemented edges does not destroy the canonicity of the ROBDD if the edges to be complemented are selected properly.

Example 6.22 The ROBDDs for a function with and without complemented edges are shown in Figure 6.10. Complemented edges are indicated by dots on them.

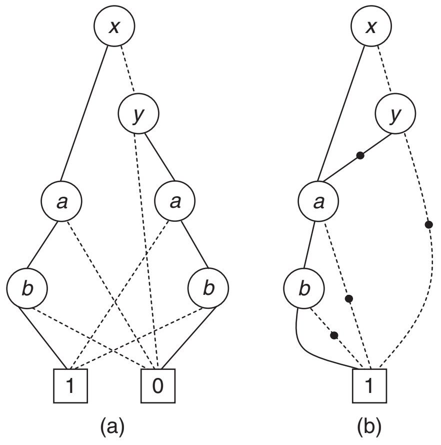


FIGURE 6.10

ROBDDs (a) without and (b) with complemented edges.

Secondly, a global unique table can be maintained wherein every node representing a unique function is given a unique label. Before creating a new node the table is checked to see if the function corresponding to this new node exists in the table. If not, the node is created, given a new label, and added to the unique table. If the function already exists, the node in the table corresponding to this function is returned.

Thirdly, dynamic variable ordering [Rudell 1993] can effectively reduce BDD sizes. A BDD variable ordering good for some functions may be bad for other functions. In the manipulation of ROBDD, new functions can be created. As a result, originally good variable ordering may become inadequate. Dynamic variable ordering provides a way of adjusting variable ordering to keep BDD sizes small. The description of an efficient implementation of an ROBDD package can be found in [Rudell 1990].

6.2.3.5 ***AIG***

An ***and-inverter graph*** (AIG) is a ***directed acyclic graph*** (DAG) $G = (V, E)$ consisting of vertices V representing AND2 (two-input AND) gates and directed edges $E \subseteq V \times V$ connecting gates. Inverters are denoted by markers on edges. Since operators $\{\wedge, \neg\}$ are functionally complete, any Boolean function can be represented in an AIG. Most Boolean functions can be represented compactly using AIGs.

The simple AIG data structure allows quick and cheap **structural hashing** among AIG nodes. Two AIG nodes with the same inputs under the same complementation conditions are merged (similar to the second reduction rule of ROBDD). Unlike ROBDD, however, the AIG representation is not canonical even when structural hashing is applied.

Example 6.23 Figure 6.11 shows the AIGs of function $f = a\neg cd + \neg b\neg cd$ without and with structural hashing in 6.11a and 6.11b, respectively.

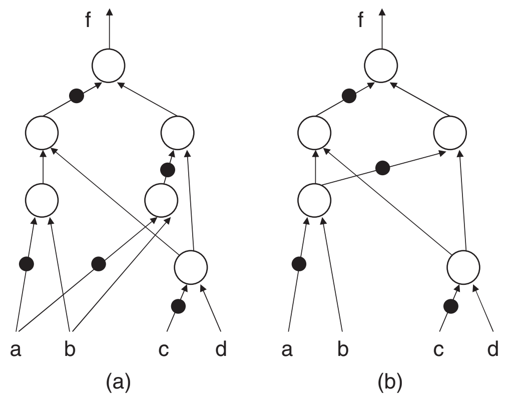


FIGURE 6.11

AIGs (a) without and (b) with structure hashing.

From the practical point of view, what make AIGs distinct from circuit netlists composed of AND2 gates and inverters are threefold:

1. Structural hashing — Structural hashing is applied during AIG construction; it propagates constants and ensures that each node is structurally unique. Accordingly AIGs are stored in a compact form.
2. Complemented edges — AIGs represent inverters as attributes on edges and thus do not require extra memory. Such complemented edges facilitate fast manipulation of AIGs and, in particular, lead to efficient structural hashing.
3. Regularity — As a result of regularity, memory management of an AIG package can be done by a simple customized memory manager which uses *fixed* amount of memory for each node (thanks to the fixed number of inputs of each node). By allocating memory for nodes in a topological order, we can optimize AIG *traversal*, which is repeatedly performed in many logic synthesis algorithms, in the same order. Experience suggests that many AIG-based applications have reduced **memory footprint** (namely, the amount of main memory used or referenced during a program's execution).

These features make a modern AIG package particularly efficient for Boolean function representation and reasoning.

We analyze the usefulness of AIGs for Boolean manipulation. The AND operation has a constant time complexity since the conjunction of two given AIGs can be done by adding an AIG node. The OR operation is essentially the same as the AND operation except for the markings on the input and output edges of the added AIG node, and thus is of constant time complexity. The complementation corresponds to marking an edge and is therefore of constant time complexity, too.

SAT and TAUTOLOGY checkings using AIGs are NP-complete and coNP-complete, respectively. When used as a Boolean reasoning engine, an AIG package can be viewed as a solver performing satisfiability checking over circuits rather than over CNF formulas, and is similar to ***automatic test pattern generation*** (ATPG).

AIGs can also be used in verification applications, such as equivalence checking and even model checking. For instance, checking if two given AIGs under comparison are functionally equivalent can be reduced to TAUTOLOGY (SAT) checking by adding an XNOR (XOR) gate, which can be expressed in terms of AND2 and INV gates, with its two-inputs feeded in by the outputs of the two AIGs. The two AIGs are equivalent if and only if the output of the XNOR (XOR) gate is tautological (unsatisfiable). Hence the equivalence checking problem is coNP-complete. When it comes to synthesis, AIGs are used in multilevel logic minimization and technology mapping. In the academic system ABC [ABC 2005], AIGs are used as a unifying data structure for both logic synthesis and verification.

A new binary format called AIGER [Biere 2007] was recently proposed to enable compact representation of AIGs in files and memory. With memory requirements of about three bytes per AIG node, AIGER has become a standard representation for circuit-based problems in *SAT Competitions* and *Hardware Model Checking Competitions*, organized annually as satellite events of *International Conference on Theory and Applications of Satisfiability Testing* and *International Conference on Computer Aided Verification*, respectively.

6.2.3.6 ***Boolean network***

A (combinational) logic circuit can be represented with a **Boolean network**, a directed graph $G = (V, E)$ with nodes V and directed edges E. Every node $i \in V$ is associated with a logic function f_i and a Boolean variable x_i, called the **output variable** of node i, representing the output of function f_i. Hence the relation between variable x_i and function f_i obeys $(x_i \equiv f_i)$. Every edge $(i, j) \in E$ connecting from node i to node j signifies that variable x_i is an input to function f_j, and we call that node i (j) is a **fanin (fanout)** of node j (i). That is, variable x_i syntactically appears in the Boolean expression of f_j as x_i or $\neg x_i$. We say x_i is a **(structural) support variable** of f_j . If, in addition, the Boolean difference $\frac{\partial f}{\partial x_i}$ is satisfiable, then x_i is a **functional support variable** of f_j, as defined previously.

A node i without any fanin is a **primary input** and its associated logic function is x_i, *i.e.*, identical to its output variable. Moreover, a subset of V is specified as **primary outputs**. Among the variables of node outputs, we say those of the primary inputs are the **primary input variables**, those of the primary outputs are the **primary output variables**, and others are **local** (or **intermediate**) **variables**.

The sets of fanins and fanouts of node i are denoted as *FI*(i) and *FO*(i), respectively. The **transitive fanins** *TFI*(i) and **transitive fanouts** *TFO*(i) of a node i are defined recursively as

$$TFI(i) = \{k \in V | k = i, \text{or } k \in FI(j) \text{ for } j \in TFI(i)\}$$

and

$$TFO(i) = \{k \in V | k = i, \text{or } k \in FO(j) \text{ for } j \in TFO(i)\}$$

respectively.

A (combinational) Boolean network can be acyclic or cyclic. Any acyclic circuit must behave combinationally because no internal states can be maintained and the output only depends on the current input assignment, rather than on the prior input assignments; a cyclic circuit, in contrast, may possibly exhibit combinational behavior as well [Kautz 1970]. Because the existence of cyclic structures substantially complicates the analysis and optimization of logic design, most logic synthesis systems assume that combinational circuits are acyclic. In the sequel we shall assume that a Boolean network is acyclic. Therefore, $TFI(i) \cap TFO(i) = \{i\}$.

A node function f_i is a **local function**, in the sense that it is in terms of the output variables of the immediate fanins of node i. The function of node i can be alternatively expressed purely in terms of the primary input variables. In this case, it is called the **global function** g_i of node i. Function g_i can be derived from f_i by recursively substituting f_j for y_j, for $j \in TFI(i)$, until no further substitution is possible. This substitution process is guaranteed to terminate because of the assumption of *acyclic* combinational Boolean networks.

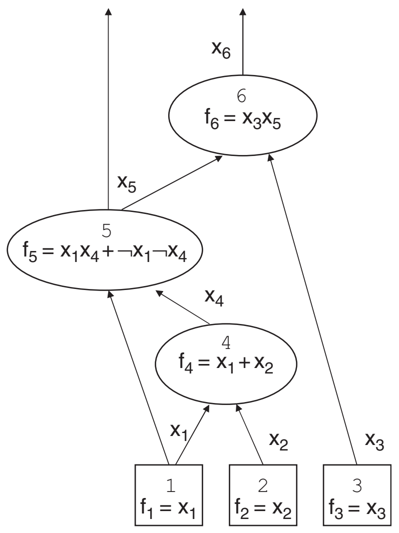


FIGURE 6.12

Boolean network example.

Example 6.24 Figure 6.12 shows a Boolean network example, where nodes 1, 2 and 3 are the primary inputs, and nodes 5 and 6 are the primary outputs. A local function f_i is shown in the corresponding node i. The global function of node i can be obtained by either recursive composition or quantification. For instance, the global function

$$g_5 = x_1(x_1 + x_2) + \neg x_1 \neg(x_1 + x_2)$$

by recursive composition, or equivalently

$$g_5 = \exists x_4.(x_1x_4 + \neg x_1 \neg x_4)(x_4 \equiv (x_1 + x_2))$$

by quantification.

As for the implementation issue, how to represent the logic function f_i of a node i in a Boolean network is a matter of choice. Our previously mentioned data structures, such as the truth table, SOP, BDD, AIG, and Boolean network representations, can be adopted. Compared with AIGs, generic Boolean networks may lack special structures to be exploited for effective Boolean reasoning. They however are suitable for generic circuit representation.

6.2.4 Boolean representation conversion

6.2.4.1 *CNF vs. DNF*

SOP-to-POS and POS-to-SOP conversions can be achieved by applying double complements. By applying De Morgan's Law, an SOP (a POS) formula φ becomes a POS (an SOP) one φ' after the first complement. We can then convert the POS (SOP) formula φ' to an SOP (a POS) one φ'' by the distributive law. Finally, applying De Morgan's Law again for the second complement, we convert the SOP (POS) formula φ'' to a POS (an SOP) one φ'''. Note that the conversions may suffer from an exponential blow-up in formula sizes due to the intermediate step of applying the distributive law.

Example 6.25 The $2n$-input Achilles heel function $(x_1 + y_1)(x_2 + y_2) \cdots (x_n + y_n)$ has 2^n product terms in an SOP representation but has a linear-sized POS representation.

There exist Boolean functions whose SOP- and POS-formula sizes are inevitably exponential in the number of input variables. For example, the n-input odd parity function $(x_1 \oplus x_2 \cdots \oplus x_n)$ has 2^{n-1} product terms in an SOP representation and is equally large in a POS representation. As another example, integer multiplication over n-bit operands, comparison of two n-bit operands, and addition and subtraction of n-bit operands all have SOP and POS realizations that grow exponentially with n.

An interesting application of Boolean representation conversion is on Boolean reasoning. Recall that SAT (respectively TAUTOLOGY) checking is trivial for DNF (respectively CNF) formulas. If we are interested in knowing the satisfiability of a CNF formula, we may covert it into DNF and then check the satisfiability of the DNF formula, which is a constant time checking. Similarly we may check the tautology of a DNF formula by converting it into CNF. The hardness of Boolean reasoning, of course, is shifted to the representation conversion process. Another application of Boolean representation conversion is on quantifier elimination for QBFs. Observe that the universal (respectively existential) quantification is easy for CNF (respectively DNF) formulas. The QBF $\forall x_i.\varphi(\boldsymbol{x})$ with $\varphi(\boldsymbol{x})$ in CNF equals the induced quantifier-free Boolean formula of removing every appearance of literals x_i and $\neg x_i$ in $\varphi(\boldsymbol{x})$; similarly the QBF $\exists x_i.\varphi(\boldsymbol{x})$ with $\varphi(\boldsymbol{x})$ in DNF equals the induced quantifier-free Boolean formula of removing every appearance of literals x_i and $\neg x_i$ in $\varphi(\boldsymbol{x})$. It is thus of linear time complexity. Therefore given a QBF, we can convert the formula back and forth between CNF and DNF to eliminate quantifiers. As a consequence, any SOP-POS converter can be used as a Boolean reasoning engine and QBF solver.

Example 6.26 The QBF

$$\forall a.(a + b + \neg c)(a + \neg b + c)(\neg a + b + c)$$

equals the quantifier-free Boolean formula

$$(b + \neg c)(\neg b + c)(b + c)$$

The QBF

$$\exists a.(ab\neg c + a\neg bc + \neg abc)$$

equals the quantifier-free Boolean formula

$$b\neg c + \neg bc + bc$$

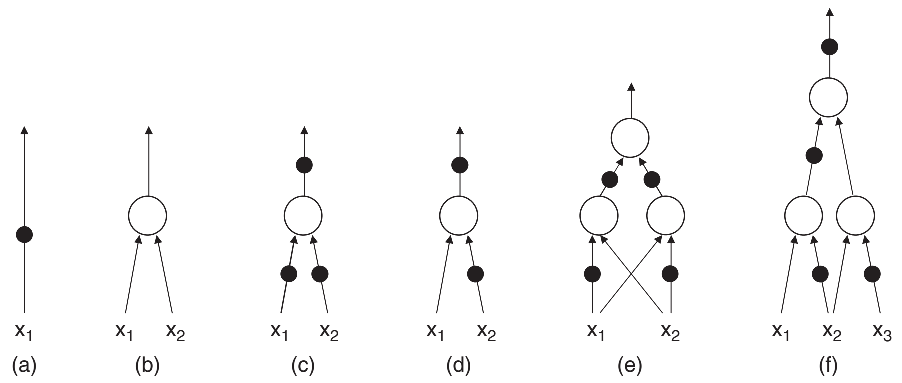


FIGURE 6.13

(a) AIG of $\neg x_1$. (b) AIG of $(x_1 \wedge x_2)$. (c) AIG of $(x_1 \vee x_2)$. (d) AIG of $(x_1 \Rightarrow x_2)$. (e) AIG of $(x_1 \Leftrightarrow x_2)$. (f) AIG of $(x_1 \wedge \neg x_2) \vee (x_2 \Rightarrow x_3)$.

6.2.4.2 *Boolean formula vs. circuit*

A Boolean formula φ can be translated into a circuit, *e.g.*, an AIG, in linear time. The translation can be done by following the inductive construction of φ with the rules of Equation (6.1).

Example 6.27 Figure 6.13a-e show the AIGs of $\neg x_1$, $x_1 \wedge x_2$, $x_1 \vee x_2$, $x_1 \Rightarrow x_2$, and $x_1 \Leftrightarrow x_2$. They form the templates of the basic formation rules of Equation (6.1). Given an arbitrary Boolean formula, its AIG can be built from these templates, *e.g.*, the AIG of $(x_1 \wedge \neg x_2) \vee (x_2 \Rightarrow x_3)$ is shown in Figure 6.13f.

Any (combinational) circuit, on the other hand, represents some Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$, which can be specified with a Boolean formula. Recall Example 6.24, which shows how an output function of a circuit can be obtained.

6.2.4.3 *BDD vs. Boolean network*

A two-input multiplexor is a switch with two data inputs i_0, i_1, one control input c, and one output o, with $o = i_0$ if $c = 0$ and $o = i_1$ if $c = 1$. Because a non-terminal node in a BDD can be seen as a two-input multiplexor and BDDs are universal for functional representation, any Boolean function can be implemented using a circuit whose only constituent gates are two-input multiplexors. Translating a BDD to a multiplexor-based Boolean network is a straightforward process by substituting every BDD node with a multiplexor, and can be accomplished in time linear in the size of the BDD.

Given a Boolean network, the ROBDD of a primary output function in terms of the primary input variables can be constructed. A naïve approach is to build an OBDD representing the global function of the Boolean network and then reduce it. Rather, a more effective way is to traverse the circuit from primary inputs to primary outputs using a series of Boolean manipulations over ROBDDs based on node

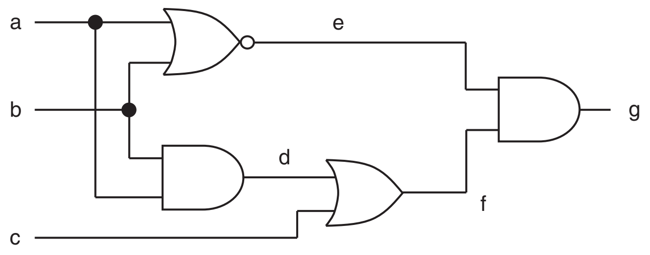


FIGURE 6.14

Multilevel circuit.

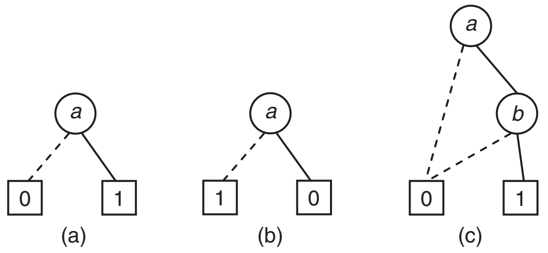


FIGURE 6.15

(a) ROBDD for primary input a. (b) ROBDD for $\neg a$. (c) ROBDDs for $a \wedge b$.

functions. For a primary input, its ROBDD is a graph with a single non-terminal node and two terminal nodes. For a functional node, its ROBDD can be constructed using a series of complement and/or BddApply operations.

Example 6.28 Consider the circuit of Figure 6.14. The ROBDD for primary input a is shown in Figure 6.15a. Similarly, the ROBDD for primary input b will have one node with control variable b with a 0-child (1-child) corresponding to the 0 (1) terminal node. The ROBDD for $\neg a$ is shown in Figure 6.15b. We can create the ROBDD for signal d by performing an AND operation on the ROBDDs for the primary inputs a and b. This ROBDD is shown in Figure 6.15c. We can create the ROBDD for signal f by performing an OR operation on the ROBDD for signal d and the ROBDD for the primary input c.

As an application, ROBDD-based circuit equivalence checking can be achieved by the conversion from Boolean networks to ROBDDs. Since ROBDDs are a canonical representation of Boolean functions, in order to check two circuits C_1 and C_2 for equivalence, we can use the following method.

1. Choose an ordering for the primary inputs of the circuits.
2. Create ROBDDs for the primary outputs of the two circuits.
3. Check if the ROBDDs are isomorphic. If so, the circuits are equivalent. If not, the circuits are not equivalent.

In order to check two ROBDDs for equivalence, we can use the canonicity property of ROBDDs and perform a linear-time graph isomorphism check as per Definition 6.2. Notice that any ordering will suffce, as long as the same ordering is chosen for both circuits. However, the size of the ROBDDs created is strongly dependent on the ordering chosen.

6.2.5 Isomorphism between sets and characteristic functions

A very profound application of Boolean functions is the concept of characteristic functions in representing sets. It is a very important idea leading to a leap in capacity of many logic synthesis and verification algorithms. A **characteristic function** is a (total) function $\chi_A : U \to \mathbb{B}$, where U is a finite set often in the form of $\mathbb{B}^n$ for some n, such that $\chi_A(e) = 1$ if and only if $e \in A$, that is, the onset of χ_A equals A. It serves as a predicate indicating the membership property. In other words, the function χ_A answers a query, whether an element $e \in U$ is in $A \subseteq U$. Essentially, any finite set $A \subseteq U$ can be represented with a characteristic function χ_A. Thereby set operations (*e.g.,* intersection $\cap$, union $\cup$, and complement) over sets are in effect Boolean operations (*e.g.,* conjunction $\wedge$, disjunction $\vee$, and negation $\neg$, respectively) over characteristic functions. Note that constant functions 0 and 1 are characteristic functions of the empty set $\emptyset$ and universal set U, respectively. Some applications of characteristic functions are given below.

Incompletely Specified Function as Characteristic Function. To represent an incompletely specified Boolean function $I : \mathbb{B}^n \to \{0, 1, -\}$, three characteristic functions r, f, d can be used to represent its onset, offset and dcset, respectively. That is, for a minterm $m \in \mathbb{B}^n$,

$$\begin{array}{rcll} r(m) & = & 1 \text{ if and only if } I(m) & = & 0 \\ f(m) & = & 1 \text{ if and only if } I(m) & = & 1, \text{ and} \\ d(m) & = & 1 \text{ if and only if } I(m) & = & - \end{array}$$

As the three sets form a **partition** on $\mathbb{B}^n$, *i.e.*, the three sets are pairwise disjoint and union to $\mathbb{B}^n$, two characteristic functions are suffcient in representing an incompletely specified function. However, even so three characteristic functions are often used for the sake of convenience in Boolean manipulation.

Boolean Relation as Characteristic Function. A relation is more general than a function as it allows one-to-many mappings, which are prohibited in a function. A Boolean relation can be treated as a set of input-output mapping pairs, and thus can be represented by a characteristic function.

Example 6.29 Given a set of Boolean functions $f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})$, they can be converted into a Boolean relation

$$R(\mathbf{x}, \mathbf{y}) = \bigwedge_{i=1}^{m} (y_i \equiv f_i(\mathbf{x}))$$

by introducing a vector of output variables $\boldsymbol{y} = (y_1, \ldots, y_m)$. For truth assignments $a \in \mathbb{B}^n$ and $b \in \mathbb{B}^m$ on variables $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, relation $R(a, b)$ valuates to true if and only if the ith bit of b equals the value of $f_i(a)$ for $i = 1, \ldots, m$. In other words, $R(a, b) = 1$ if and only if a and b are consistent assignments under the mapping of functions $f_1, \ldots, f_m$.

Circuit Consistency Condition as Characteristic Function. The consistency condition imposed by a circuit can be converted into a Boolean formula, in particular, a CNF formula by Tseitin's procedure [Tseitin 1970], where every gate of a circuit translates into a set of clauses of fixed sizes and, further, the CNF formula of a circuit is the conjunction of the clauses of all gates. Therefore the conversion is done in time linear to the circuit size.

Example 6.30 The CNF formula of the consistency condition imposed by an AND2 gate with inputs a, b and output c is

$$\begin{aligned} & (a \wedge b) \Leftrightarrow c \\ = \; & ((a \wedge b) \Rightarrow c)(c \Rightarrow (a \wedge b)) \\ = \; & (\neg a \vee \neg b \vee c)(\neg c \vee (a \wedge b)) \\ = \; & (\neg a \vee \neg b \vee c)(\neg c \vee a)(\neg c \vee b) \end{aligned}$$

Using the above three clauses for an AND2 gate, we can obtain the CNF formula

$$\begin{aligned} & (\neg x_1 \vee x_2 \vee x_4)(\neg x_4 \vee x_1)(\neg x_4 \vee x_2) \wedge \\ & (\neg x_2 \vee x_3 \vee x_5)(\neg x_5 \vee x_2)(\neg x_5 \vee \neg x_3) \wedge \\ & (x_4 \vee \neg x_5 \vee x_6)(\neg x_6 \vee \neg x_4)(\neg x_6 \vee x_5) \wedge \\ & (x_6 \vee x_7)(\neg x_6 \vee \neg x_7) \end{aligned}$$

for the consistency condition imposed by the AIG of Figure 6.16. Note that the first three clauses correspond to the AIG node of x_4, the second three clauses correspond to the AIG node of x_5, the third three clauses correspond to the AIG node of x_6, and the last two clauses correspond to the inversion of x_6 for x_7. Hence for given an AIG, the so-constructed CNF formula is of size linear in the number of nodes.

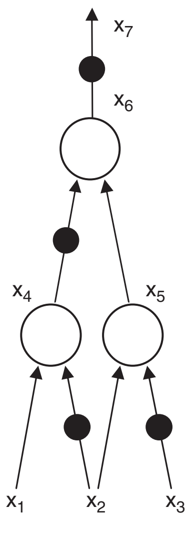


FIGURE 6.16

AIG example for CNF conversion.

Note that the function represented by the so-constructed CNF formula is not the same as the primary output functions of a given circuit. A circuit and its CNF formula are equivalent only in the sense that the CNF formula is true under a truth assignment if and only if the truth assignment is consistent in the circuit. A circuit implements some Boolean functions whereas such a CNF formula represents a Boolean relation.

At first glance, Tseitin's linear-time translation from circuits to CNF formulas seems contradictory to the exponential cost of the SOP-to-POS conversion because we may covert in linear time any SOP formula to an AIG and then further convert the AIG to a CNF formula by Tseitin's procedure. This paradox can be clarified by observing that in Tseitin's conversion new extra variables are present in the resultant POS/CNF formula. It differs from the previous SOP-to-POS conversion where no new variables are created. In fact, a Boolean relation derived from the new conversion reduces to a Boolean function as derived from the old one when the intermediate variables (those other than the primary input and output variables) are existentially quantified out and further a positive co-factor is performed on the Boolean relation with respect to the primary output variable. The existential quantification and conversion back to a POS formula, however, may result in exponential blow-up in formula sizes.

Example 6.31 Figure 6.17 shows the AIG of function $f = x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$. By Tseitin's conversion, the CNF formula is of size linear to n due to the allowance of intermediate variables. Without intermediate variables, the POS representation of f must have 2^n clauses.

Set Manipulation as Boolean Manipulation. By dealing with characteristic functions, we are able to manipulate sets of elements simultaneously rather than manipulate individual elements separately. For instance, the intersection of

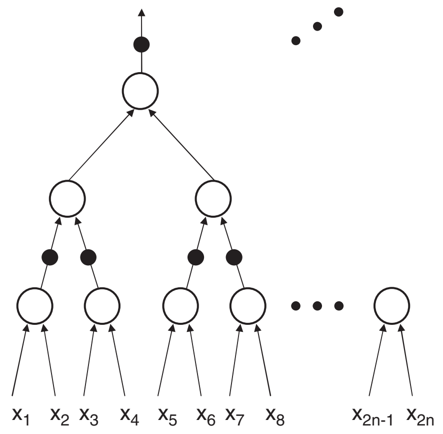


FIGURE 6.17

AIG of function $f = x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$.

two sets A and B can be done by performing $\chi_A \wedge \chi_B$ instead of examining, for every element $e \in A$, whether e is in B as well. It leads to substantial improvements to many logic synthesis and verification algorithms. Such approaches that manipulate sets of objects simultaneously are known as **(implicit) symbolic algorithms**, in contrast to the traditional **(explicit) enumerative algorithms** (which enumerate individual objects separately).

Example 6.32 Let set U be the universe $\{0, 1, 2, 3, 4, 5, 6, 7\}$, set $A \subseteq U$ be $\{0, 1, 2, 4\}$, and set $B \subseteq U$ be $\{2, 3, 4, 6\}$. Consider the binary encoding with Boolean variables x_1, x_2, and x_3 such that element 0 is encoded as $\neg x_1 \neg x_2 \neg x_3$, 1 as $\neg x_1 \neg x_2 x_3$, 2 as $\neg x_1 x_2 \neg x_3$, 3 as $\neg x_1 x_2 x_3$, 4 as $x_1 \neg x_2 \neg x_3$, 5 as $x_1 \neg x_2 x_3$, 6 as $x_1 x_2 \neg x_3$, and 7 as $x_1 x_2 x_3$. Then the characteristic functions of these sets with respect to the binary encoding are

$$\begin{aligned} \chi_U &= 1 \\ \chi_A &= \neg x_1 \neg x_2 + \neg x_1 \neg x_3 + \neg x_2 \neg x_3, \text{ and} \\ \chi_B &= \neg x_1 x_2 + x_1 \neg x_3 \end{aligned}$$

It can be checked that formula $\neg \chi_A$ corresponds to the characteristic function of the set $U \backslash A$, formula $\chi_A \wedge \chi_B$ corresponds to that of $A \cap B$, and formula $\chi_A \vee \chi_B$ corresponds to that of $A \cup B$.

Example 6.33 Image and pre-image computations are key operations in logic synthesis and formal verification. The **image** of $A \subseteq \mathbb{B}^n$ under the functional vector $f = (f_1, \ldots, f_m)$ is the set $\{q \in \mathbb{B}^m \mid q = \boldsymbol{f}(p), p \in A\}$. The characteristic function of the image is

$$Img_f(A) = \exists \boldsymbol{x}. \bigwedge_{i=1}^{m} (y_i \equiv f_i(\boldsymbol{x})) \wedge \chi_A(\boldsymbol{x})$$

which refers to the newly introduced $\boldsymbol{y}$ variables taking on the function values. In contrast, the **pre-image** of $B \subseteq \mathbb{B}^m$ under the functional vector $\boldsymbol{f} = \{f_1, \ldots, f_m\}$ is the set $\{p \in \mathbb{B}^n \mid q = \boldsymbol{f}(p), q \in B\}$. The characteristic function of the pre-image is

$$PreImg_f(B) = \exists \boldsymbol{y}. \bigwedge_{i=1}^{m} (y_i \equiv f_i(\boldsymbol{x})) \wedge \chi_B(\boldsymbol{y})$$

which refers to the $\boldsymbol{x}$ variables only.

6.2.6 **Boolean reasoning engines**

Among the introduced data structures, BDD packages and SAT solvers are the most widely used Boolean reasoning engines. They are extensively used in various ***symbolic***, or called ***implicit***, algorithms, such as image computation, don't care computation, state reachability analysis, and so on. Any Boolean reasoning engine can be more or less used in developing symbolic algorithms. In the sequel when a computational task is expressed in terms of a QBF, we should be aware that its computation is already achievable by Boolean manipulation using a BDD package.

Although BDD-based algorithms and symbolic algorithms were once almost synonymous in the 1990s, recently other data structures were developed as alternatives to BDDs. Due to the capacity limit of BDDs, more and more symbolic algorithms are based on other data structures. Notably, Boolean reasoning engines using SAT and AIGs, for instance, are gaining in popularity in hardware synthesis and verification. Moreover, hybrid Boolean reasoning engines combining complementary data structures may become important tools. In fact, combinational equivalence checking of multi-million gate designs has been demonstrated in an industrial setting through such hybrid solvers combining BDD and AIG [Kuehlmann 1997].

6.3 COMBINATIONAL LOGIC MINIMIZATION

Logic synthesis is typically divided into two phases: **technology independent optimization** and **technology dependent optimization**. The former aims at simplifying Boolean expressions and logic netlist structures regardless of the target technology node for manufacturing, whereas the latter aims at optimizing circuits under the target implementation technology. This divide-and-conquer separation is often beneficial in orthogonalizing various design concerns. Simplified Boolean expressions are often good for optimization with respect to the target implementation technology. Also it allows a designer to migrate a design from one technology node to another without substantial re-optimization. Our study will begin with the first phase, and then proceed to the second one in Section 6.4.

In technology independent optimization, combinational logic minimization consists of two-level and multilevel logic minimization. Two-level logic minimization is a relatively simple and well-studied subject in both theory and practice. As a multilevel logic netlist can be seen as a network of two-level logic components, the results of two-level minimization are in part applicable to multilevel minimization. Not only optimized two-level SOP representations can be used as a starting point for multilevel synthesis, but two-level minimization techniques can also be used in minimizing multilevel netlists. Hence we delve into two-level logic minimization before considering the multilevel counterpart.

6.3.1 Two-level logic minimization

There are a variety of two-level logic implementations. The most common one is the SOP implementation, where the first level of logic corresponds to AND gates and the second level to OR gates. NOR-NOR structures, NAND-NAND structures, AND-XOR structures, and OR-AND structures are also possible.

Example 6.34 The function of Figure 6.18a can be reexpressed in POS form and implemented as the circuit shown in Figure 6.18c. An SOP implementation can be directly converted into an equivalent NAND-NAND implementation by replacing all the AND gates and OR gates by NAND gates. A NAND-NAND implementation of the function of Figure 6.18a is shown

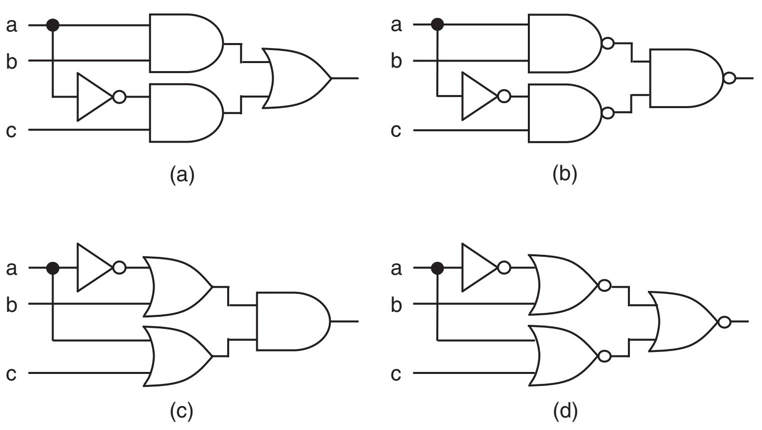


FIGURE 6.18

Two-level logic implementations.

in Figure 6.18b. Similarly, a POS implementation can be directly converted into a NOR-NOR implementation as shown in Figure 6.18d.

Two-level logic is typically implemented as a ***programmable logic array*** (PLA) [Fleisher 1975] in a NOR-NOR form followed by inverters at the outputs. PLAs have the advantage of being very structured and are therefore amenable to automated logic and layout synthesis. Even though PLAs are no longer a popular IC implementation style, they can be an important ingredient in modern system designs because their regular structures [Mo 2004] provide a solution to alleviate the infamous process variation problem of IC manufacturing in the nanometer regime.

6.3.1.1 ***PLA implementation vs. SOP minimization***

Despite the fact that many regular functions have a minimum two-level logic representation whose size grows exponentially with the number of inputs to the function (*e.g.,* parity functions and adders), two-level logic circuits can efficiently implement control logic.

The hardware cost of a PLA implementing some SOP formula is directly reflected in the formula. The number of literals (respectively product terms) of the formula corresponds to the number of transistors (respectively product lines) of the PLA. Therefore, minimizing an SOP expression not only reduces PLA area cost, but also improves circuit performance due to the reduction in capacitive loads.

Example 6.35 An NMOS PLA is shown in Figure 6.19a, whose output marked f implements the logic function of Figure 6.18. Note that while the input plane and output plane are both NOR-planes, we have inverters at the outputs. An SOP representation can be directly mapped to a NOR-NOR PLA with output inverters by complementing each literal in the input plane. The function $f = a \cdot b + \neg a \cdot c$ has been implemented as

$$\neg(\neg(\neg(\neg a + \neg b) + \neg(a + \neg c)))$$

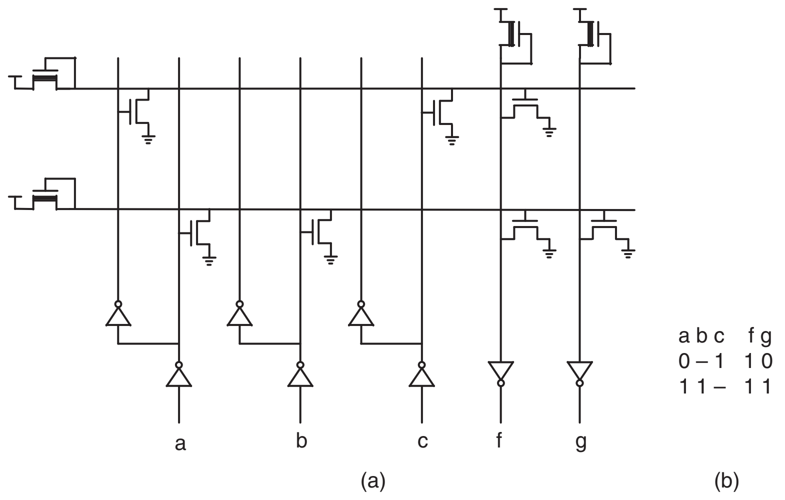


FIGURE 6.19

(a) Programmable logic array. (b) Multiple-output cover.

PLAs can implement multiple-output functions that share product terms across outputs as shown in Figure 6.19a. The multiple-output cover is represented as shown in Figure 6.19b. The two outputs share the cube $a \cdot b$ in their onsets. Therefore, in the PLA of Figure 6.19a the first row from the bottom feeds transistors in both columns in the output plane. The number of columns in a PLA equals two times the number of inputs plus the number of outputs, the number of rows equals the number of product terms in the cover, the number of transistors in the input plane equals the number of "1" or "0" literals in the input part of the multiple-output cover, and the number of transistors in the output plane equals the number of 1's in the output part of the multiple-output cover.

6.3.1.2 *Terminology*

We define terminology and notation used for two-level logic minimization.

As a notational convention, we write a cube (*i.e.*, a product term) c in a bit-vector form $c = [c_1 \ldots c_n]$, where c_i is "0" if the i^{th} variable x_i appears complemented in c, c_i is "1" if variable x_i appears uncomplemented in c, and c_i is "–" if variable x_i does not appear in c.

Example 6.36 A cube $c = x_1 \neg x_2$ in the Boolean space spanned by variables x_1, x_2, x_3 can be represented as [10–].

For multi-output functions, the notion of cubes is slightly generalized. A cube of a Boolean function f with n inputs and m outputs is written as $c = [c_1 \cdots c_n | c_{n+1} \cdots c_{n+m}]$, which consists of the *input part* with c_i's for $1 \leq i \leq n$ and *output part* with c_i's for $n+1 \leq i \leq n+m$. In the input part, c_i is defined the same as before; in the output part, c_i is "0," "1," and "–" if the input part of c belongs to the offset, onset, and dcset, respectively, of the $(i - n)^{th}$ output of f. For single-output functions, we may not write the $(n+1)^{st}$ bit of the cube if the function is fully specified.

A **minterm**, defined in Section 6.2, corresponds to a cube in which every variable of a Boolean space appears. Minterms and cubes may be used to represent the values of a set of input variables, *e.g.*, $x\neg yz$ is shorthand for $x = 1, y = 0$, and $z = 1$. Therefore, there is a natural correspondence between an input assignment and a vertex in the Boolean n-space. This correspondence may be extended to cubes where absent variables are assumed to be unassigned. Thus, if a circuit C has inputs v, w, x, y, and z then applying the cube $x\neg yz$ to C is shorthand for applying $v = X, w = X, x = 1, y = 0$, and $z = 1$, where "X" is used to denote an unknown value.

A cube q *contains* another cube r if the literals in the input part of cube q are a subset of the literals in the input part of cube r and the outputs in the output part of q are a superset of the outputs in the output part of cube r. In bit-vector notation, the cube [0−|1] of a two-input, single-output function contains the cube [00|1]. Similarly, the cube [0−|11] of a two-input, two-output function contains the cube [0−|10]. A cube is said to be contained by a cover if every minterm contained by the cube is contained by some cube in the cover. For example, the cover {00−−, −1−1} contains the cube [0−−1].

If a cube q contains only onset and dcset vertices of a Boolean function f, then q is called an **implicant** of f. A **prime implicant** (or **prime**) of f is an implicant which is not contained by any other implicant of f and which is not entirely contained in the dcset of f. An alternate operational definition, which is crucial in ESPRESSO, of a prime implicant is as follows. An implicant is prime if no 0- or 1-literal can be "raised" (to include more minterms) to a "−" without resulting in the implicant intersecting the offset of any component of the multiple-output function. For instance, a cube [111] of a three-input, single-output function would be a prime cube if each of [11−], [1−1] and [−11] intersected the offset. A literal in a cube is said to be prime if raising that particular literal to a "−" results in a cube that intersects the offset. Thus, [110] may not be a prime cube of a function f because [11−] is an implicant of f, but the first two literals may be prime in the implicant [110] because [−10] and [1−0] intersect the offset of f. All the literals contained in a cube have to be prime in order for the cube to be prime.

An **essential prime implicant** (or **essential prime**) is a prime implicant which includes one or more onset vertices which are not included in any other prime implicant. These vertices are termed **essential vertices**. An **optional prime implicant** is a prime implicant for which all vertices are included in other prime implicants.

A minimal cover for a function f is generated by selecting all of the essential prime implicants and a minimal set of optional prime implicants such that all vertices in the onset of f are included in the cover.

Example 6.37 For the example in Figure 6.1b, there are three essential prime implicants and no optional prime implicants. The minimal cover would be $f = \neg x_3 + \neg x_1 x_2 + x_1 \neg x_2$.

A **relatively essential vertex** of a cube q in a cover C is a vertex in the onset that is contained by q and is not contained in any other cube in C.

Example 6.38 In Figure 6.1b, $x_1\neg x_2x_3$ is a relatively essential vertex of the cube $x_1\neg x_2$, while the other vertex in this cube, $x_1\neg x_2\neg x_3$, is not a relatively essential vertex since it is also contained in the cube $\neg x_3$.

A two-input, two-output function can also be represented as a multiple-output cover, with cubes that have input as well as output parts.

Example 6.39 The two-output function $F = \{11|01, 00|10, 10|11\}$ has two cubes in each of its components F_1 and F_2. If the inputs are a and b, then F_1 can be represented as $\neg a\neg b + a\neg b$, and F_2 is $ab + a\neg b$. The cube $a\neg b$ is shared by F_1 and F_2, because its output part indicates that it belongs to both their onsets.

In order to keep cover sizes small, it is desirable to ensure some form of minimality for the cover. An easily satisfiable property is that no cube c of a cover contains another cube d of the cover. Such a cover is minimal with respect to **single cube containment**.

An implicant in a cover is **irredundant** if it contains an essential or a relatively essential vertex. Thus, removing the implicant changes the functionality of the cover. Else it is **redundant** and can be safely removed from the cover. A cover is prime if each of the implicants in the cover is prime. A cover is irreundant if each of the implicants is irredundant. The definitions apply to both completely specified and incompletely specified functions.

6.3.2 SOP minimization

Two-level Boolean minimization is used to find an SOP representation for a Boolean function that is optimum according to a given cost function. The typical cost functions used are the number of product terms, the number of literals, or a combination of both.

With any of these cost functions, the problem of two-level minimization contains the subproblem of finding the solution of a minimum covering problem which has been shown to be NP-complete [Garey 1979]. Nevertheless, sophisticated exact minimizers (*e.g.,* [Dagenais 1986; Rudell 1987) have been developed whose average-case behavior for most commonly encountered functions is acceptable. Furthermore, heuristic minimization methods exist (*e.g.,* [Hong 1974; Brayton 1984]) which have been shown to produce results that are close to the minimum within reasonable amounts of time, even for large Boolean functions.

Two-level Boolean minimization for a given function consists of two steps:

1. generating the set of prime implicants, and
2. selecting a minimum set of prime implicants to cover all onset minterms.

6.3.2.1 *The Quine-McCluskey method*

The first algorithmic method proposed for two-level minimization is the Quine-McCluskey method [McCluskey 1956], which follows the two steps outlined above.

Prime Implicant Generation. The set of prime implicants can be generated by iteratively merging two cubes which differ in exactly one position, where one is of literal x and the other is of literal $\neg x$ assuming variable x is the corresponding variable in the position. For instance, two cubes $c_1 = [00-1]$ and $c_2 = [01-1]$ can be merged as $[0--1]$. This merging process continues until no more merging is possible. Initially all onset and dcset minterms are the cubes to start with. Upon termination, a maximal cube (not contained by every other cube) is a prime implicant provided that it is not entirely contained by the dcset.

Example 6.40 Consider the completely specified Boolean function shown in Figure 6.20a. It has been represented as a list of minterms. Each minterm has an associated decimal value obtained by converting the binary number represented by the minterm into a decimal number; for instance the value of 0000 is 0 and that of 1100 is 12. The cubes generated by merging the pairs of cubes are shown in Figure 6.20b and 6.20c. We have five prime implicants, marked as **A, B, C, D**, and **E**, for the function in this example.

Prime Implicant Table. A **prime implicant table** is a table with rows indexed by onset minterms and columns indexed by prime implicants. An entry at position (i, j) in the table is marked "X" if prime implicant j contains onset minterm i.

Example 6.41 Figure 6.21 shows the prime implicant table of the previous example.

Since we want a minimum set of prime implicants that covers all the onset minterms, we have to select a minimum set of columns in a prime implicant table such that there is at least one X in every row. This is the classical *minimum unate covering problem* which has been shown to be NP-complete [Garey 1979]. Nevertheless there are several reduction techniques that help simplify solving the unate covering problem:

Simplification by Essential Prime Implicants. A row with a single X represents a (relatively) essential vertex, and the corresponding column represents a (relatively) essential prime implicant. The column must be selected in the final cover because any prime cover for the function will have to contain

(a)

0	0000
5	0101
7	0111
8	1000
9	1001
10	1010
11	1011
14	1110
15	1111

(b)

0, 8	–000	→ E
5, 7	01–1	→ D
7, 15	–111	→ C
8, 9	100–	
8, 10	10–0	
9, 11	10–1	
10, 11	101–	
10, 14	1–10	
11, 15	1–11	
14, 15	111–	

(c)

8, 9, 10, 11	10––	→ B
10, 11, 14, 15	1–1–	→ A

FIGURE 6.20

Prime implicant generation.

	A	B	C	D	E
0000					x
0101				x	
0111			x	x	
1000		x			x
1001		x			
1010	x	x			
1011	x	x			
1110	x				
1111	x		x		

FIGURE 6.21

Prime implicant table.

the prime that contains the onset minterm corresponding to this row. Therefore we can simplify the prime implicant table by removing the columns corresponding to (relatively) essential prime implicants and removing the rows covered by these removed columns.

Example 6.42 In the prime implicant table of Figure 6.21 **A, B, D**, and **E** are essential prime implicants. We select the essential prime implicants since they have to be contained in any prime cover. This results in a cover for the function, since selecting columns **A, B, D**, and **E** results in the presence of X in every row.

Some functions may not have essential prime implicants. Consider the hypothetical prime implicant table of Figure 6.22a. There is no row with a single X. It is necessary to make an arbitrary selection of a prime to begin with. Assume that prime **A** is selected. We obtain the reduced table of Figure 6.22b after deleting column **A** and the first two rows contained by **A** from the table of Figure 6.22a.

Simplification by Column Dominance. A column **U** of a prime implicant table is said to dominate another column **V** if **U** contains every row contained by **V**. We can delete the dominated columns, since selecting the dominating column will result in covering more uncontained minterms than the dominated column. Note that the dominating column might not exist in a minimum solution. Further if minimizing the literal count was our objective, then we can only delete dominated columns that correspond to primes with equal or more literals than the dominating prime.

	A	B	C	D	E	F	G	H
0000	x							x
0001	x	x						
0101		x	x					
0111			x	x				
1000							x	x
1010						x	x	
1110					x	x		
1111				x	x			

(a)

	B	C	D	E	F	G	H
0101	x	x					
0111		x	x				
1000						x	x
1010					x	x	
1110				x	x		
1111			x	x			

(b)

	C	D	E	F	G
0101	x				
0111	x	x			
1000					x
1010				x	x
1110			x	x	
1111		x	x		

(c)

FIGURE 6.22

Cyclic prime implicant table.

Example 6.43 In the reduced table of Figure 6.22b column **B** is dominated by column **C** and column **H** is dominated by column **G**. Reducing the table of Figure 6.22b yields the table of Figure 6.22c. In this table **C** and **G** are relatively essential prime implicants. Choosing **C** and **G** results in the selection of **E**, which completes the cover $f = \{\mathbf{A}, \mathbf{C}, \mathbf{E}, \mathbf{G}\}$. We are not guaranteed that this cover is minimum; we have to backtrack to our arbitrary choice of selecting prime **A** and delete prime **A** from the table, *i.e.,* explore the possibility of constructing a cover that does not have **A** in it. This results in $f = \{\mathbf{B}, \mathbf{D}, \mathbf{F}, \mathbf{H}\}$.

Simplification by Row Dominance. A row *i* of a prime implicant table is said to dominate another row *j* if *i* has a 1 in every column in which *j* has a 1. Any minimum expression derived from a table which contains both rows *i* and *j* can be derived from a table which only contains the dominated row.

Example 6.44 In Figure 6.22c, row 0111 dominates row 0101 and can be deleted; row 1010 dominates row 1000 and can be deleted as well.

A Branch-and-Bound Covering Strategy. The covering procedure of the Quine-McCluskey method is summarized below. The input to the procedure is the prime implicant table *T*.

1. Delete the dominated primes (columns) and the dominating minterms (rows) in *T*. Detect essential primes[1] in *T* by checking to see if any minterm is contained by a single prime implicant. Add these essential prime implicants to the selected set. Repeat until no new essential primes are detected.
2. If the size of the selected set of prime implicants equals or exceeds the best solution thus far, return from this level of recursion. If there are no elements left to be contained, declare the selected set as the best solution recorded thus far.

[1]These primes may not be essential primes of the original function or table.

3. Heuristically select a prime implicant.
4. Add this prime implicant to the selected set and recur for the sub-table result ing from deleting the prime implicant and all minterms that are contained by this prime implicant. Then, recur for the sub-table resulting from deleting this prime implicant without adding it to the selected set.

6.3.2.2 *Other methods*

State-of-the-art exact two-level logic minimization algorithms, such as ESPRESSO [Rudell 1987] and Scherzo [Coudert 1995], are all based on the Quine-McCluskey method, but are able to outperform the Quine-McCluskey method significantly due to superior prime generation, implicant table generation, and covering techniques. In particular, with decision diagram based data structures, Scherzo [Coudert 1995] was able to outperform ESPRESSO by two orders of magnitude in terms of speed. Introductions to ESPRESSO and decision diagram based two-level logic minimization can be found in [Devadas 1994] and [Minato 1996], respectively. A good overview on two-level logic minimization can be found in [Coudert 1994].

6.3.3 Multilevel logic minimization

Two-level logic is limited because not all Boolean functions can be efficiently represented in the SOP form. Multilevel logic implementation of a function is often faster and smaller than two-level logic. Therefore multilevel realizations are the preferred means of implementing combinational logic in ***very large scale integrated*** (VLSI) systems. Because of the increased potential for reusing sub-circuits, there are more degrees of freedom in implementing a Boolean function than in the two-level case. This increased freedom, however, largely expands the search space in identifying an optimal solution.

The area of multilevel logic synthesis has blossomed since the mid-1980s. Many of the methods developed have been successfully used in commercially available computer-aided design packages. There are two types of basic approaches, rule-based local transformations and algorithmic transformations. Rule-based local transformations were developed at IBM in the late 1970s, known as the LSS system [Darringer 1981]. A rule transforms a pattern for a local set of gates and interconnections into another equivalent one when certain patterns are recognized in logic netlists. The transformations have somewhat limited optimization capability since they are local in nature and do not have a global perspective of the design.

Algorithmic transformations began to evolve in about 1981, in parallel with activity in two-level logic synthesis and influenced by it. The algorithmic counterpart uses two phases: a technology-independent step based on algorithms for manipulating general Boolean functions [Brayton 1982] and a technology mapping step (the subject of Section 6.4) where the design described in terms of generic Boolean functions is mapped into a set of gates that can be implemented in the design method of choice (gate arrays, standard cells, or macrocells). Both rule-based methods (*e.g.*, [Darringer 1984; Bartlett 1986]) and algorithmic methods (*e.g.,* [Brayton 1987; Bostick 1987]) have been successful. Algorithmic methods for logic synthesis are our main focus.

We describe the various logic transformations used in algorithmic logic synthesis systems, most of which use algebraic [Brayton 1982, 1984] and Boolean [Bostick 1987; Devadas 1989] operations in technology-independent optimization, and use graph covering methods [Keutzer 1987] in technology mapping. We first introduce technology-independent optimization and focus primarily on area minimization. Implementation details of the algorithms can be found in [Brayton 1987, 1990].

6.3.3.1 ***Logic transformations***

The goal of multilevel logic optimization is to obtain multilevel representation of a Boolean function optimal with respect to some design constraints. In order to restructure a logic function, a collection of different operations is helpful. The operations described below are commonly used and can be composed in a script file for orchestrated optimization.

Decomposition. Decomposition of a Boolean function is the process of reexpressing a single function as a composition of new functions.

Example 6.45 The process of translating the expression

$$F = a \cdot b \cdot c + a \cdot b \cdot d + \neg a \cdot \neg c \cdot \neg d + \neg b \cdot \neg c \cdot \neg d$$

to the set of expressions

$$\begin{aligned} F &= X \cdot Y + \neg X \cdot \neg Y \\ X &= a \cdot b, \text{ and} \\ Y &= c + d \end{aligned}$$

is decomposition.

Extraction. Extraction, related to decomposition, is applied to multiple functions. It is the process of identifying and creating new intermediate functions and their corresponding output variables, and reexpressing the original functions in terms of the original as well as the new variables.

Extraction creates nodes which feed multiple outputs. The operation identifies common subexpressions among different logic functions forming a network. New nodes corresponding to the common subfunctions are created and each of the logic functions in the original network is simplified with respect to these new nodes. The optimization problem of extraction is to find a set of intermediate functions such that the resulting network has minimum area, delay, or power.

Example 6.46 Extraction applied to the following three functions

$$\begin{aligned} F &= (a + b) \cdot c \cdot d + e \\ G &= (a + b) \cdot \neg e, \text{ and} \\ H &= c \cdot d \cdot e \end{aligned}$$

may yield

$$\begin{aligned} F &= X \cdot Y + e \\ G &= X \cdot \neg e \\ H &= Y \cdot e \\ X &= a + b, \text{ and} \\ Y &= c \cdot d \end{aligned}$$

Factoring. A factored form is a parenthesized representation of a tree network where each internal node is an AND or an OR gate and each leaf is a literal. Like SOP, factored forms are a way of representing Boolean functions and are perhaps a more natural way for multilevel circuits than the SOP representation.

A factored-form Boolean expression can be implemented using a complex CMOS gate. The number of transistors of the logic gate is closely related to the number of literals of the factored form as can be seen from the following example.

Example 6.47 Figure 6.23 shows a complex CMOS gate implementing the factored form $f = a + (b + c)d$. In general, excluding the possible output buffer, $2n$ transistors are needed to implement a factored form with n literals.

Consequently the literal count of a factored form can be used as a good estimate of hardware cost. The optimization problem associated with factoring is to find a factored form with a minimum number of literals.

Factoring is the process of deriving a factored form from an SOP representation of a function.

Example 6.48 The expression

$$F = a \cdot c + a \cdot d + b \cdot c + b \cdot d + e$$

can be factored into

$$F = (a + b) \cdot (c + d) + e$$

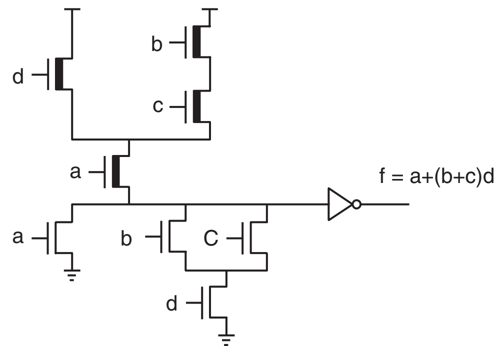


FIGURE 6.23

Factored form *vs*. complex CMOS gate implementation.

Substitution. Substitution, also called **resubstitution**, of a function *G* into *F* is the process of reexpressing *F* as a function of its original inputs and *G*.

Example 6.49 Substituting

$$G = a + b$$

into

$$F = a + b \cdot c$$

produces

$$F = G \cdot (a + c)$$

This operation creates an arc in the Boolean network connecting the node of the substituting function, namely *G*, to the node of the function being substituted into, namely *F*.

Elimination. Elimination, collapsing, or **flattening** is the the inverse operation of substitution. If *G* is a fanin node of *F*, collapsing *G* into *F* reexpresses *F* without *G*. It undoes the operation of substituting *G* into *F*.

Example 6.50 If

$$\begin{aligned} F &= G \cdot a + \neg G \cdot b \text{ and} \\ G &= c + d \end{aligned}$$

then collapsing *G* into *F* results in

$$\begin{aligned} F &= a \cdot c + a \cdot d + b \cdot \neg c \cdot \neg d \text{ and} \\ G &= c + d \end{aligned}$$

If the node *G* is not a primary output and does not fan out to other nodes, then it may be removed from the Boolean network, resulting in a network with one less node.

Flattening a logic function into the SOP form could result in an exponential growth in representation.

Example 6.51 Consider the flattening of the nodes g_1 through g_k into F with

$$\begin{aligned} F &= g_1 \cdot g_2 \cdots g_k \\ g_1 &= a_1 + b_1 \\ g_2 &= a_2 + b_2 \\ &\vdots \\ g_k &= a_k + b_k \end{aligned}$$

After flattening, the SOP representation for F will have 2^k product terms.

Given a Boolean network, we may compute the **value** of a node, which represents the saved literal count due to the existence of this node rather than collapsing this node into its fanout nodes. For nodes with little or negative values, we may eliminate them from the Boolean network by collapsing them into their fanouts. It should be noted that eliminating a node may change other nodes' values.

6.3.3.2 *Division and common divisors*

To realize the above logic transformations, it is important to define operations which, when given functions f and p, find functions q and r such that $f = p \cdot q + r$, if such q and r exist. This operation is called the **division** of f by p generating **quotient** q and **remainder** r. The function p is called a **divisor** of f if r is not null and a **factor** if r is null.

The conditions for p being a Boolean factor or a Boolean divisor are stated in the following propositions.

Proposition 6.1. *A logic function p is a Boolean factor of a logic function f if and only if $f \cdot \neg p = 0$ (that is, the onset of f is contained in the onset of p).*

Proposition 6.2. *If $f \cdot p \neq 0$, then p is a Boolean divisor of f.*

For a given division operation, the resulting q and r may depend upon the particular representation of f and p. Moreover for any logic function, there are many Boolean factors and divisors. This fact poses a problem in choosing a good factor and divisor. If the domain is restricted to a particular subset of expressions, then the division operation is unique and much easier to carry out. A restricted version of such division is called **algebraic division**.

6.3.3.3 *Algebraic division*

We begin the description of algebraic division with some definitions. The **support** of a Boolean expression f denoted as $sup(f)$ is the set of all variables v that syntactically occur in f as v or $\neg v$. For example, if $f = a + \neg a + b \cdot c$, then $sup(f) = \{a, b, c\}$. We say that f is **orthogonal** to g, written as $f \perp g$, if $sup(f) \cap sup(g) = \emptyset$. For example, $f = a + b$ and $g = c + d$ are orthogonal.

The function g is an **algebraic divisor** of f if there exist h and r such that $f = g \cdot h + r$, where $h \neq 0$, $g \perp h$, and the remainder r is minimal, *i.e.,* has *as few cubes as possible*. Under this condition on the remainder, the **quotient** h, denoted as f/g, is in fact unique. We say the function g divides f *evenly* if $f = g \cdot h$, where $h \neq 0$, $g \perp h$, and $r = 0$.

We consider two main problems of algebraic optimization, namely computing quotients f/g given f and g, and determining divisors g of a given function f.

Computing the Quotient. Given two covers (*i.e.*, *sets* of cubes) $f = \{b_1, b_2, \ldots, b_{|f|}\}$ and $g = \{a_1, a_2, \ldots, a_{|g|}\}$, we define $h_i = \{c_j \mid a_i \cdot c_j \in f\}$ for all $i = 1, 2, \ldots, |g|$, *i.e.,* h_i corresponds to all the multipliers of the cube a_i in g that produce elements of f. It is easy to see that

$$f/g = \bigcap_{i=1}^{|g|} h_i = h_1 \cap h_2 \ldots \cap h_{|g|}$$

Example 6.52 Consider two covers

$$\begin{aligned} f &= a \cdot b \cdot c + a \cdot b \cdot d + d \cdot e, \text{ and} \\ g &= a \cdot b + e. \end{aligned}$$

We have $|g| = 2$ and $|f| = 3$. With $3 \times 2 = 6$ comparisons, we obtain

$$\begin{aligned} h_1 &= \{c, d\}, \text{ and} \\ h_2 &= \{d\} \end{aligned}$$

Hence $h_1 \cap h_2 = d$, and

$$f = (a \cdot b + e) \cdot d + a \cdot b \cdot c$$

The above algorithm requires $O(|f| \cdot |g|)$ operations. Encoding and sorting the cubes of f and g can reduce the complexity to $O((|f| + |g|) \log(|f| + |g|))$ [McGeer 1987].

Kernels and Algebraic Divisors. Given an efficient method for algebraic division, optimization can be carried out if good algebraic divisors can be found. The set of algebraic divisors is defined as $D(f) = \{g \mid f/g \neq 0\}$. The **primary divisors** of f are defined as $P(f) = \{f/c \mid c \text{ is a cube}\}$.

Example 6.53 If

$$f = a \cdot b \cdot c + a \cdot b \cdot d \cdot e$$

then

$$f/a = b \cdot c + b \cdot d \cdot e$$

is a primary divisor.

Proposition 6.3. *Every divisor of f is contained in a primary divisor, i.e., if g divides f, then* $g \subseteq p \in P(f)$.

Proof. Let $c \in f/g$ be a cube. Then $g \subseteq f/(f/g)$ and $f/(f/g) \subseteq f/c \in P(f)$. □

A function g is termed **cube-free** if the only cube that divides g evenly is 1. The **kernels** of f are defined as $K(f) = \{k \mid k \in P(f), k \text{ is cube-free}\}$. For a kernel $k \in K(f)$, its **cokernel** is the cube c with $f/c = k$.

Example 6.54 If

$$f = a \cdot b \cdot c + a \cdot b \cdot d \cdot e$$

then

$$f/a = b \cdot c + b \cdot d \cdot e$$

is a primary divisor but not cube-free since b is a factor of $f/a = b \cdot (c + d \cdot e)$. However, $f/(a \cdot b) = c + d \cdot e$ is a kernel, and $a \cdot b$ is a cokernel.

The following theorem (originally proven in [Brayton 1982]) is the basis of algebraic optimization methods.

Theorem 6.2. *Two expressions f and g have a non-cube common divisor d if and only if there exist kernels $k_f \in K(f)$ and $k_g \in K(g)$ such that $k_f \cap k_g$ has two or more terms (i.e., $k_f \cap k_g$ is not a cube).*

Proof. For the "if" part, $k_f \cap k_g$ is clearly a common divisor of f and g. It remains to prove the "only if" part. □

Assume d divides both f and g, and d has two or more terms. Then there is a cube-free SOP expression e such that e divides d. Also e divides f and g as well. By Proposition 6.3, $e \subseteq k_f \in P(f)$ and $e \subseteq k_g \in P(g)$ for some k_f and k_g. Since e is cube-free, k_f and k_g are cube-free as well. Hence, $k_f \in K(f)$ and $k_g \in K(g)$. Finally, since $e \subseteq k_f \cap k_g$, $k_f \cap k_g$ must have two or more terms. □

We can therefore use the kernels of f and g to locate common divisors. Note that these are not the only common divisors of f and g, but they are good common divisors to consider during logic optimization. We compute the set of kernels for each logic expression, then form intersections among kernels from the different logic expressions. If this intersection set contains no non-cube elements, then by Theorem 6.2, we need only look for divisors consisting of single cubes. Otherwise, we have found an algebraic divisor common to two or more expressions.

Computing the Kernels. The kernels of a function f can be computed using the algorithm of Figure 6.24. The kernel generation algorithm first makes f cube-free by finding its largest cube factor. It then selects the literals of f in a lexicographical order and divides them into f; the resulting quotient is a kernel if it is cube-free. (Note that this kernel might contain other kernels, too.) If it is not cube-free, then it is made cube-free by selecting its largest cube factor. Note that in this context the largest cube is the cube with the most number of

```
KERNELS($f$){
    $c_f$ = largest cube (with maximum number of literals) factor of $f$ ;
    $K$ = KERNEL1(0, $f/c_f$) ;
    if ($f$ is cube-free)
        return($f \cup K$) ;
    return($K$) ;
}

KERNEL1($j$, $g$){
    $R = g$ ;
    $N$ = Maximum index of variables in $g$ ;
    for($i = j + 1$; $i$ ″ $N$; $i = i + 1$) {
        if ($l_i$ in 1 or no cubes of $g$) continue ;
        $c$ = largest cube dividing $g/l_i$ evenly ;
        if (for all $k$ ″ $i$, $l_k \notin c$) /* Pruning Condition */
            $R = R \cup$ KERNEL1 ($i$, $g/(l_i \cap c)$) ;
    }
    return($R$) ;
}
```

FIGURE 6.24

Procedure to determine all the kernels of a single-output logic function.

literals. The procedure is repeated on the resulting functions until functions with no kernels (called the level-0 kernels of f) are found. A major efficiency is obtained by noting that if the largest cube factor extracted contains an already selected literal, then the current branch can be terminated, since all the kernels that can be found by continuing have already been generated. This leads to an algorithm in which no cokernel is duplicated.

Example 6.55 Consider

$$f = a \cdot b \cdot c \cdot d + a \cdot b \cdot c \cdot e + a \cdot b \cdot e \cdot f$$

In the routine **KERNELS** $c_f = a \cdot b$. Therefore,

$$f/c_f = c \cdot d + c \cdot e + e \cdot f$$

In the next step we call **KERNEL1**(0, $c \cdot d + c \cdot e + e \cdot f$).

In **KERNEL1** we set $R = \{c \cdot d + c \cdot e + e \cdot f\}$. Since the ordering is lexicographic, we have $l_1 = a, l_2 = b$, etc. Note that $N = 6$. The literals l_1 and l_2 are in none of the terms of R, and we move to $l_3 = c$. The largest cube dividing $(c \cdot d + c \cdot e + e \cdot f)/c$, which is $d + e$, is 1.

We therefore make a recursive call to **KERNEL1**(3, $(c \cdot d + c \cdot e + e \cdot f) = (c \cap 1)$). This call returns with $\{d + e\}$. In the parent **KERNEL1** R is set to $\{c \cdot d + c \cdot e + e \cdot f, d + e\}$. We skip $l_4 = d$ and move to $l_5 = e$. The largest cube evenly dividing $(c \cdot d + c \cdot e + e \cdot f)/e$, which is $c + f$, is 1. We next call **KERNEL1**(5, $(c \cdot d + c \cdot e + e \cdot f)/(e \cap 1)$). This returns with $c + f$.

We end with $K = R = \{c \cdot d + c \cdot e + e \cdot f,\ d + e,\ c + f\}$.

If the largest cube factor extracted contains an already selected literal, then the current branch can be terminated, since all kernels that can be found by continuing have already been generated. We illustrate the pruning condition with the following example.

Example 6.56 Consider

$$f = a \cdot b \cdot c \cdot (d + e) \cdot (k + l) + a \cdot f \cdot g + h$$

In the first call to **KERNEL1**, we will generate the kernels corresponding to

$$f/a = b \cdot c \cdot (d + e) \cdot (k + l) + f \cdot g$$

KERNEL1 calls itself recursively to compute

$$f/(a \cdot b) = c \cdot (d + e) \cdot (k + l)$$

Since $f/(a \cdot b)$ is not cube-free, the next recursive call to **KERNEL1** will use $(d + e) \cdot (k + l)$. All the kernels of this expression will be generated.

We move up one level in the recursion and compute

$$f/(a \cdot c) = b \cdot (d + e) \cdot (k + l)$$

At this stage, we note that $f/(a \cdot c)$ is not cube-free, and the largest cube dividing this expression evenly is b. However, b is an already selected literal implying that we have already generated the kernels for the cube-free expression $(d+e)\cdot(k+l)$. We do not have to recursively call **KERNEL1** for this branch and can go ahead to $f/(a \cdot d)$.

It is possible to modify the **KERNEL1** procedure to generate only the level-0 kernels which do not contain other kernels. This modification is based on the observation that if no kernels of g are found in the **for** loop, then g is a level-0 kernel.

Factoring Algorithm. A function can be algebraically factored using the generic factoring algorithm shown in Figure 6.25.

The procedure **DIVIDE** performs algebraic division and reexpresses f as $g \cdot h + r$. The procedure **CHOOSE_DIVISOR** is critical to obtaining a good factorization. One alternative is to select an arbitrary level-0 kernel as a divisor. This may not produce the best final result. Another alternative is to select a kernel which when substituted into the original function maximally reduces the total number of literals.

Example 6.57 Given

$$X = a\cdot c + a\cdot d + a\cdot e + a\cdot g + b\cdot c + b\cdot d + b\cdot e + b\cdot f + c\cdot e + c\cdot f + d\cdot f + d\cdot g$$

if, in the procedure **CHOOSE_DIVISOR**, we choose literals in lexicographical order, we obtain

$$X = a\cdot(c+d+e+g) + b\cdot(c+d+e+f) + c\cdot(e+f) + d\cdot(f+g)$$

However, if we choose kernels, we obtain a better factorization

$$X = (c+d+e)\cdot(a+b) + f\cdot(b+c+d) + g\cdot(a+d) + c\cdot e$$

which has fewer literals.

Extraction and Resubstitution Algorithm. To identify cube-free expressions that occur in multiple functions $\{f_i\}$, we do the following.

1. Generate kernels for each f_i.
2. Select a pair of kernels $k_1 \in K(f_i)$ and $k_2 \in K(f_j)$ for $i \neq j$ such that $k_1 \cap k_2$ is not a cube. If no such pair exists, stop.

```
GFACTOR(f){
    if (number of terms in f is 1 )
        return(f) ;
    g = CHOOSE_DIVISOR(f) ;
    (h, r) = DIVIDE(f, g) ;
    f = GFACTOR(g) · GFACTOR(h) + GFACTOR(r) ;
    return(f) ;
}
```

FIGURE 6.25

Procedure to algebraically factor a function.

3. **Set a new variable v equal $k_1 \cap k_2$.**
4. **Update the associated functions to**

$$f_i = v \cdot (f_i/(k_1 \cap k_2)) + r_i$$

where r_i is the remainder of the division $f_i/(k_1 \cap k_2)$.

Common cubes are extracted as follows.

1. **Select a pair of cubes $c_1 \in f_i$, $c_2 \in f_j$ for $i \neq j$ such that $c_1 \cap c_2$ consists of two or more literals. If no such pair exists, stop.**
2. **Set a new variable u equal $c_1 \cap c_2$.**
3. **Update each function f_i with the new variable u wherever possible in the network.**

Example 6.58 Consider the factored functions

$$\begin{aligned} X &= a \cdot b \cdot (c \cdot (d+e) + f + g) + h, \quad \text{and} \\ Y &= a \cdot i \cdot (c \cdot (d+e) + f + j) + k \end{aligned}$$

We have $d + e$ being a level-0 kernel of both functions. Extraction results in

$$\begin{aligned} L &= d+e, \\ X &= a \cdot b \cdot (c \cdot L + f + g) + h, \quad \text{and} \\ Y &= a \cdot i \cdot (c \cdot L + f + j) + k \end{aligned}$$

Now, we select $c \cdot L + f + g$ as a level-0 kernel of the reexpressed X and $c \cdot L + f + j$ as a level-0 kernel of reexpressed Y. We obtain

$$\begin{aligned} M &= c \cdot L + f \\ L &= d+e \\ X &= a \cdot b \cdot (M + g) + h, \quad \text{and} \\ Y &= a \cdot i \cdot (M + j) + k \end{aligned}$$

Now X and Y have no kernel intersections that are not cubes. We now extract common cubes. The cubes $a \cdot b \cdot M$ in X and $a \cdot i \cdot M$ in Y have two literals in common. Extraction produces

$$\begin{aligned} N &= a \cdot M \\ M &= c \cdot L + f \\ L &= d+e \\ X &= b \cdot (N + a \cdot g) + h, \quad \text{and} \\ Y &= i \cdot (N + a \cdot j) + k \end{aligned}$$

Because we are continually recomputing level-0 kernels on the reexpressed functions, it is possible to obtain decompositions corresponding to level-k kernels for $k > 0$. If we collapse L into M into N above, we obtain

$$\begin{aligned} N &= a \cdot (c \cdot (d+e) + f) \\ X &= b \cdot (N + a \cdot g) + h, \quad \text{and} \\ Y &= i \cdot (N + a \cdot j) + k \end{aligned}$$

where N contains a level-1 kernel of the original X and Y, since it contains the level-1 kernel M which contains the level-0 kernel $d + e$.

Algebraic Resubstitution with Complement. Algebraic factorization and resubstitution can be performed with the complement of a given divisor.

Example 6.59 Consider

$$f = a \cdot b + a \cdot c + \neg b \cdot \neg c \cdot d$$

where we choose $b + c$ as a level-0 kernel of f and decompose f as

$$\begin{aligned} f &= a \cdot X + \neg b \cdot \neg c \cdot d, \quad \text{and} \\ X &= b + c \end{aligned}$$

In many cases it is useful to check if the complement of the new variable is an algebraic divisor for the function. In this case we can obtain

$$\begin{aligned} f &= a \cdot X + \neg X \cdot d, \quad \text{and} \\ X &= b + c. \end{aligned}$$

6.3.3.4 *Common divisors*

One of the key problems in algebraic optimization is the identification of good (common) divisors. We have described the use of kernels for determining a good set of divisors for algebraic factoring, decomposition, and extraction. The problem of finding a kernel and finding a single-cube or multiple-cube divisor can be reduced to the combinatorial optimization problem of rectangle covering [Rudell 1989]. This formulation of the problem is not only elegant, but it also favors the development of fast and effective algorithms.

Before introducing the method, we give some definitions.

A **(combinatorial) rectangle** (R, C) of a matrix B, with entries $B_{ij} \in \{0,1,*\}$, is a subset of rows R and subset of columns C such that $B_{ij} \in \{1, *\}$ for all $i \in R$ and $j \in C$. Note that the rows and columns forming the rectangle do not have to be contiguous.

A rectangle (R_1, C_1) is said to *strictly contain* rectangle (R_2, C_2) if $R_2 \subseteq R_1$ and $C_2 \subset C_1$, or $R_2 \subset R_1$ and $C_2 \subseteq C_1$.

A rectangle (R, C) of B is said to be a **prime rectangle** if it is not strictly contained in any other rectangle of B.

The **corectangle** of a rectangle (R, C) is the pair (R, C') where C' is the set of columns not in C.

A set of rectangles $\{(R^k, C^k)\}$ forms a **rectangle cover** of matrix B if $B_{ij} = 1$ implies that $i \in R^k$ and $j \in C^k$ for some k. Thus, each 1-entry in B must be covered by at least one rectangle from the cover. A covering need not be disjoint, and therefore a 1-entry in B can be covered by more than one rectangle. The *-entries of B are not required to be covered by any rectangle in the cover and therefore represent don't-care points in the matrix.

Example 6.60 In the following matrix

	1	2	3	4	5
1	1	1	1	0	0
2	1	*	1	0	*
3	0	1	1	0	1
4	1	0	1	1	1

The tuple ({1,2}, {2,3}) is a rectangle, but it is not prime as it is contained by the prime rectangle ({1,2}, {1,2,3}). The tuple ({2,4}, {1,3,5}) is another prime rectangle while ({2,3}, {1,2}) is not a rectangle.

Each rectangle (R^k, C^k) has an associated weight or cost defined by a weight function $w(R^k, C^k)$. The weight of a rectangle cover is then defined as the sum

$$\sum_k w(R^k, C^k)$$

The ***minimum-weighted rectangle covering problem*** is that of finding a rectangle cover of a matrix with minimum total weight.

Rectangles and Kernels. Rectangles provide an alternate way of looking at the kernels of a function. By representing a Boolean expression as a **cube-literal matrix**, where each row corresponds to a cube in the expression and the columns correspond to all the distinct literals, each prime rectangle is a cokernel while each corectangle of a prime rectangle is a kernel of the expression.

Example 6.61 Consider the expression $g = a \cdot b \cdot e + a \cdot c \cdot d + b \cdot c \cdot d$. It can be represented using a cube-literal matrix shown below.

	a	b	c	d	e
a · b · e	1	1	0	0	1
a · c · d	1	0	1	1	0
b · c · d	0	1	1	1	0

Consider the prime rectangle $(R, C) = (\{2,3\}, \{3,4\})$ and its corectangle $(R, C') = (\{2,3\}, \{1,2,5\})$. The rectangle obviously corresponds to a cube $c \cdot d$ that is common to all the product terms corresponding to rows in R. Since the rectangle is prime, it is the largest cube common to all the product terms in R. If this cube is extracted from these product terms, the resulting expression is cube-free and is also a divisor of the original function g. In other words, the resulting expression is a kernel of g. The expression resulting from the extraction of the cube corresponds to the corectangle $(R, C') = (\{2,3\}, \{1,2,5\})$, which is $a + b$.

From the rectangle interpretation of kernels, it is also possible to understand more clearly the notion of the level of a kernel. A level-0 kernel is the corectangle of a prime rectangle which has no other rectangle containing its column set, *i.e.,* a rectangle of maximal width. The corectangle of a prime rectangle of maximal height, *i.e.,* one whose row set is not contained in any other rectangle, corresponds to a kernel of maximal level.

Common-Cube Extraction. Common-cube extraction is the process of finding cubes common to two or more expressions and extracting the common cube to simplify each of the expressions. To optimize the network it is necessary to find the particular cubes to introduce that provide an optimal decomposition. The optimal decomposition can be defined as minimizing the total number of literals summed over all expressions or minimizing the total number of literals given a bound on the number of levels of logic in the final circuit.

Common cubes can be easily identified using the cube-literal matrix described above.

Example 6.62 Consider the equations

$$\begin{aligned} F &= a\cdot b\cdot c + a\cdot b\cdot d + e\cdot g \\ G &= a\cdot b\cdot f\cdot g, \text{ and} \\ H &= b\cdot d + e\cdot f \end{aligned}$$

The cube-literal matrix for these expressions is

	a b c d e f g
$F_1 : a \cdot b \cdot c$	1 1 1 0 0 0 0
$F_2 : a \cdot b \cdot d$	1 1 0 1 0 0 0
$F_3 : e \cdot g$	0 0 0 0 1 0 1
$G_1 : a \cdot b \cdot f \cdot g$	1 1 0 0 0 1 1
$H_1 : b \cdot d$	0 1 0 1 0 0 0
$H_2 : e \cdot f$	0 0 0 0 1 1 0

The rectangle ({1,2,4}, {1,2}) corresponds to the common cube $a{\cdot}b$ which is present in functions F and G. If this common cube is extracted as a new function X, the equations can be rewritten as

$$\begin{aligned} F &= X\cdot c + X\cdot d + e\cdot g \\ G &= X\cdot f\cdot g \\ H &= b\cdot d + e\cdot f, \text{ and} \\ X &= a\cdot b \end{aligned}$$

The process of extracting a cube modifies a Boolean network. A new node is added to the Boolean network with a logic function which is the common-cube divisor. All functions which the cube divides are replaced with the algebraic division of the function by the single cube. In order to extract cubes efficiently in an iterative algorithm, it is necessary to modify the cube-literal matrix incrementally to reflect the extraction of the cube. The advantage is that the cube-literal matrix does not have to be recreated as each cube is extracted.

The modifications required to form the new cube-literal matrix are the following. A new row is added to reflect the new single cube expression added to the network. The entries covered by the rectangle are marked with a * to reflect that the position has been covered. However, the * allows other rectangles to cover the same position.

The choice of the weight function for a rectangle measures the optimization goal for cube extraction. To minimize the total number of literals in the network, the weight of a rectangle is chosen so that the weight of a rectangle cover of the cube-literal matrix equals the total number of literals in the network after the new single-cube functions are added to the network. Hence, a minimum weighted rectangle cover corresponds to the optimal simultaneous extraction of a collection of cubes. The weight of a rectangle is defined as:

$$w(R,C) = \begin{cases} |C| & \text{if } |R| = 1 \\ |C| + |R| & \text{if } |R| > 1 \end{cases}$$

If there is a single row in the rectangle, then it corresponds to leaving the cube unchanged in the network. Hence, the weight of the rectangle counts the number of literals in the cube, which equals the number of columns. When the number of rows is greater than one, this corresponds to creating a new single cube function with $|C|$ literals and substituting this new function into $|R|$ other cubes at a cost of $|R|$ literals.

Note that the above weight does not reflect the savings obtained in terms of the number of literals by extracting a common cube. Therefore, when searching for a cube to extract it is useful to define a second function called the **value** of the rectangle. For cube extraction, the value of the rectangle should indicate the savings obtained from extracting the corresponding cube. Since the number of literals before cube extraction is the number of 1-entries in the rectangle and the number of literals after cube extraction is the weight of the rectangle, the value $v(R, C)$ of a rectangle is defined as

$$v(R,C) = \left|\left\{(i,j)\middle|B_{ij} = 1, i \in R, j \in C\right\}\right| - w(R,C)$$

Example 6.63 For the rectangle ({1,2,4}, {1,2}) in the cube-literal matrix of the previous example, the weight is the number of rows plus the number of columns, which equals 5. There are 6 positions in this rectangle and each of them has a 1. Therefore, the value of the rectangle is $6 - 5 = 1$. Therefore only one literal can be saved by extracting this rectangle, as illustrated in the previous example.

Kernel Intersection. As described previously, intersections among the kernels of a collection of expressions are useful for finding common multiple-cube divisors between two or more expressions. If two functions share a common multiple-cube divisor, then the common divisor can be found as the intersection of a kernel from each of the functions.

The Boolean matrix associated with the optimal kernel intersection problem is called the **cokernel-cube matrix**. A row in this matrix corresponds to a cokernel (and its associated kernel) and each column corresponds to a cube present in some kernel, called a **kernel-cube**. The entry B_{ij} is set to 1 if the kernel

associated with row *i* contains the cube associated with column *j*. Then a rectangle of the cokernel-cube matrix identifies an intersection of kernels. The columns of the rectangle identify the cubes in the subexpression, and the rows in the rectangle identify the particular functions the subexpression divides.

Example 6.64 Consider the functions

$$\begin{aligned} F &= a \cdot f + b \cdot f + a \cdot g + c \cdot g + a \cdot d \cdot e + b \cdot d \cdot e + c \cdot d \cdot e \\ G &= a \cdot f + b \cdot f + a \cdot c \cdot e + b \cdot c \cdot e, \text{ and} \\ H &= a \cdot d \cdot e + c \cdot d \cdot e \end{aligned}$$

The kernels and cokernels of each of the functions are shown below.

Function	Cokernel	Kernel
F	a	d · e + f + g
F	b	d · e + f
F	d · e	a + b + c
F	f	a + b
F	c	d · e + g
F	g	a + c
G	a	c · e + f
G	b	c · e + f
G	f	a + b
G	c · e	a + b
H	d · e	a + c

Note that functions *F* and *G* are themselves kernels but have not been shown above for ease of presentation. Let us number the cubes in the original function from 1 to 13, with $a \cdot f$ being 1, $b \cdot f$ being 2, and so on. The cokernel-cube matrix for this set of kernels is shown below. Note that instead of 1's in the matrix, we have numbers. These numbers indicate a cube of the original functions formed by multiplying the cokernel corresponding to a row and the cube corresponding to a column. For example, in the third row under column *a* we have the number 5 corresponding to the the fifth cube $a \cdot d \cdot e$.

	a	b	c	ce	de	f	g
F : a	0	0	0	0	5	1	3
F : b	0	0	0	0	6	2	0
F : d · e	5	6	7	0	0	0	0
F : f	1	2	0	0	0	0	0
F : c	0	0	0	0	7	0	4
F : g	3	0	4	0	0	0	0
G : a	0	0	0	10	0	8	0
G : b	0	0	0	11	0	9	0
G : f	8	9	0	0	0	0	0
G : c · e	10	11	0	0	0	0	0
H : d · e	12	0	13	0	0	0	0

Rectangle ({3,4,9,10}, {1,2}) identifies the subexpression $a + b$. This corresponds to the factorization of the equations into the form

$$\begin{aligned} F &= d \cdot e \cdot X + f \cdot X + a \cdot g + c \cdot g + c \cdot d \cdot e \\ G &= c \cdot e \cdot X + f \cdot X \\ H &= a \cdot d \cdot e + c \cdot d \cdot e \text{ and} \\ X &= a + b \end{aligned}$$

Whenever a new subexpression is identified, it is inserted into the Boolean network. This insertion consists of adding a new node to the network and dividing the node into each of the expressions which this node divides. A new cokernel-cube matrix is then created for the modified Boolean network.

To reduce the complexity of extracting each factor from the network it is desirable to modify the cokernel-cube matrix incrementally as each subexpression is identified. To do this, new rows are added to the cokernel-cube matrix for each kernel of the new subexpression. The cubes which are formed by the insertion of this new factor into the network are then marked as covered. This includes the points directly contained in the rectangle and other points which are labeled with the same number. These points are marked * so that other rectangles can cover them.

The weight of a rectangle of the cokernel-cube matrix is chosen to reflect the number of literals in the network if the corresponding common subexpression is inserted into the network. A minimum weighted rectangle cover of the cokernel-cube matrix then corresponds to a simultaneous selection of a set of subexpressions to add to the network in order to minimize the total number of literals.

Let w_j^c be the number of literals in the kernel-cube for column j. w_j^c is also called the column weight of column j. If a rectangle (R, C) is used to identify a subexpression, then a new function is formed from the columns of C. This new function has $\sum_{j \in C} w_j^c$ literals. Let w_i^r be 1 plus the number of literals the cokernel corresponding to row i. w_i^r is also called the row weight of row r. The chosen subexpression divides the expressions indicated by the rows R of the rectangle. After algebraic division by the subexpression, each of these expressions consists of a sum of the corresponding cokernel cubes multiplying the literal for the new expression. The number of literals in the affected functions after the extraction of the subexpression corresponding to the rectangle is $\sum_{i \in R} w_i^r$. Therefore, the weight of a rectangle (R, C) in the cokernel-cube matrix is defined as:

$$w(R, C) = \sum_{i \in R} w_i^r + \sum_{j \in C} w_j^c$$

The value of a rectangle measures the difference in the number of literals in the network if the particular rectangle is selected. The number of literals after the rectangle is selected is the weight of the rectangle as defined above. Let V_{ij} be the number of literals in the cube which is covered by position (i, j) of the cokernel-cube matrix. Then the number of literals before extraction of the rectangle is simply $\sum_{i\in R,\, j\in C} V_{ij}$. As elements of the cokernel-cube matrix are covered, their values V_{ij} are set to 0. This includes the elements V_{ij} covered by the matrix and all other elements which represent the same cube in the network. The value of a rectangle (R, C) of the cokernel-cube matrix is thus defined as

$$v(R,C) = \sum_{i\in R, j\in C} V_{ij} - w(R,C)$$

Example 6.65 For the rectangle ({3,4,9,10}, {1,2}) of the cokernel-cube matrix in the previous example, $\sum_{i\in R,\, j\in C} V_{ij} = 3+3+2+2+2+2+3+3 = 20$, $\sum_{i\in R} w_i^r = 3+2+2+3 = 10$, $\sum_{j\in C} w_j^c = 1+1 = 2$. Therefore, the value of the rectangle is 20 – 10 – 2 = 8. Eight literals can be saved by extracting the expression corresponding to the rectangle, as can be verified in the example above.

Rectangle Covering. Since minimum-weighted rectangle covering corresponds to optimum algebraic extraction, it offers a unified approach to the extraction, factorization, and decomposition of Boolean expressions. However, the minimum-weighted rectangle covering problem is NP-complete [Rudell 1989] and thus heuristic algorithms are resorted.

There are two types of algorithms for rectangle covering. The first type of algorithm is greedy and selects one rectangle at a time and modifies the matrix to reflect the extraction of the rectangle. The advantage of this technique is that it immediately takes into account common factors between the newly extracted function and the rest of the logic network. The disadvantage of this approach is that it selects only one rectangle at a time and does not easily account for the simultaneous extraction of multiple rectangles. The second type of algorithm finds the best collection of factors to extract at each step by solving the minimum-weighted rectangle covering problem heuristically. First, all the prime rectangles are generated, and a collection of rectangles are then extracted. Second, the matrix is updated, and the entire process is repeated to find factors between the new expressions and the remainder of the logic network. A detailed exposition of this approach can be found in [Rudell 1989].

6.3.3.5 *Boolean division*

So far we have primarily described algebraic optimization methods. Apparently the optimality of algebraic division is limited. For example, the Boolean expression $f = a\neg b + ad + \neg ab + bd + \neg ac + \neg bc + cd$ can not be factored into $f = (a + b + c)(\neg a + \neg b + d)$ through algebraic division. It motivates the development of Boolean division.

To do so, in **Boolean resubstitution** we would like to reexpress a given Boolean function $f(\boldsymbol{x})$ in terms of a given divisor $g(\boldsymbol{x})$. The computation can be done by first building the function

$$h(\boldsymbol{x},y) = f(\boldsymbol{x}) \wedge (y \equiv g(\boldsymbol{x}))$$

where y is a newly introduced Boolean variable representing the output signal of function g. We then minimize function h with respect to the don't care set $y \not\equiv g(\boldsymbol{x})$ while insisting y to be a support variable of h. If h after minimization is "simpler" than function f, then the resubstitution is successful.

Boolean resubstitution can be formalized more generally as **functional dependency** [Jiang 2004]. We say that a function $f(\boldsymbol{x})$ **functionally depends** on a set of functions $g_1(\boldsymbol{x}), \ldots, g_m(\boldsymbol{x})$ if there exists some function h such that

$$f(x) = h(g_1(x), \ldots, g_m(x))$$

The necessary and sufficient condition, informally speaking, is that the set $\{g_1, \ldots, g_m\}$ of functions must be more distinguishing than f on the domain elements. That is, for every $a, b \in \mathbb{B}^n$ with $f(a) \neq f(b)$ there must exist some g_i such that $g_i(a) \neq g_i(b)$. ROBDD and SAT based computation of functional dependency can be found in [Jiang 2004] and [Lee 2007; Mishchenko 2007a], respectively.

To see that Boolean resubstitution is a special case of functional dependency, for $\boldsymbol{x} = (x_1, \ldots, x_n)$ we set $g_i(\boldsymbol{x}) = x_i$ for $i = 1, \ldots, n$ and $g_{n+1}(\boldsymbol{x}) = g(\boldsymbol{x})$. Thus functional dependency reduces to Boolean resubstitution $f(\boldsymbol{x}) = h(\boldsymbol{x}, g(\boldsymbol{x}))$. In fact, we can minimize the support variables of h by setting as many $g_i(\boldsymbol{x}) = 0$ (or 1) as possible to remove x_i from the support set of h.

6.3.4 **Combinational complete flexibility**

The aforementioned multilevel logic minimization approaches, such as decomposition, extraction, factoring, substitution, and elimination, change the structure of a Boolean network. In contrast, in this section we study how to perform logic minimization without changing a multilevel network structure. More specifically, given a structurally optimized multilevel network, we may further minimize it by simplifying the logic expression within every node.

To minimize the logic function of a node u in a Boolean network, we would like to characterize the don't care conditions of the node u, such that we may choose the best among the set of valid functions, called *permissible functions*, that can implement u without changing the functionality of the entire Boolean network. Notice that node u imposes a topological constraint on a permissible function whose inputs are restricted to the fanins of node u in the Boolean network.

In fact, don't cares exist pervasively in a multilevel logic netlist because the Boolean space is largely expanded due to the existence of many intermediate variables. Let X be the set of primary input variables and Y the set of all other variables of a Boolean network. In the $\mathbb{B}^{|X|+|Y|}$ Boolean space, only $2^{|X|}$ valuations are consistent because the valid valuations are determined by the assignments on the primary input variables. Consequently a lot of invalid valuations may not appear in the Boolean network and can be exploited for logic minimization. Moreover the effect of one signal may be conditionally blocked by other signals and cannot affect the valuations of primary outputs. Based on these reasons, flexibility may exist to some extent in a multilevel logic network.

The don't-care conditions arising in multilevel logic can either be specified by the user or can be an artifact of the network structure. Essentially there are three types of don't cares: satisfiability don't cares (SDC), observability don't cares (ODC), and external don't cares (XDC). *Internal* don't-cares arise in multilevel logic because of the structure of a Boolean network. They are divided into satisfiability and observability don't-cares. User specified don't-cares or don't-cares derived from considerations other than the network structure are called *external* don't-cares.

In the following discussion, for a Boolean network, let X be the set of primary input variables, Y the set of all other variables, and $Z \subseteq Y$ the set of primary output variables. For a node i in a Boolean network, its output variable is denoted as y_i and its local or intermediate input variables, other than primary input variables, are denoted as Y_i; its local function is denoted as $f_i(X, Y_i)$ and its global function, in terms of only primary input variables, is denoted as $g_i(X)$. Of course, since we consider only acyclic Boolean networks, f_i depends only on a subset of the Y variables that are not in the transitive fanout cone TFO_i of node i.

External Don't-Cares. External don't-cares are specified for every primary output, which indicate under what valuations on the primary input variables X the value of the output is immaterial.

Satisfiability Don't-Cares. Satisfiability don't-cares are a result of the existence of the additional intermediate variables introduced at the intermediate nodes of a Boolean network. A node with output variable y_i and immediate function $f_i(X, Y_i)$ of a Boolean network imposes the relation

$$y_i \equiv f_i(X, Y_i) \tag{6.14}$$

which characterizes the set of valuations on variables X and Y that are consistent under the constraint imposed by node i. Therefore the set of satisfiability don't cares of the entire Boolean network is given by

$$SDC(X, Y) = \bigvee_i (y_i \not\equiv f_i(X, Y_i)) \tag{6.15}$$

which gives all the valuations on variables X and Y that will never occur due to the network structure and is so called because each of the relations $y_i \equiv f_i(X, Y_i)$ must be satisfied during the correct operation of the network. In order to optimize a given node i we are typically interested in the satisfiability don't cares imposed by the transitive fanin cone TFI_i of node i.

Example 6.66 Consider the network

$$\begin{aligned} y_1 &= x_1 \wedge x_2 \\ y_2 &= x_2 \vee x_3, \text{ and} \\ y_3 &= y_1 \oplus y_2 = \neg y_1 y_2 \vee y_1 \neg y_2 \end{aligned}$$

It implements function $g_3 = (x_1 \wedge x_2) \oplus (x_2 \vee x_3)$. We have the option of eliminating y_1 and y_2 or expanding the Boolean space to include these variables. If we do the latter there are assignments of variables which will never occur. For example, the assignment $y_1 = 1$ and $y_2 = 0$ will never happen. The assignments that will never occur are expressed by

$$SDC = (y_1 \not\equiv (x_1 \wedge x_2)) \vee (y_2 \not\equiv (x_2 \vee x_3)) \vee (y_3 \not\equiv (y_1 \oplus y_2))$$

To optimize f_3, the satisfiability don't care set

$$SDC_3 = (y_1 \not\equiv (x_1 \wedge x_2)) \vee (y_2 \not\equiv (x_2 \vee x_3))$$

imposed by the fanin nodes 1 and 2 of node 3 is of particular interest. Furthermore, SDC_3 in terms of the local input variables of node 3 can be computed by

$$\forall x_1, x_2, x_3 . (y_1 \not\equiv (x_1 \wedge x_2)) \vee (y_2 \not\equiv (x_2 \vee x_3)) = y_1 \neg y_2$$

which ensures that the computed SDC in term of variables y_1 and y_2 is valid under any valuation on the X variables. Accordingly, we may optimize f_3 using the impossible condition $y_1 \neg y_2$. So $f_3 = \neg y_1 y_2$ is another permissible function for node 3.

Observability Don't-Cares. Observability don't-cares occur in a network because at each node there is a network structure that limits the observability of the value of the node as seen at primary outputs.

To compute the observability don't cares ODC_i of a node i in a Boolean network N. We construct a new Boolean network N' from N by treating y_i as a (pseudo) primary input and removing node i and other induced nodes without fanouts from N. The condition that node i is observable at primary output j is given by

$$\frac{\partial g_j'}{\partial y_i} = [g_j(X, y_i = 0) \not\equiv g_j'(X, y_i = 1)] \tag{6.16}$$

where g_j' is the global function of j in network N'. That is, Formula (6.16) gives the input conditions under which the g_j' produces different values under different y_i values, *i.e.*, the conditions under which output j is sensitive to y_i. Therefore the conditions under which the value of y_i cannot be observed at any output are characterized by

$$\begin{aligned} ODC_i(X) &= \bigwedge_{y_j \in Z} (g_j'(X, y_i = 0) \equiv g_j'(X, y_i = 1)) \\ &= \bigwedge_{y_j \in Z} \neg \left(\frac{\partial g_j'}{\partial y_i} \right) \end{aligned}$$

The above computation assumes the external don't-care set is empty. For non-empty XDC, the observability of a node at some primary output should be conditioned on the external don't care set of the primary output.

Local Don't-Cares and Node Minimization. Note that SDC is in terms of X and Y variables; XDC and ODC are in terms of X variables. To minimize a node i, they are not directly useful unless they are expressed in terms of the local input variables of node i. Don't cares in terms of the local input variables are called *local don't cares*. Let

$$DC_i(X) = \bigwedge_{y_k \in Z} XDC_k(X) \vee ODC_i(X)$$

Let D_i be the local don't cares of node i. Then it can be computed by

$$D_i(Y_i) = \neg\left(\exists X. \bigwedge_{y_j \in Y_i} \left(y_j \equiv g_j(X)\right) \wedge \neg DC_i(X)\right) \tag{6.17}$$

It should be noted that we cannot simply project $DC_i(X)$ to the local space spanned by Y_i using image computation. Rather we should project the care set into the local input space and then take the complement. It is because the former may mistakenly include some care minterm in the local space if there exists some care minterm and don't care minterm in the global space mapping to the same image. On the other hand, notice that, even though SDC is absent from Formula (6.17), it has been implicitly computed in the image computation.

With the local don't cares $D_i(Y_i)$ of node i, we can minimize the SOP expression of node i using two-level logic minimization methods. The don't-care generation and logic minimization procedure can be summarized as follows.

1. Select a node i in the Boolean network.
2. Compute its local don't care set D_i.
3. Minimize the cover of node i with respect to D_i.

Therefore by treating a multilevel netlist as a network of PLAs, two-level minimization methods can be applied as a baseline tool for multilevel logic minimization.

The above computation assumes that the rest of the Boolean network is not changed. One generalization is to consider *compatible don't cares* among multiple nodes simultaneously. Since the don't care conditions of different nodes may be conflicting with each other, they must be made compatible. The high computational complexity however restricts the application of compatible don't cares. Often a network is iteratively optimized one node at a time with respect to its local don't cares.

Complete Flexibility. The characterization of don't cares, including SDC, ODC, and XDC, can be unified through the concept of *complete flexibility* [Mishchenko 2002]. The *complete flexibility* (CF) of a node in a Boolean network is a Boolean relation that characterizes the set of all possible input-output behaviors of the node assuming that the rest of the network is not changed. The complete flexibility subsumes all the above don't cares. In addition, it is more powerful in capturing non-determinism, and can be generalized for a non-deterministic Boolean network [Mishchenko 2006a] where each node represents some relation allowing one-to-many mappings, not possible for functions.

Consider computing the complete flexibility of node i in a Boolean network N. Let $S(X, Z)$, given from specification, be the *specification relation* specifying all the allowed input-output behavior of the Boolean network. Hence $S(X, Z)$ subsumes XDC. Let

$$E_i(X, Y_i) \;=\; \bigwedge_{y_j \in Y_i} \left(y_j \equiv g_j{}'(X)\right)$$

be the *environment relation* characterizing the set of consistent assignments on variables X and Y_i. Hence $\neg E_i(X, Y_i)$ subsumes the SDC of node i. Let

$$I_i(X, y_i, Z) = \bigwedge_{y_j \in Z} \left(y_j \equiv g_j{}'(X, y_i)\right)$$

be the *influence relation* characterizing the allowed valuations on y_i consistent with those on X and Z, where $g_j{}'$ is a primary output function of network N', same as that obtained in the ODC computation. Hence

$$R_i(X, y_i) = \forall Z.[I_i(X, y_i, Z) \Rightarrow S(X, Z)]$$

subsumes the ODC of node i. The complete flexibility CF_i of node i in terms of the local input variables Y_i can be obtained by

$$\begin{aligned} CF_i(Y_i, y_i) &= \forall X.[E_i(X, Y_i) \Rightarrow R_i(X, y_i)] \\ &= \forall X.[E_i(X, Y_i) \Rightarrow \forall Z.[I_i(X, y_i, Z) \Rightarrow S(X, Z)]] \\ &= \forall X, Z.[\neg E_i(X, Y_i) \vee \neg I_i(X, y_i, Z) \vee S(X, Z)] \\ &= \forall X, Z.\neg[E_i(X, Y_i) \wedge I_i(X, y_i, Z) \wedge \neg S(X, Z)] \end{aligned} \tag{6.18}$$

ROBDD Implementation. Notice that all of the above computations can be realized using ROBDDs as operations over Boolean functions.

6.3.5 Advanced subjects

AIG-based Multilevel Logic Minimization. In addition to the division-based transformations, we may approach the multilevel logic minimization problem with a new view using the AIG representation.

Any Boolean expression can be converted into an AIG in polynomial time while structural hashing can be applied during the AIG construction. The obtained AIG can then be further simplified through *rewriting* [Bjesse 2004; Mishchenko 2006b]. This simplification is in terms of AIG nodes and/or levels, rather than the conventional literal or cube counts.

By grouping the nodes of the AIG into clusters (such that each cluster consists of a set of connected nodes rooted at some node producing its output, and the fanins of a cluster are outputs of some other clusters), each cluster can be seen as a complex logic node in a Boolean network. Therefore an AIG can be considered as a data structure that encompasses a set of multilevel logic netlists subject to different interpretations of cluster boundaries, called **cuts**. Given an AIG, the problem of multilevel logic minimization now boils down to the enumeration of good cuts, see, *e.g.*, [Ling 2007; Mishchenko 2007b]. This approach to logic minimization is taken by the ABC package [ABC 2005].

Sequential Logic Minimization. The aforementioned combinational logic minimization methods can be applied to simplify sequential circuits. For a given sequential circuit, treating the register outputs as primary inputs and register inputs as primary outputs results in the combinational methods being applicable to sequential circuit optimization. The optimization, of course, does not take full advantage of sequential flexibilities.

We can in fact pursue more progressive logic transformations. *State minimization* [Kohavi 1978], *state encoding* [Villa 1997], and logic minimization using *unreachable states* or *state equivalence* [Kohavi 1978] as don't cares, for example, are valid transformation methods because they do not change the input-output behavior of a sequential circuit. Furthermore, it is possible to characterize complete flexibility in the sequential domain [Yevtushenko 2001; Mishchenko 2005], similar to the combinational counterpart. In the computation, however, we have to manipulate finite automata, rather than Boolean formulas.

The above approaches are *state-based* in the sense that we have to know some state information for a given sequential circuit. The expensive derivation of state information limits their applicability to large designs. In contrast, there are *structure-based* transformations, which are carried out according to circuit structures and do not rely on state information. *Retiming* [Leiserson 1983, 1991] and *resynthesis* [Malik 1991], for example, are practical transformation methods for sequential logic minimization.

Although most designs are sequential and practical sequential optimization techniques are available, logic synthesis flows for the industrial design typically consist of only combinational optimization methods. This phenomenon can be attributed to the hardness of sequential circuit equivalence verification [Jiang 2006]. From the complexity viewpoint, sequential equivalence checking is PSPACE-complete, which is considered much harder than the coNP-complete combinational equivalence checking problem. In industrial practice, combinational equivalence checking is considered "solvable." (In fact, equivalence checking of industrial circuits with multi-million gates has been demonstrated [Kuehlmann 1997]. Of course there are special cases of combinational circuits that are hard to verify, *e.g.*, multipliers with different circuit structures.) On the contrary, for sequential equivalence checking, there are almost no good approaches that are general enough and work for the majority of practical test-cases. Making sequential circuit optimization scalable and verifiable is an important research subject.

6.4 TECHNOLOGY MAPPING

The logic optimization algorithms described thus far operate on Boolean networks. The optimization aims at simplifying logic expressions and is independent of the target implementation technology. To finish the logic synthesis steps, we need to implement logic gates with physical layouts. One solution to it is to perform

technology mapping, which is one of the most important tasks in technology dependent optimization. It takes on a technology-independently optimized logic netlist, and expresses the netlist using a set of pre-designed and pre-characterized gate layouts from a technology library. Typically, the goal is to make optimal use of all of the gates in the library to produce a circuit with minimum area subject to the delay constraint for critical-path delay no greater than a target value.

Technology mapping algorithms are constrained by the structure of the logic netlists produced by technology-independent optimization. It is not the role of technology mapping to change the structure of the circuit radically, for example, by finding common sub-expressions between two or more parts of the circuit. Likewise, it is not the main role of technology mapping to reduce the number of levels of logic along the critical path. The role of technology mapping is to make the actual gate choice to implement the logic netlist, for example, choosing the fastest gates along the critical path and using the most area-efficient combination of gates off the critical path.

A technology mapping algorithm should ideally achieve several goals. It should be able to adapt to a variety of different libraries because an algorithm which depends on characteristics of a particular library is of limited use, and an algorithm which is geared to a subset of the gates in a library is limited in its optimization potential. To practically achieve this goal of adaptability, a user must be able to provide new gates to the technology mapper without understanding its detailed operation, and these gates should be used effectively.

6.4.1 **Technology libraries**

The introduction of **gate arrays** and **standard cells** brought comparable benefits to IC designers. A gate array is an array of transistors and routing channels which can be configured into an IC through a metalization process during semiconductor fabrication. The metalization phases are used for cell definition, such as defining a NOR cell, and for interconnecting the cells. The electrical characteristics of cells after metalization have been carefully defined and are embodied in a databook. Standard cells are combinational and sequential logic gates whose electrical characteristics have been carefully defined and embodied in a library. Standard cells are similar to gate arrays in that they are precharacterized in a databook, but they offer additional degrees of freedom since they go through all the mask steps of semiconductor processing.

Logic gates of VLSI circuits, especially for ASICs, are usually restricted to be implemented by selections from a *technology library* of *gates*. A **gate** is a primitive element available in a particular implementation technology; a **technology library** is a collection of these gates. A technology library is assumed to consist of a finite collection of gates. For example, the gates in a static CMOS gate-array (or standard-cell) design typically include inverters, NAND gates, NOR gates, and a variety of complex gates, whereas the gates in an ***emitter-coupled logic*** (ECL) gate-array are typically NOR gates and XOR gates.

These libraries are typically composed of a few hundred gates and sequential elements like latches and flip-flops for which highly optimized layouts have been manually designed for a particular technology. Each gate is assigned a number of values associated with the different cost functions under which it will be optimized. For example, each gate is assigned a value called the *area* of the gate representing the physical area occupied by the gate. The logic designers are then restricted to using these gates in their logic circuits.

Example 6.67 The combinational subset of a very simple library is shown in Figure 6.26. The library cell names, associated area costs, their functions, and their representations in terms of two-input NAND (NAND2) gates and inverters (INV's) are shown.

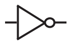
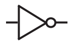
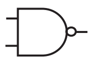
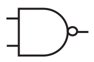
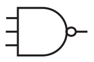
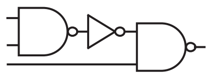
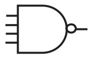
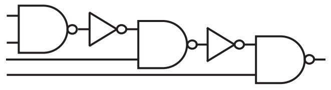
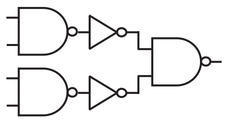
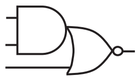
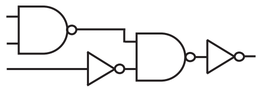
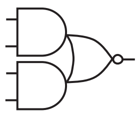
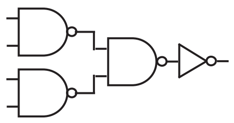
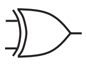
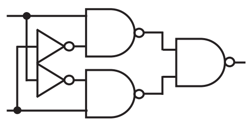
Gate	Cost	Symbol	Pattern DAG
INV	2		
NAND2	3		
NAND3	4		
NAND4	5		 
AOI21	4		
AOI22	5		
XOR	4		

FIGURE 6.26

Gate library.

Given a technology library, the problem of technology mapping is finding a multilevel circuit equivalent to the given Boolean network such that it is comprised of gates in the library and has minimum cost, which could be the area, delay, testability, or power consumption of the resulting circuit.

6.4.2 Graph covering

A systematic approach to technology mapping is based on the notion of graph covering. With this formulation, the technology mapping problem can be viewed as the optimization problem of finding a minimum cost covering of the **subject graph** by choosing from the collection of **pattern graphs** for all gates in the library. A **cover** is a collection of pattern graphs such that every node of the subject graph is contained in one or more of the pattern graphs. Moreover, one restriction of any cover is that the inputs of one pattern in the covering must be the outputs of some other pattern in the covering. Otherwise it would imply that the inputs of one pattern come from internal nodes in another pattern. As these internal signal values are not visible outside the pattern, any covering without such a restriction would not be meaningful.

Example 6.68 The cover shown in Figure 6.27a is legitimate while that in Figure 6.27b is not.

In graph covering, the Boolean network to be covered is often represented in a special form, where each gate is either of a NAND2 or an INV. It is termed the **subject graph**, or **subject DAG**. In addition to the Boolean network to be covered, each library gate is also represented in this special form. Each realization is termed a **pattern graph**, or **pattern DAG**. Note that a gate may have more than one associated pattern DAG.

Example 6.69 In Figure 6.26, the pattern DAGs of the library cells are shown. The NAND4 gate has more than one pattern DAGs.

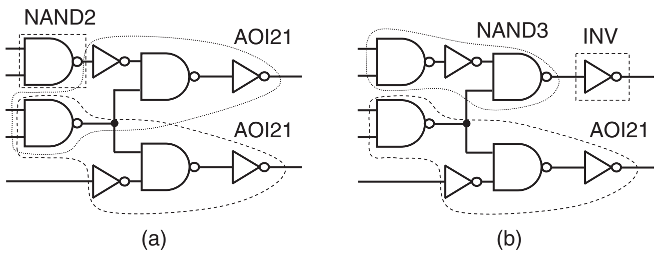


FIGURE 6.27

Graph coverings: (a) legal and (b) illegal.

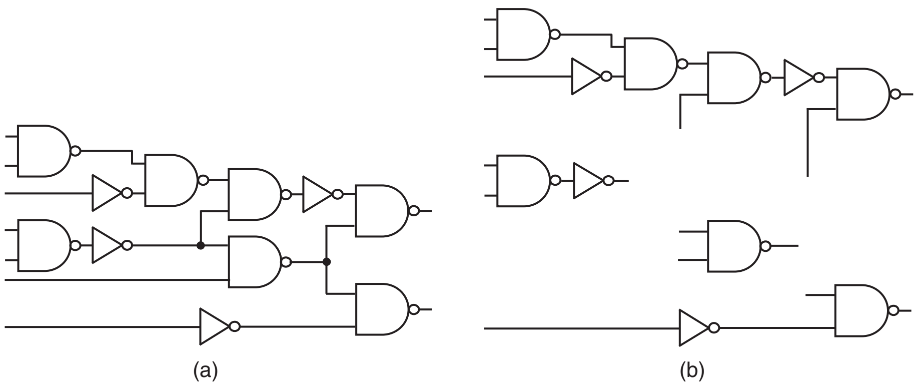


FIGURE 6.28

(a) Subject DAG example. (b) Subject DAG decomposed into a forest of trees.

Example 6.70 Figure 6.28a shows a subject DAG example.

The optimization problem of technology mapping can now be stated as: Find a minimum cost covering of the subject DAG by the pattern DAGs.

6.4.3 **Choice of atomic pattern set**

The choice of which atomic patterns to use for the subject and pattern graphs is an important consideration for graph covering algorithms. This decision influences the range of solutions for the covering problem and the number of patterns needed.

Why subject and pattern graphs are in terms of NAND2 and INV is motivated by the following observation. Adding additional functions such as a NOR2 gate, an AND2 gate, or an OR2 gate cannot provide higher-quality solutions; likewise, adding NAND, NOR, AND, or OR gates with more than two inputs cannot provide higher-quality solutions. This observation is based on the fact that given a cover for a subject graph using a larger set of functions, it is possible to show an equivalent cover where each function is replaced by an equivalent set of NAND2 gates and inverters.

Restricting ourselves to only a NAND2 gate and inverter does come at the price of increasing the number of patterns needed to represent some logic functions, as can be seen from the following example. Experience has shown that the increase in the number of patterns (and hence the increase in the memory and time required for technology mapping) is not significant.

Example 6.71 The logic function

$$f = \overline{a \cdot b \cdot c \cdot d + e \cdot f \cdot g \cdot h + i \cdot j \cdot k \cdot l + m \cdot n \cdot o \cdot p}$$

requires only one pattern corresponding to a tree of five NAND4 gates. However, representing all patterns for this same function using NAND2 gates and INV's requires 18 patterns.

6.4.4 Tree covering approximation

One technique (following the paradigm established in the domain of code generation [Aho 1976]) for solving the graph covering problem is to partition the subject graph into a forest of trees and solve the covering problem on each of the trees. A tree is a DAG where every node (including primary inputs) has a single fanout. The tree necessarily has a single sink (primary output) called the *root* and the sources (primary inputs) of the tree are called the *leaves* of the tree.

Example 6.72 The subject DAG of Figure 6.28a can be partitioned into a forest of trees as shown in Figure 6.28b.

The motivation for looking at the problem of tree covering is the existence of an efficient algorithm for the optimal tree covering problem [Keutzer 1987].

The application of the tree covering to technology mapping proceeds as follows. The first step is to convert the Boolean network into the NAND2-INV form, that is, every logic gate after the conversion is of type either NAND2 or INV. This subject DAG is then partitioned into a forest of trees by cutting the graph at each multiple-fanout stem. The resulting trees are optimally covered one tree at a time. Finding the optimum covering of a tree is done by generating the complete set of matches for each node in the tree (that is, the set of tree patterns which are candidates for covering a particular node) and then selecting the optimum match from among the candidates using a dynamic programming algorithm.

Example 6.73 Consider a Boolean network given by

$$\begin{aligned} Z &= X + \bar{Y} + h \\ Y &= W \cdot \bar{d} \\ X &= e \cdot f \cdot g, \text{ and} \\ W &= a \cdot b + c \end{aligned}$$

A NAND2-INV representation of the Boolean network is given in Figure 6.29a. The trivial covering of the subject DAG by pattern DAGs from the library of Figure 6.26 is also illustrated in Figure 6.29a. The cost of this trivial covering corresponds to the cost for seven NAND2 gates and five INV's, giving a cost of 31. A substantially better covering that exploits the larger gates in the library is shown in Figure 6.29b. The cost of this covering is the cost of two INV's, two NAND2's, one NAND3, and one NAND4 for a total cost of 19. A covering which utilizes an AOI gate with a lower cost of 17 is shown in Figure 6.29c.

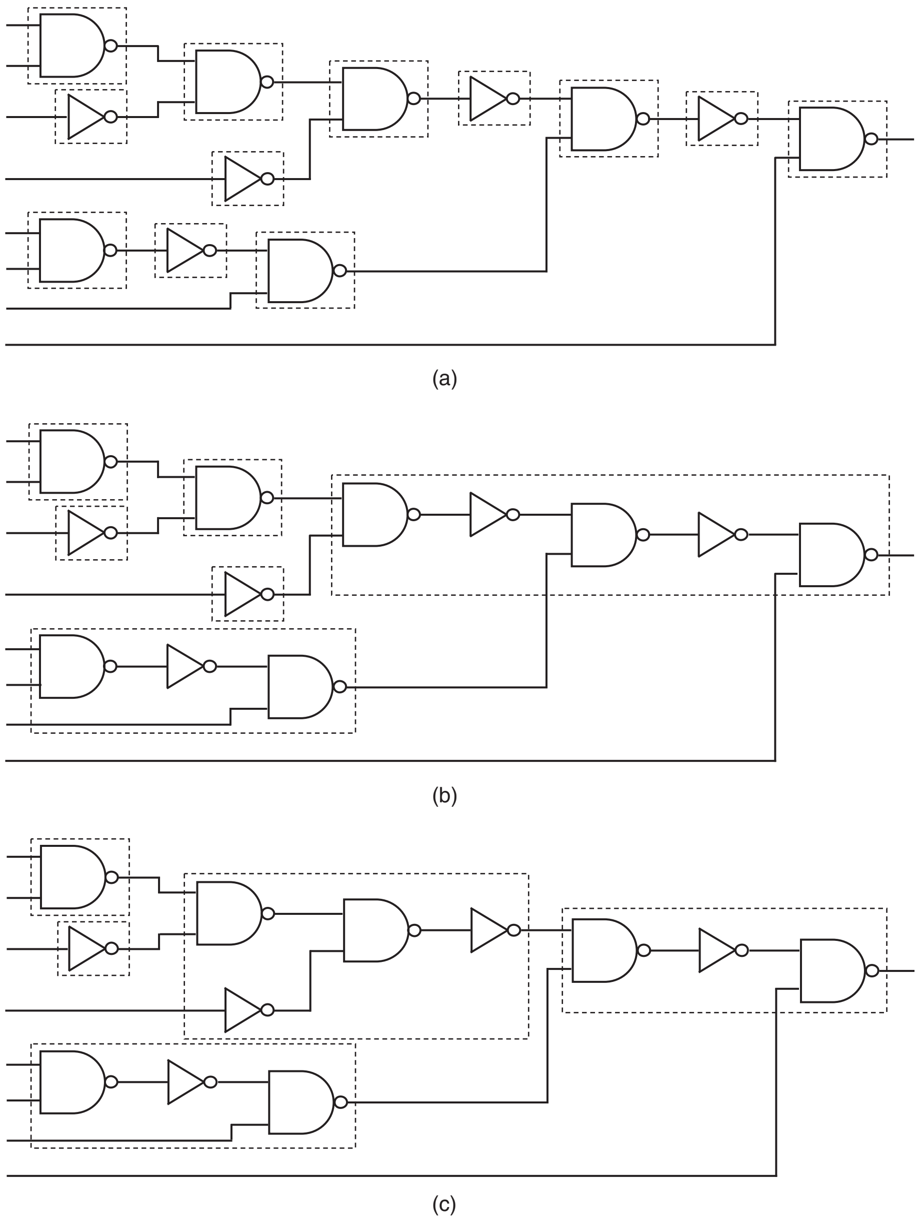


FIGURE 6.29

Tree coverings: (a) Trivial covering. (b) Better covering. (c) Optimum covering.

6.4.5 **Optimal tree covering**

A solution to establishing the initial set of candidate matches for a tree is to attempt to match each pattern at each node in the tree. If there are p patterns in the pattern set and n nodes in the subject graph, then this approach has complexity $O(n \cdot p)$.

Having generated a set of candidate matches for each node in the subject graph, an optimal tree cover must then be selected from among the candidates. Dynamic programming can be used for this purpose. Dynamic programming is a general technique for algorithm design which can be applied when the solution to a problem can be built from the solutions of a number of sub-problems.

Consider the problem of finding a minimum area cover for a subject tree T. A scalar cost is assigned to each tree pattern, and the cost for a cover is the sum of the costs for each pattern in the cover. The key observation is that the minimum-area cover for a tree T can be derived from the minimum-area covers for every node below the root of T. This is the ***principle of optimality*** for tree covering and is used as follows to find an optimal cover for T. For every match at the root of the tree the cost of an optimal cover containing that match equals the sum of the cost of the corresponding gate and the sum of the costs of the optimal covers for the nodes which are inputs to the match.[2] Note that the optimal covers for each input to the match at the root can be computed once and stored; it is not necessary to recompute the optimal cover for each input of each match.

Because each node in the tree is visited only once, the complexity of this algorithm is proportional to the number of nodes in the subject tree times the maximum number of matches at any node in the subject tree. The maximum number of matches is a function of the library size and is therefore a constant independent of the subject tree size. As a result the covering algorithm has linear complexity in the size of the subject tree, and the memory requirements are also linear in the size of the subject tree.

Example 6.74 We illustrate the optimum covering algorithm on the tree of Figure 6.30. We walk from the primary inputs to the primary output of the tree and determine the best match at each gate output. At each gate output, the match selected for the sub-tree whose root is the gate output has been shown along with the total cost of the optimal cover for this sub-tree. For the first-level gates, only NAND2 and INV matches are possible. At the output of gate 2 the only match is with a NAND2, and therefore the total cost is 8. At the output of gate 12 two matches are possible, with a NAND2 or with a NAND3. The former will result in a cost of 8, so we pick the latter which has a cost of 4. At the output of gate 4 the best match corresponds to an AOI gate with a cost of 9. The final cost at the primary output is 17. The optimum covering corresponds to that of Figure 6.29c.

[2]Recall the rules for legal coverings stated in Section 6.4.2.

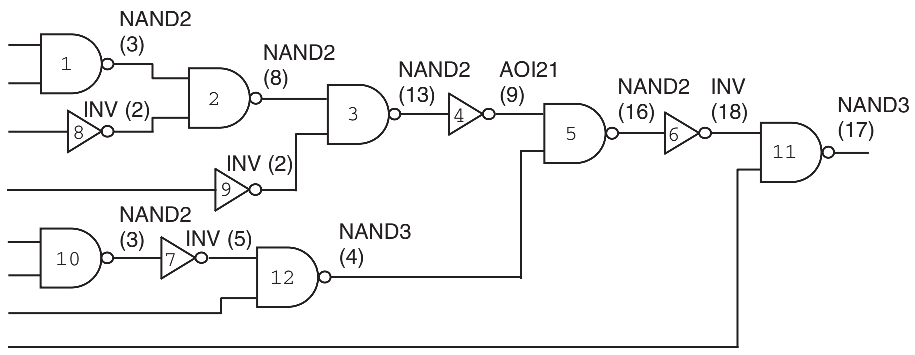


FIGURE 6.30

Dynamic programming for optimum tree covering.

6.4.6 Improvement by inverter-pair insertion

A simple way to improve the quality of circuits produced by the tree covering algorithm is by inserting inverter pairs. Redundant inverters are added to each tree to improve the number of patterns which can match at each node. This leads to an examination of more possible covers for each tree, leading directly to an improvement in the optimization quality.

The technique works as follows. Each edge in the subject tree and each edge in a pattern which connects two NAND gates is replaced with a pair of inverters. An extra pattern consisting of a pair of inverters is added to the matching patterns. This extra pattern is given zero area cost and zero delay cost. The tree covering algorithm is then applied unmodified.

Because of the optimality of the tree covering algorithm adding these extra inverters cannot lead to a cover with a greater cost. Each pair of inverters can be covered by the inverter-pair pattern, which leads to the solution which existed before the inverters were added. However, the advantage is that the tree covering algorithm is able to make the optimal choice between covering the extra inverters with the inverter-pair pattern at no cost or splitting the inverters between two patterns if this leads to a cover with less cost. The only disadvantage is that the number of nodes in the subject tree and the pattern trees has increased. The increase in the number of nodes is bounded by a factor of three (two extra inverter nodes for each node in the subject tree); however, the actual increase is typically less because redundant inverters are added only at the output of a NAND gate and not at the output of each inverter in the subject tree.

6.4.7 Extension to non-tree patterns

Some gates in a technology library cannot be represented in a tree form. Common examples are the XOR gate shown at the bottom of Figure 6.26, a two-to-one multiplexor, and a three-input majority gate (logic function $f = a \cdot b + a \cdot c + b \cdot c$). However, a simple extension allows these patterns to be included.

A leaf-DAG is a DAG where the only nodes with fanout greater than one are the primary inputs. Patterns which are trees, and patterns which are leaf-DAGs can be used directly by the tree covering algorithm. Hence the leaf-DAG patterns may include the XOR pattern shown in Figure 6.26. Note, however, that because of the multiple-fanout of one of these matches, the XOR gate must match at the leaves of the tree.

6.4.8 Advanced subjects

The success of the graph covering formulation has helped formulate the logic synthesis and optimization problem as an integration of technology-independent and technology-dependent portions. Graph covering based technology mapping is able to address a morass of technology specific issues, such as technology libraries and their area and timing characterization, which would significantly complicate higher level optimizations. The major limitation of graph covering, however, is its dependence on the structure of the given subject graph. This limitation was overcome in [Lehman 1997], where logic decomposition during technology mapping is proposed as a way of bridging the gap between technology-independent optimization and technology mapping. The approach was further developed in [Chatterjee 2006].

In our discussion, we focused on standard cell technology mapping. As the mapping algorithms heavily depend on the target implementation technology, different design styles may need different technology mapping methods. For instance, technology mapping for FPGAs [Scholl 2001], and even for standard cells [Kravets 2001], can be formulated very differently.

6.5 TIMING ANALYSIS

After correct logical functioning, the speed of an integrated circuit is one of the most important design characteristics. Timing optimization is thus an important aspect of logic synthesis. Any optimization system is only as good as the models that guide it, and as a result good timing optimization is entirely dependent on accurate timing analysis. For these reasons we spend a good deal of attention on techniques for accurate timing estimation of synchronous sequential circuits.

Accurate timing estimation relies on **component delay calculation** and **circuit delay calculation**. Component delay calculation is the method used for actually calculating the delay of individual components, such as gates and wires, within a circuit. In calculating gate delays, timing data such as the **inertial** and **propagation delays** of gates are typically gathered from extensive transistor-level and/or device-level simulation of the circuit components. In calculating wire delays, timing data arising from the parasitic capacitances and resistances of wires can be estimated through simulation or can be back-annotated from the final circuit layout. In our discussion we are mainly concerned about gate delays as wire delays can be embedded into the gate delays by the delay model to be introduced.

If we view a circuit as a graph, then the method used for delay calculation at the vertices of the graph is gate delay calculation while circuit delay calculation is the model used for calculating delay for the entire graph.

Below we present a simple gate delay model and then focus on the topic of circuit delay calculation, which is the most challenging and relevant problem in timing estimation for the developer of a logic optimization system.

Gate Delay Model. A popular (CMOS) gate delay model is a simple linear model [Sutherland 1999]: The delay T_d of a gate g is given by the equation

$$T_d = T_p + T_e \times \frac{C_{out}}{C_{in}} \tag{6.19}$$

where T_p is the **parasitic delay** of the gate, T_e is the **logical effort**, C_{in} is the input capacitance, and C_{out} is the capacitive load at the gate output. It does not consider more refined details such as the effect of slow rising or falling transitions on the transistors associated with this gate. In this model, parameters T_p, T_e, and C_{in} are fixed constants for a standard cell whereas C_{out} varies depending on the fanout load of a gate (which may include wiring capacitances).

Gate delay calculations are performed extensively in timing analysis and logic optimization, and as a result tradeoffs have evolved between the accuracy of a model and the runtime of calculation. Although Equation (6.19) is a simple approximation, it is good enough for logic optimization purposes. More accurate nonlinear models are possible and often stored as look-up tables. Delay calculation often depends on the circuit implementation method.

Circuit Delay Calculation. We explain how to use gate delay calculation to compute the delay of an entire synchronous circuit. A simple implementation model of a clocked, or synchronous, sequential circuit is shown in Figure 6.31, where a clocked memory element (register), *e.g.*, an edge-triggered flip-flop, is used. At each active clock edge the next state is loaded into the flip-flops and becomes the current state.

Registers have a **propagation delay** associated with the interval between a clock edge and valid outputs. In order to guarantee that an input is not sampled when invalid, a period of validity extending slightly before and after the active edge is specified. Specification of a **setup time** t_s and **hold time** t_h dictates that the register inputs must be valid and stable during a period that begins t_s before the active clock edge and ends t_h after the edge.

Given a sufficiently long clock period and appropriate constraints on the timing of transitions on the inputs, the inputs to the flip-flops can be guaranteed to be stable at each active clock edge, ensuring correct operation. Correct operation depends on the assumptions that:

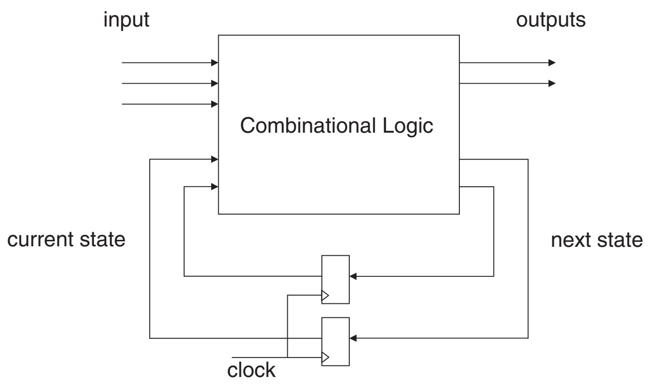


FIGURE 6.31

Clocked model for a sequential circuit.

1. The clock period is longer than the sum of the maximum propagation delay through the combinational logic, the setup time of the registers, and the maximum propagation delay through the registers.
2. The circuit's input signals are stable and valid for a sufficient period surrounding each active clock edge to accommodate both the maximum propagation delay through the combinational logic and the setup time of the registers.
3. The minimum propagation delay through the combinational logic exceeds the hold time requirement of the registers.

The most important constraint above is the first one. The length of the clock period of a sequential circuit is directly related to the maximum propagation delay through the combinational logic of the circuit.

Given that the delay calculation of the sequential circuit primarily depends on the delay of the combinational logic, we will focus on the problem of correctly computing the maximum propagation delay of a multilevel combinational circuit. We will show in the next section how to optimize a circuit so as to minimize the delay through the circuit.

For some time the most common approach to estimating and validating the delay of a synchronous circuit was **timing simulation**. The approach is diminishing in utility because of the incompleteness and excessiveness of input stimuli required to accurately determine circuit performance. Instead, **timing verification** is being used for validating the timing of circuits, and we will focus exclusively on using timing verification for estimating and validating the timing of a synchronous circuit.

Terminology. Before delving into timing analysis, we introduce terminology that will allow us to discuss timing issues. A combinational circuit can be viewed as a DAG $G = (V, E)$ where vertices or nodes V in the graph correspond to gates in the circuit and edges E correspond to connections in the circuit. Primary inputs are **sources** $\subseteq V$ while primary outputs are **sinks** $\subseteq V$. A **path** in a combinational circuit is an alternating sequence of vertices and edges, $\{v_0, e_0, \ldots, v_n, e_n, v_{n+1}\}$, where edge $e_i = (v_i, v_{i+1})$, $1 \leq i \leq n$, connects the output of vertex v_i to an input of vertex v_{i+1}. For $1 \leq i \leq n$, v_i is a gate g_i, v_0 is a primary input, and v_{n+1} is a primary output. Each e_i is a wire (or a two-terminal net) in the actual circuit.

Let $p = \{v_0, e_0, \ldots, v_n, e_n, v_{n+1}\}$ be a path. The inputs of v_i other than e_{i-1} are referred to as the **side-inputs** to p, that is, the set of signals not on p but feeding to the gates on p.

Each gate g_i (or wire e_i) is assumed to have a delay which can be a fixed quantity under the **fixed delay model** or can vary in a given range under the **monotone speedup delay model**.

A **controlling value** at a gate input is the value that determines the value at the output of the gate independent of the other inputs. For example, 0 is a controlling value for an AND gate. A **non-controlling value** at a gate input is the value which is not a controlling value for the gate. For example, 1 is a non-controlling value for an AND gate. We say that a gate g has the **controlled value** if one of its inputs has a controlling value; otherwise, we say that g has the **non-controlled value.**

Path sensitization studies the conditions under which signals can propagate from the primary inputs to the primary outputs of a combinational circuit. The conditions depend on the delay models and modes of operation assumed for the circuit.

We will precisely characterize the delay of a multilevel logic circuit, and see that the delay of a multilevel circuit depends on various assumptions relating to the mode of operation of the circuit and the delay model chosen. We begin with the simplest **topological timing analysis**, which is conservative but sound. The complexity of the analysis is linear in the circuit size. We will then introduce **functional timing analysis**, which is accurate at the cost of computation overhead.

6.5.1 Topological timing analysis

Most timing analyzers fall into *the topological timing analysis* category, where the topologically longest path in the circuit is assumed to dictate the critical delay of the circuit. We describe a topological timing analyzer that determines the longest path in the circuit without regard to the Boolean functionality of the circuit.

Circuit speed is measured by most optimization systems using a fixed delay model, where each gate and wire in the network has a given and fixed delay. Typically, a worst-case design methodology is followed, where the given delay for the gate is an upper bound on the actual delay of the fabricated gate.

The **arrival time** of a signal s, denoted A_s, is the time at which the signal settles to its steady state value. For a given circuit, using the arrival times of the primary inputs we can compute the arrival time of every signal in the circuit. For a gate in the circuit, the arrival time of the gate output equals the maximum among the arrival times of the gate inputs plus the gate delay. That is, the arrival time of the output signal o of a gate g with gate delay d can be computed by

$$A_o = \max_{i \in FI(g)} \{A_i\} + d$$

where $FI(g)$ denotes the set of fanin signals of g.

The **required time** of a signal s, denoted R_s, is the time at which the signal is required to be stable. For a given circuit, using the required times of the primary outputs we can compute the required time of every signal in the circuit. For a gate in the circuit, the required time of any input of the gate equals the minimum among the required times of the gate outputs minus the gate delay. That is, the required time of any input signal i of a gate g with gate delay d can be computed by

$$R_i = \min_{o \in FO(g)} \{R_o\} - d$$

where $FO(g)$ denotes the set of fanout signals of g.

The **slack time** of a signal s, denoted S_s, is the difference between its required time and arrival time, *i.e.*,

$$S_s = R_s - A_s$$

The slack value of a signal measures its looseness in terms of timing criticality. Negative slack values indicate timing violation.

Starting with the primary input arrival times, we can compute the arrival time for every signal in a **topological order** from primary inputs to primary outputs. Similarly, using the primary output required times, we can compute the required times for every signal in a reverse topological order from primary outputs to primary inputs. Thus the slack at each node can be obtained as well.

Example 6.75 The arrival time, required time, and slack of each signal in Figure 6.32 are shown as a 3-tuple. We are given the arrival times for the four primary inputs and the required time for the output. The delay of each node is indicated within the node. The arrival time of signal `e` is the maximum of the arrival times of primary inputs `a` and `b` (= 1) plus the delay of the node (= 1), equaling 2. Similarly the arrival times of the other signals can be calculated. On the other hand, given a required time of 8 at output `h`, the required times forsignals `f` and `g` can be computed as 8 minus the delay of the output node (= 2), equaling 6. However, given the required time of 6 at `f`, the required times at signals `e` and `g` are calculated to be 4. The required time for signal `g` is the minimum of the computed required times, namely 4. This is intuitive because, if `g` does not stabilize by time 4, `f` will not stabilize by time 6 and the output `h` will not stabilize by time 8. Similarly, the required times at the other signals can be calculated.

The **topologically longest path** of a circuit is a path where each signal has the minimum slack. Static timing analyzers assume that the **critical delay** of the circuit is the delay of the topologically longest path. Under this (pessimistic) assumption the longest path is also called the **critical path**.

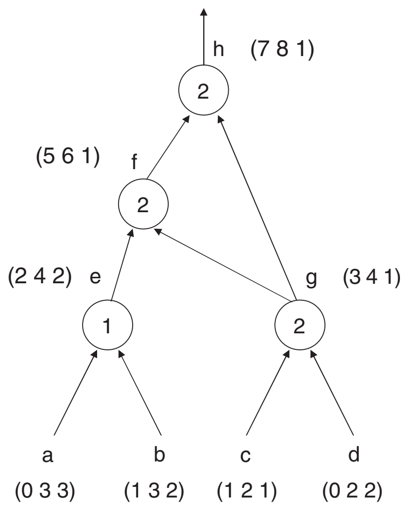


FIGURE 6.32

Topological timing analysis.

6.5.2 **Functional timing analysis**

The problem with topological analysis of a circuit is that not all critical paths in a circuit need be responsible for the circuit delay. Critical paths in a circuit can be **false**, *i.e.*, not responsible for the delay of a circuit. The **critical delay** of a circuit is defined as the delay of the longest **true** path in the circuit. Thus, if the topologically longest path in a circuit is false, then the critical delay of the circuit will be less than the delay of the longest path. The critical delay of a combinational logic circuit is dependent on not only the topological interconnection of gates and wires, but also the Boolean functionality of each node in the circuit. Topological analysis only gives a conservative upper bound on the circuit delay.

Example 6.76 Assume the fixed delay model, and consider the carry bypass circuit of Figure 6.33. The circuit uses a conventional ripple-carry adder (the output of gate `11` is the ripple-carry output) with an extra AND gate (gate `10`) and an additional multiplexor. If the propagate signals `p0` and `p1` (the outputs of gates `1` and `3`, respectively) are high, then the carry-out of the block `c2` is equal to the carry-in of the block `c0`. Otherwise it is equal to the output of the ripple-carry adder. The multiplexor thus allows the carry to skip the ripple-carry chain when all the propagate bits are high. A carry-bypass adder of arbitrary size can be constructed by cascading a set of individual carry-bypass adder blocks, such as those of Figure 6.33.

Assume the primary input `c0` arrives at time $t = 5$ and all the other primary inputs arrive at time $t = 0$. Let us assign a gate delay of 1 for AND and OR gates and gate delays of 2 for the XOR gates and the multiplexor. The longest path including the late

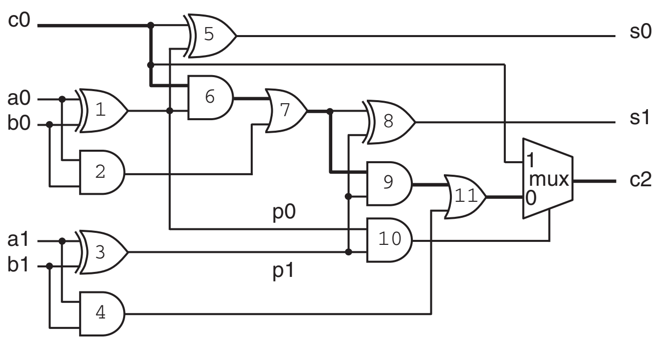


FIGURE 6.33

2-bit carry-bypass adder.

arriving input in the circuit is the path shown in bold, call it *P*, from `c0` to `c2` through gates 6, 7, 9, 11, and the multiplexor (the delay of this path is 11). A transition can never propagate down this path to the output because in order for that to happen the propagate signals have to be high, in which case the transition propagates along the bypass path from `c0` through the multiplexor to the output. This path is false since it cannot be responsible for the delay of the circuit.

For this circuit, the path that determines the worst-case delay of `c2` is the path from `a0` to `c2` through gates 1, 6, 7, 9, 11, and the multiplexor. The output of this critical path is available after 8 gate delays. The critical delay of the circuit is 8 and is less than the longest path delay of 11.

6.5.2.1 *Delay models and modes of operation*

Whether a path is a true or false delay path closely depends on the **delay model** and the **mode of operation** of a circuit.

In the commonly used **fixed delay model**, the delay of a gate is assumed to be a fixed number *d*, which is typically an upper bound on the delay of the component in the fabricated circuit. In contrast, the **monotone speedup delay model** takes into account the fact that the delay of each gate can vary. It specifies the delays as an interval $[0, d]$, with the lower bound 0 and upper bound *d* on the actual delay.

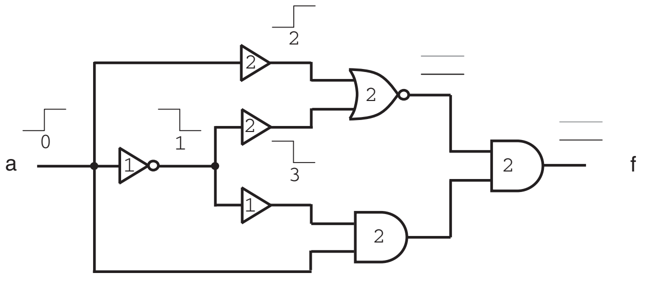
Consider the operation of a circuit over the period of application of two consecutive input vectors υ_1 and υ_2. In the **transition mode** of operation, the circuit nodes are assumed to be ideal capacitors and retain their values set by υ_1 until υ_2 forces the voltage to change. Thus, the timing response for υ_2 is also a function of υ_1 (and possibly other previously applied vectors). In contrast, in the **floating mode** of operation the nodes are not assumed to be ideal capacitors, and hence their state is unknown until it is set by υ_2. Thus, the timing behavior for υ_2 is independent of υ_1.

Transition Mode and Monotone Speedup. In our analysis of the carry-bypass adder we assumed fixed delays for the different gates in the circuit and applied a vector pair to the primary inputs. It was clear that an event (a signal transition, either 0 → 1 or 1 → 0) could not propagate down the longest path in the circuit. A precise characterization is that the path cannot be **sensitized,** and thus false, under the transition mode of operation and under (the given) fixed gate delays. Varying the gate delays in Figure 6.33 does not change the sensitizability of the path shown in bold.

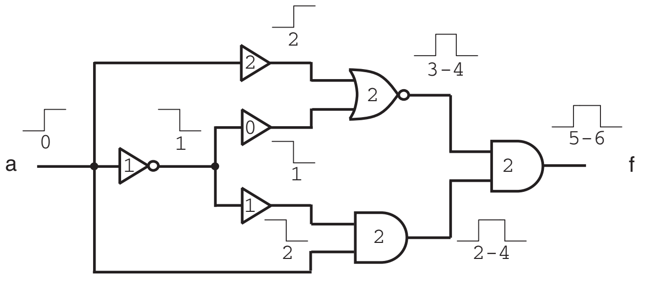
False path analysis under the fixed delay model and the transition mode of operation, however, may be problematic as seen from the following example.

Example 6.77 Consider the circuit of Figure 6.34a, taken from [McGeer 1989]. The delays of each of the gates are given inside the gates. In order to determine the critical delay of the circuit we will have to simulate the two vector pairs corresponding to `a`, making a 0 → 1 transition and a 1 → 0 transition. Applying 0 → 1 and 1 → 0 transitions on `a` does not change the output `f` from 0. Thus, one can conclude that the circuit has critical delay 0 under the transition mode of operation for the given fixed gate delays.

Now consider the circuit of Figure 6.34b which is identical to the circuit of Figure 6.34a except that the buffer at the input to the NOR gate has been sped up from 2 to 0. We might expect that speeding up a gate in a circuit would not increase the critical delay of a circuit. However, for the 0 → 1 transition on `a`, the output `f` switches both at time 5 and time 6, and the critical delay of the circuit is 6.



(a)



(b)

FIGURE 6.34

Transition mode with fixed delays.

This example shows that a sensitization condition based on transition mode and fixed gate delays is unacceptable in the **worst-case design methodology**, where we are given the upper bounds on the gate delays and are required to report the (worst-case) critical path in the circuit. Unfortunately, if we use only the upper bounds of gate delays under the transition mode of operation, an erroneous critical delay may be computed.

To obtain a useful sensitization condition, one strategy is to use the transition mode of operation and monotone speedup as the following example illustrates.

Example 6.78 Consider the circuit of Figure 6.35, which is identical to the circuit of Figure 6.34a, except that each gate delay can vary from 0 to its given upper bound. As before, in order to determine the critical delay of the circuit, we will have to simulate the two vector pairs corresponding to a making a 0 → 1 transition and a 1 → 0 transition. However, the process of simulating the circuit is much more complicated since the transitions at the internal gates may occur at varying times. In the figure, the possible combinations of waveforms that appear at the outputs of each gate are given for the 0 → 1 transition on a. For instance, the NOR gate can either stay at 0 or make a 0 → 1 → 0 transition, where the transitions can occur between [0, 3] and [0, 4], respectively. In order to determine the critical delay of the circuit, we scan all the possible waveforms at output f and find the time at which the last transition occurs over all the waveforms. This analysis provides us with a critical delay of 6.

Timing analysis for a worst-case design methodology can use the above strategy of monotone speedup delay simulation under the transition mode of operation. The strategy however has several disadvantages. Firstly, the search space is 2^{2n} where n is the number of primary inputs to the circuit, since we may have to simulate each possible vector pair. Secondly, monotone speedup delay simulation is significantly more complicated than fixed delay simulation. These difficulties have motivated delay computation under the floating mode of operation.

Floating Mode and Monotone Speedup. Under floating mode, the delay is determined by a single vector. As compared to transition mode, critical delay under floating mode is significantly easier to compute for the fixed or monotone speedup delay model because large sets of possible waveforms do not need

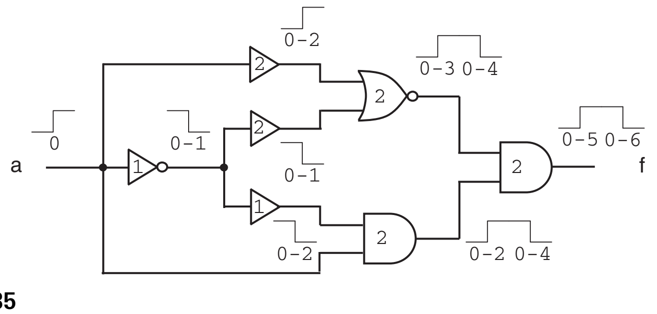


FIGURE 6.35

Transition mode with monotone speedup.

to be stored at each gate. Single-vector analysis and floating mode operation, by definition, make pessimistic assumptions regarding the previous state of nodes in the circuit. The assumptions made in floating mode operation make the fixed delay model and the monotone speedup delay model equivalent.[3]

6.5.2.2 *True floating mode delay*

The necessary and sufficient condition for a path to be responsible for circuit delay under the floating mode of operation is a delay-dependent condition.

The fundamental assumptions made in single-vector delay-dependent analysis are illustrated in Figure 6.36. Consider the AND gate of Figure 6.36a. Assume that the AND gate has delay d and is embedded in a larger circuit, and a vector pair $\langle \upsilon_1, \upsilon_2 \rangle$ is applied to the circuit inputs, resulting in a rising transition occurring at time t_1 on the first input to the AND gate and a rising transition at time t_2 on the second input. The output of the gate rises at a time given by $\max\{t_1, t_2\} + d$. The abstraction under floating mode of operation only shows the value of υ_2. In this case a 1 arrives at the first and second inputs to the AND gate at times t_1 and t_2, respectively, and a 1 appears at the output at time $\max\{t_1, t_2\} + d$. Similarly, in Figure 6.36b two falling transitions at the AND gate inputs result in a falling transition at the output at a time that is the minimum of the input arrival times plus the delay of the gate.

Now consider Figure 6.36c, where a rising transition occurs at time t_1 on the first input to the AND gate and a falling transition occurs at time t_2 on the second input. Depending on the relationship between t_1 and t_2 the output will either stay at 0 (for $t_1 \geq t_2$) or glitch to a 1 (for $t_1 < t_2$). It is possible to accurately determine whether the AND gate output is going to glitch or not if a simulation is carried out to determine the range of values that t_1 and t_2 can have on $\langle \upsilon_1, \upsilon_2 \rangle$. (This was illustrated in Figure 6.35.) However, under the floating mode of operation we only have the vector υ_2. The 1 at the first input to the AND gate arrives at time t_1, and the 0 at the second input arrives at time t_2. The output of the AND on υ_2 obviously settles to 0 on υ_2, but at what time does it settle? If $t_1 \geq t_2$, then the output of the gate is always 0, and the 0 effectively arrives at time

[3]To understand this effect, consider a circuit C with fixed values on its gate delays. Let p be a path through C and υ be a vector applied to C. In order to determine if p is responsible for the delay of C on v, we inspect the side-inputs of p. At any gate g on p, the side-inputs have to be at non-controlling values when the controlling or non-controlling value propagates along p through g. If the value at a side-input i to g is non-controlling on v, monotone speedup (under the transition or floating mode) allows us to disregard the time that the non-controlling value arrives, since we can always assume that it arrives before the value along p. Let the delay of all paths from the primary inputs to i be greater than the delay of the sub-path corresponding to p ending at g. Under monotone speedup, we can speed up all the paths to i, ensuring that the non-controlling value arrives in time. Under floating mode with fixed delays we cannot change the delays of the paths to i, but we can assume that υ_1, the vector applied before υ, was providing a non-controlling value! We do not have to wait for υ to provide the non-controlling value. In either case, the arrival time of non-controlling values on side-inputs does not matter.

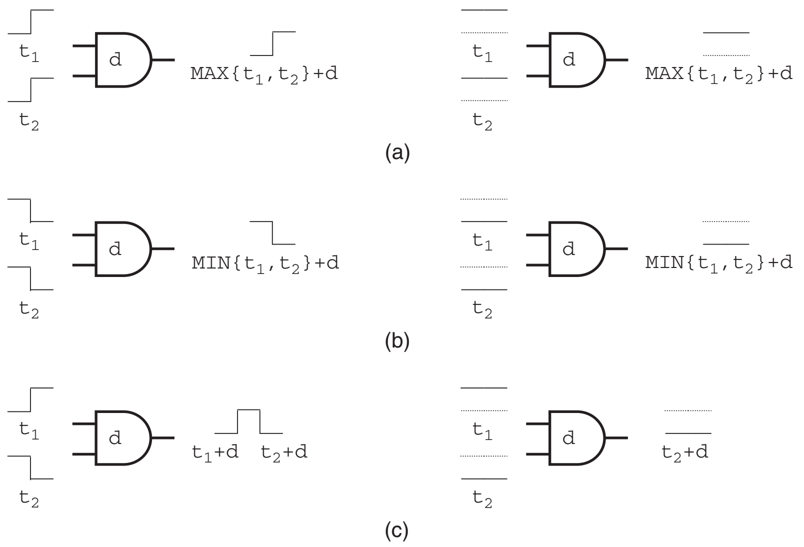


FIGURE 6.36

Fundamental assumptions made in floating mode operation.

0. If $t_1 < t_2$, then the gate output becomes 0 at $t_2 + d$. In order not to underestimate the critical delay of a circuit all single-vector sensitization conditions *have* to assume that the 1 (the non-controlling value for the AND gate) arrives before the 0 (the controlling value for the AND gate), *i.e.*, that $t_1 < t_2$. Under the floating mode of operation this corresponds to assuming that the values on the previous vector υ_1 were non-controlling. (The above assumption also captures the essence of transition mode delay under the monotone speedup delay model. Given that the AND gate is embedded in a circuit, under the monotone speedup model the sub-circuit that is driving the first input can be sped up to cause the rising transition to arrive before the falling transition.)

The rules in Figure 6.36 represent a timed calculus for single-vector simulation with delay values that can be used to determine the correct floating mode delay of a circuit under an applied vector υ_2 (assuming pessimistic unknown values for υ_1) and the paths that are responsible for the delay under υ_2. The rules can be generalized as follows:

1. If the gate output is at a controlling value, pick the minimum among the delays of the controlling values at the gate inputs. (There has to be at least one input with a controlling value. The non-controlling values are ignored.) Add the gate delay to the chosen value to obtain the delay at the gate output.
2. If the gate output is at a non-controlling value, pick the maximum of all the delays at the gate inputs. (All the gate inputs have to be at non-controlling values.) Add the gate delay to the chosen value to obtain the delay at the gate output.

To determine whether a path is responsible for floating mode delay under a vector υ_2, we simulate υ_2 on the circuit using the timed calculus. As shown in [Chen 1991], a path is responsible for the floating mode delay of a circuit on υ_2 if and only if for each gate along the path:

1. If the gate output is at a controlling value, then the input to the gate corresponding to the path has to be at a controlling value and furthermore has to have a delay no greater than the delays of the other inputs with controlling values.
2. If the gate output is at a non-controlling value, then the input to the gate corresponding to the path has to have a delay no smaller than the delays at the other inputs.

Let us apply the above conditions to determine the delay of the following circuits.

Example 6.79 Consider the circuit of Figure 6.34a reproduced in Figure 6.37. Applying the vector a = 1 sensitizes the path of length 6 shown in bold, illustrating that the sensitization condition takes into account monotone speedup (unlike transition mode fixed delay simulation). Each wire has both a logical value and a delay value (in parentheses) under the applied vector.

Example 6.80 Consider the circuit of Figure 6.38. Applying the vector `(a, b, c)` = (0, 0, 0) gives a floating mode delay of 3. The paths {`a, d, f, g`} and {`b, d, f, g`} can be seen to be responsible for the delay of the circuit.

Example 6.81 Consider the circuit of Figure 6.39. Applying a = 0 and a = 1 results in a floating mode delay of 5.

We presented informal arguments justifying the single-vector abstractions of Figure 6.36 to show that the derived sensitization condition is necessary and sufficient for a path to be responsible for the delay of the circuit under the floating mode of operation. For a topologically oriented formal proof of the necessity and sufficiency of the derived condition, see [Chen 1991].

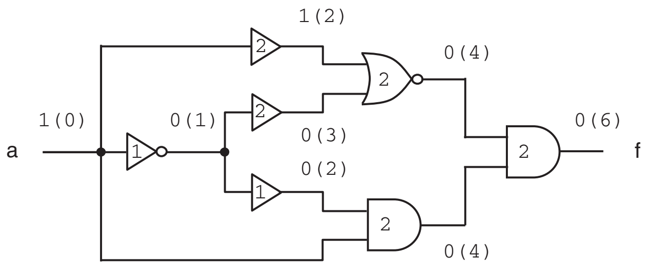


FIGURE 6.37

First example of floating mode delay computation on a circuit.

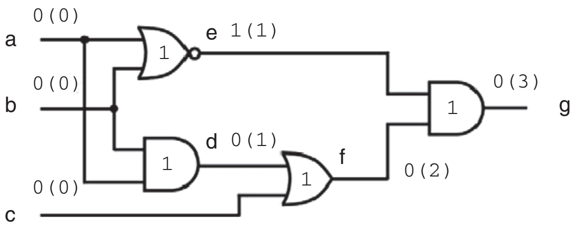


FIGURE 6.38

Second example of floating mode delay computation on a circuit.

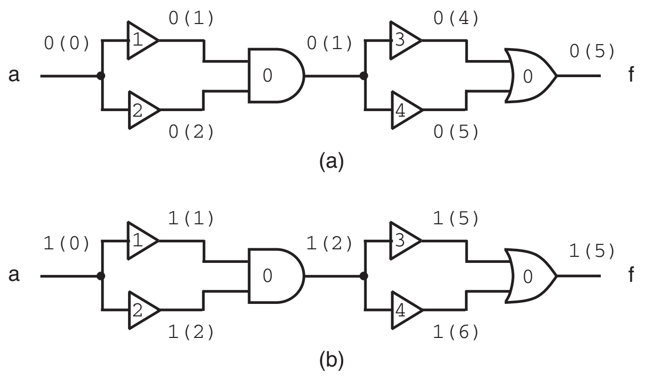


FIGURE 6.39

Third example of floating mode delay computation on a circuit.

6.5.3 **Advanced subjects**

There has been significant research done in an effort to arrive at the correct sensitization criterion in the late 1980s and early 1990s. A detailed history may be found in [McGeer 1991]. The computation of true critical delay of a circuit can be formulated with satisfiability solving [McGeer 1991; Guerra E Silva 2002] or **timed automatic test pattern generation** [Devadas 1992].

As for sequential circuit timing analysis, depending on the register types (*e.g.*, edge-triggered flip-flops and level-sensitive latches) and the number of clock phases used, their timing correctness requires careful analysis and verification. On the other hand, for IC manufacturing in the nanometer regime, process variations may cause substantial variations in circuit performance. This fabrication imperfection has motivated the development of **statistical static timing analysis** in replacement of the traditional (worst-case) static timing analysis (*i.e.*, the presented topological timing analysis). A good introduction to sequential circuit timing analysis and statistical static timing analysis can be found in [Sapatnekar 2004].

6.6 TIMING OPTIMIZATION

Being able to meet timing requirements is absolutely essential in synthesizing logic circuits. Timing optimization of combinational circuits can be performed both at the technology-independent level and during technology mapping. We consider the restructuring operations used in logic synthesis systems to improve circuit speed. We give an overview of basic restructuring methods that take into account timing constraints specified as input-arrival times of the primary inputs and output-required times of the primary outputs. The goal is to meet the timing constraints while keeping the area increase to a minimum. The methods use topological timing analysis, described in Section 6.5.1, to compute arrival times, required times, and slack times. Topological timing analysis is typically deployed in timing optimization tools due to its simple and fast calculation; functional timing analysis, in contrast, is mostly used for timing verification purposes instead due to its expensive computation cost.

6.6.1 Technology-independent timing optimization

For a given circuit to be delay minimized, the timing constraints are specified as the arrival times at the primary inputs and required times at the primary outputs. The optimization algorithm manipulates the network topology to achieve improved speed until the timing constraints are satisfied or no further decrease in the delay can be achieved.

The **critical section** of a Boolean network is composed of all the critical paths from primary inputs to primary outputs. Given a critical path, the total delay on the path can be reduced if any section of the path is sped up. **Collapsing** and **redecomposition** are the basic steps taken in restructuring. The nodes along the critical paths chosen to be collapsed and redecomposed form the **redecomposition region**.

Example 6.82 In Figure 6.40a we have a critical path `{a, x, y}`. The critical path can be reduced by first collapsing `x` and `y` and then redecomposing `y` in a different way to minimize the critical path as shown in Figure 6.40b.

Since a critical section usually consists of several overlapping critical paths, we select a minimum set of subsections, called **redecomposition points**, which when sped up will reduce the delays on all of the critical paths. (Note that it is not always possible to do so.) A weight is assigned to each candidate redecomposition point to account for possible area increase and for the total number of redecomposition points required. The goal is to select a set of points which cut all the critical paths and have the minimum total weight.

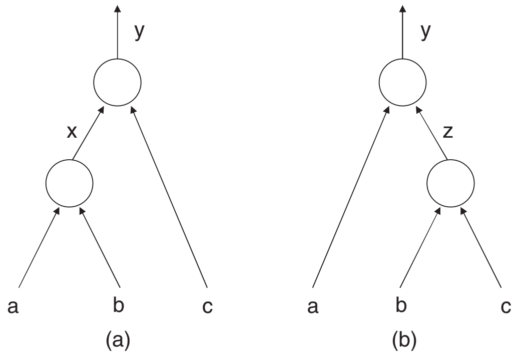


FIGURE 6.40

Collapsing and redecomposition.

Once the redecomposition points are chosen, they are sped up by the collapsing-decomposing procedure as described in Section 6.3.3. Since in a multilevel network we can reduce the area by sharing common functions, we first attempt to extract area saving divisors that do not contain critical signals. After all such divisors have been extracted, we decompose the node into a tree and place late arriving signals closer to the outputs, thus making them pass through a smaller number of gates.

Example 6.83 In Figure 6.41, the critical paths in the original network are shown in bold and begin from signals c and d. Node f is collapsed, and a divisor k is selected which has the desired property that substituting k into f, places the critical signals c and d closer to the output.

Note that the critical paths in the decomposed network may have changed. The collapsing-decomposing procedure can be iterated by identifying a new

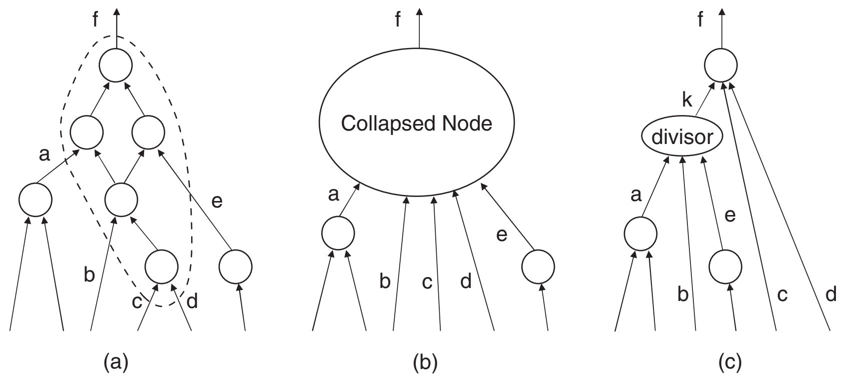


FIGURE 6.41

Basic idea of timing decomposition.

critical section. The algorithm proceeds until the requirement is satisfied or no improvement in delay can be made. A detailed exposition of speed optimization algorithms can be found in, *e.g.*, [Singh 1992; Devadas 1994].

6.6.2 Timing-driven technology mapping

Technology-independent delay optimization algorithms cannot estimate the delay of a circuit accurately, largely due to the lack of accurate technology-independent delay models. Therefore, such algorithms are not guaranteed to produce faster circuits, when circuit speed is measured after technology mapping and physical design. We will present a more accurate approach to delay optimization during technology mapping. The tree covering algorithm presented in Section 6.4.5, in the context of technology mapping for minimum area, will be modified to target circuit speed.

The most accurate estimation of the delay of a gate in a circuit can only be obtained after the entire circuit has been placed and routed. Since technology mapping has to be performed before placement and routing, an approximate delay model with reasonable accuracy has to be used. We adopt the linear delay model of Equation (6.19) of Section 6.5 in the following discussion.

6.6.2.1 *Delay optimization using tree covering*

The tree covering algorithm of Section 6.4.5 can only be used if the cost of a match at a gate can be determined by examining the cost of the match and the cost of the inputs to the match (for which the cost has already been determined). For area optimization the cost of a gate depends on the area cost of the match and the area cost of the inputs of the match. For delay optimization, the cost is signal arrival time at the output of the match. Therefore, the cost of a match for delay optimization depends not only on the structure of the tree beneath the gate, but also on the capacitive load seen by the match. This load cannot be determined at the time of the selection of the match as it depends on the unmapped portion of the tree. Several attempts have been made to generalize tree covering to produce minimum delay implementations [Rudell 1989; Touati 1990; Chaudhary 1992].

Load-Independent Tree Covering. The tree covering algorithm of Section 6.4.5 can be used to produce a minimum delay implementation of a circuit provided the loads of all the gates in the circuit are the same. Under the assumption that the delay of a gate is independent of the fanout of the gate, the tree covering algorithm provides the minimum arrival time cover, if we compute and store the arrival time at each node and choose the minimum arrival time match at each node.

Example 6.84 Consider the technology library shown in Figure 6.42 and the circuit shown in Figure 6.43a. For each gate in the library, its name, area, symbol, and pattern DAG are presented. In addition, the delay parameters for our delay model are shown. By Equation (6.19), the

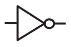
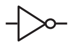
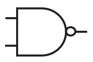
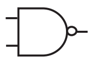
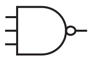
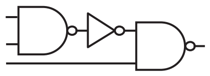
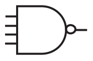
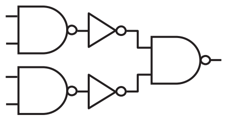
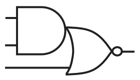
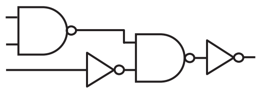
Gate	Area	Symbol	Pattern DAG	Delay Parameters
INV	1			A=0,B=1,G=1
NAND2	2			A=1,B=1,G=2
NAND3	3			A=1,B=2,G=3
NAND4	4			A=5,B=2,G=5
AOI21	3			A=1.5,B=1,G=3

FIGURE 6.42

Gate library.

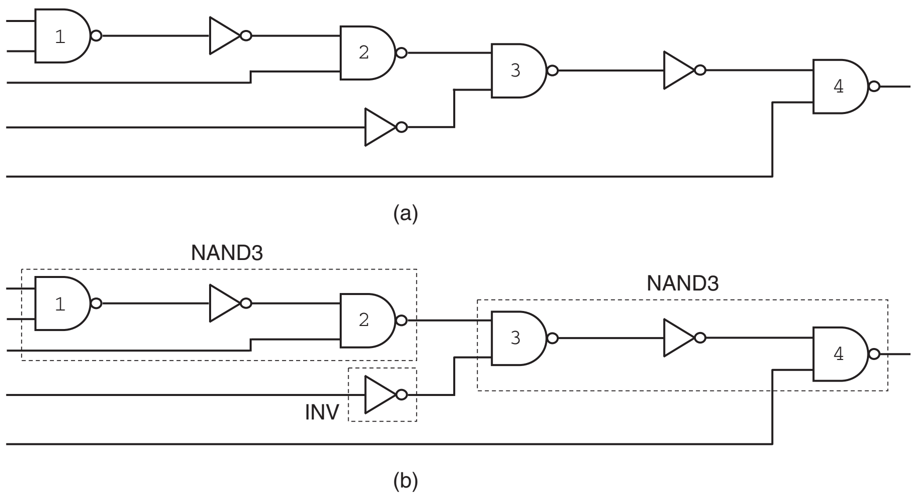


FIGURE 6.43

Circuit and its mapped implementation.

intrinsic delay, T_p, is denoted by A, the load dependent coefficient T_e/C_{in} is denoted by B, and the load C_{in} presented by the gate to any input gate is denoted by G. Note that in order to calculate the delay of a gate using Equation (6.19), we will use A and B for the gate and sum up the G values for all its fanout gates.

If the load of each gate in the circuit is considered to be 1, then the perfect match at each gate can be determined in one bottom-up pass, as in Section 6.4.5. For gate 1, this corresponds to a 2-input NAND gate with a delay of 2. The best match at gate 2 is a 3-input NAND gate with a delay of 3. The best covering for this circuit under the fixed load assumption is shown in Figure 6.43b.

Load-Dependent Tree Covering. The above load-independent tree covering does not necessarily produce the optimal solution because the load of all gates is not the same. As can be seen from the library in Figure 6.42, different gates provide different load values to their inputs.

An algorithm, originally presented in [Rudell 1989], can be used to take into account the effect of different loads. The first step of the algorithm is a pre-processing step over the technology library in order to create n load bins and quantize the load values for all the pins in the library. For each load bin, a representative load value is selected, and the remaining load values are mapped to their closest value in the chosen set. The value of n determines the accuracy and the run time of the algorithm. If n is equal to the number of distinct loads in the library, then the algorithm is most accurate. However, the larger the value of n, the more computation will be required. Instead of quantizing load values *a priori* based on the library information, a better way is to adapt the quantization intervals to each gate. In one pre-computation phase, we can determine all possible load values at a gate by examining all the possible matches at the gate. These load values can then be used to determine the values of the quantization intervals.

For a match at a gate, an array of costs (one for each load value) is calculated. The cost is the arrival time of the signal at the output of the gate. For each bin or load value, the match that gives the minimum arrival time is stored. For each input i of the match, the optimum match for driving the pin load of pin i of the match is assumed, and the arrival time for that match is used. This calculation can be done by traversing the tree once forward from the leaves of the tree to its root. The tree is then traversed backward from the root to the leaves, whereby the load values are propagated down and, for each gate, the best match at the gate is selected depending on the value of the load seen at the gate.

Example 6.85 We illustrate the algorithm using the circuit of Figure 6.43a and the library of Figure 6.42. Consider the best matches shown in Figure 6.44. Since the number of distinct load values in our example is only four, four bins are considered. For gate 1 the only match is a NAND2 gate. For each load value, the delay of this gate then gives the arrival time at the output of the match (assuming zero arrival time at the inputs). For the inverter at the output of this NAND gate, the only match is that of an inverter. Since the inverter presents a load of 1 to the NAND gate, the arrival time at the input of the inverter is the arrival time corresponding to the first bin of the NAND gate. Using this arrival time, the arrival times at the output of the inverter for all possible load values are computed and are shown in the figure.

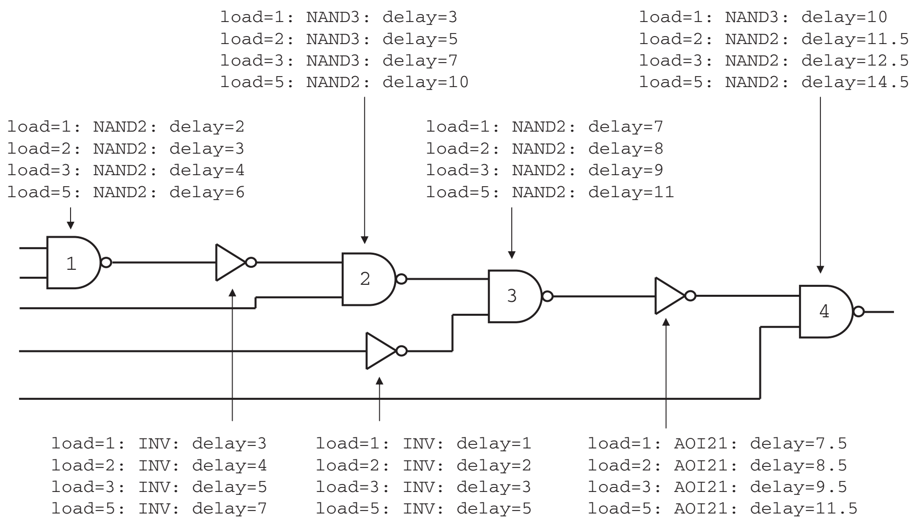


FIGURE 6.44

Technology mapping considering load values.

At gate 2, there are two possible matches corresponding to 2-input and 3-input NAND gates. If we consider the NAND2 gate, the two arrival times at the inputs of the match are 0 (corresponding to the primary input connection to gate 2) and 4 (corresponding to the inverter connection to gate 2 seeing a load of 2). The maximum arrival time at the inputs is 4. The arrival times at the output of the gate for the four different load values are 6, 7, 8, and 10. *E.g.*, for a load value of 5, a NAND2 gate has a delay $1 + 1 \times 5 = 6$. This delay added to the arrival time of 4 at the input of the NAND gate produces an arrival time of 10 at the output. For the NAND3 gate, the arrival times of all inputs are 0, and therefore the arrival times at the output are 3, 5, 7, and 11. Therefore, for the first three load values, the NAND3 is a better choice, while for the last load value the NAND2 is a better choice.

The final mapping is determined during backward traversal and depends on the load seen by gate 4. Assuming a load of 1, the best match at gate 4 is a NAND3 gate. This gate presents a load of 3 to its inputs, implying that the best match for a load value of 3 at gate 2 has to be chosen. This match is another NAND3 gate. The resulting mapping is shown in Figure 6.45a, which is coincidentally the same mapping obtained assuming constant load (Figure 6.43b). However, if the load is greater than 1, then the mapping of Figure 6.45b is better.

To improve the computation, we may apply adaptive quantization of load values. For instance, for gate 1 in the circuit of Figure 6.44, only a load value of 1 has to be considered because all possible matches at the inverter consist of only an inverter; for gate 2, load values of 2 and 3 have to be considered. This type of adaptive quantization produces results close to the optimum within reasonable amounts of computation time.

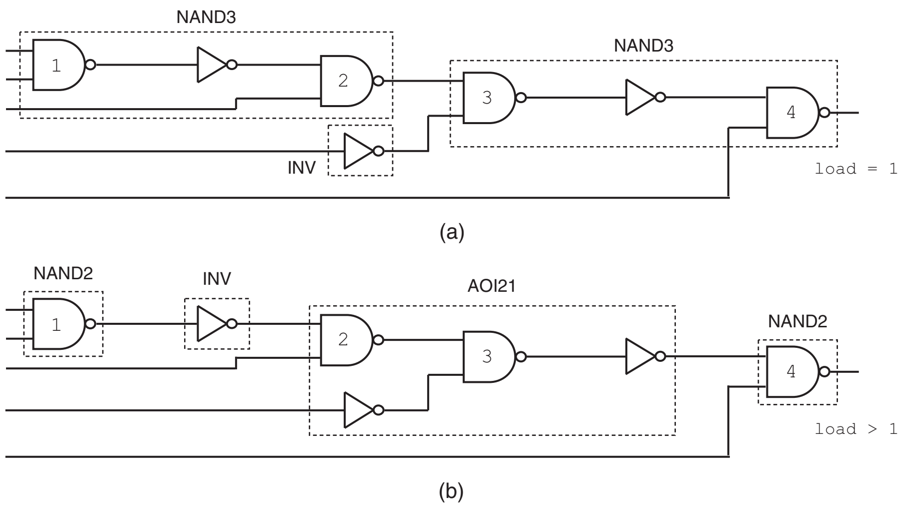


FIGURE 6.45

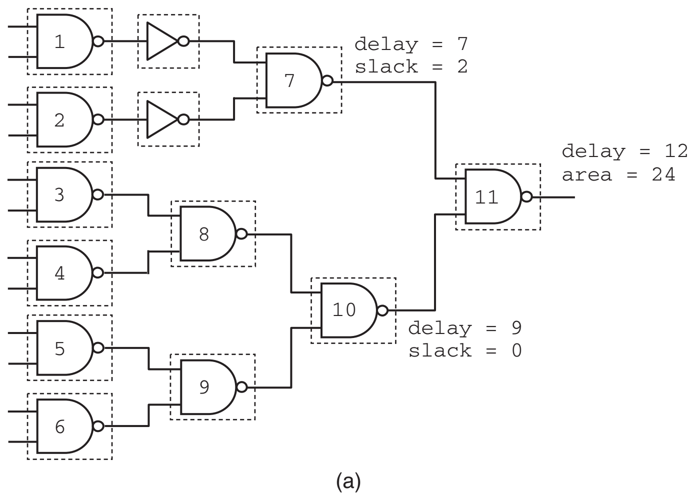
Two different implementations of the circuit depending on load value.

Note that, under the more general linear delay model, the principle of optimality of tree covering does not apply.

6.6.2.2 *Area minimization under delay constraints*

The tree covering algorithm used above can be generalized to minimize the area under a delay constraint. It may not be necessary to obtain the fastest circuit, but instead we may want to obtain a circuit that meets certain timing constraints and has the minimum possible area. This timing constraint is expressed as a required time at the root of the tree and can be propagated down the tree together with load values during backward traversal. In this case the cost of a match at a gate includes not only the arrival time but also the area of a match. During backward traversal the minimum area solution that meets the required timing constraint is chosen. If no such solution is available, then the minimum delay solution is chosen. Since not all of the sub-trees need to be maximally fast, the area of the circuit can be minimized.

Example 6.86 Consider the mapping shown in Figure 6.46a. The circuit has been mapped for minimum delay, and the arrival time at the output of gate 7 is 7. However, the required time at the output of this gate is 9, and the other match at gate 7 has an arrival time of 9 but a smaller area. Selecting this match gives us a circuit with the same delay but a smaller area, as shown in Figure 6.46b.



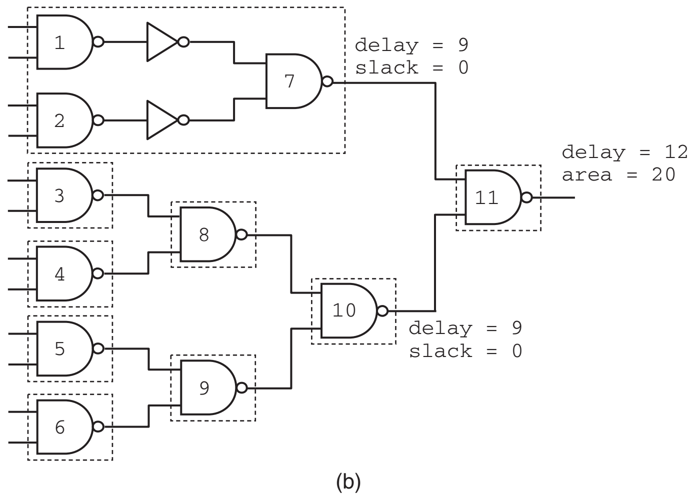


FIGURE 6.46

Example illustrating area recovery.

6.6.3 **Advanced subjects**

Fanout Optimization. Tree covering alone does not generate good quality solutions because most circuits are not trees but DAGs. In such circuits, a signal may feed two or more destinations. Due to the large amount of capacitance that has to be driven, the delay through the gate that drives this signal could be large. The optimization of this delay is called fanout optimization. Buffer insertion and gate sizing, among other techniques, are important approaches to fanout optimization. A survey on fanout optimization can be found in [Hassoun 2002].

Sequential Circuit Timing Optimization. In addition to logic restructuring, we may exploit optimization techniques special for sequential circuits. Promising sequential timing optimization methods include, for instance, retiming [Leiserson 1983, 1991] and clock skew scheduling. See, *e.g.*, [Sapatnekar 2004] for introduction.

6.7 CONCLUDING REMARKS

This chapter presents some important classic problems in combinational logic synthesis and basic techniques to solve them. Since logic synthesis has become very broad and continues to evolve, many important developments cannot be covered and only a few of them are mentioned here.

To invite and motivate future investigations, we list some logic synthesis trends:

Scalable Logic Synthesis. The capacity of logic synthesis tools is constantly being challenged by the ever-increasing complexity of modern industrial designs commonly consisting of millions of gates. The data structures and algorithms of logic synthesis tools must be effective and robust enough in order to handle large problem instances. It is interesting to note that every capacity leap in the history of logic synthesis can be attributed to some data structure revolution, *e.g.*, from truth tables to covers, from covers to BDDs, and from BDDs to AIGs and SAT. As SAT solvers have become much faster in recent years, a paradigm shift is taking place in logic synthesis. More and more SAT-based algorithms emerge in replacement of BDD-based ones. Searching for new effective data structures may transform logic synthesis tools.

Verifiable Logic Synthesis. As noted earlier, due to the hardness of verification, industrial synthesis methodologies are often conservative and mostly conduct only combinational optimization, despite the existence of practical sequential synthesis techniques.[4] This phenomenon is changing because progressive optimization methods are necessary to meet more stringent timing constraints, and also verification techniques are made more effective, especially for circuits optimized in particular ways [Jiang 2007]. To completely overcome the verification barrier, a general consensus is that essential synthesis information should be revealed to verifiers. Verifiable logic synthesis sets forth the criterion that whatever can be synthesized can be verified effectively [Brayton 2007].

Parallelizable Logic Synthesis. One way to speed up logic synthesis algorithms is to take advantage of hardware and software technologies. As multicore computers support more and more parallelism, EDA tools can benefit from this technology advancement. How to utilize parallelism in logic synthesis algorithms is a challenge for EDA companies.

[4]One exception is FPGA synthesis, where sequential optimization methods find wide applications. The reconfigurability of FPGAs makes verification not as critical as general ASIC designs because incorrect logic transformations can be rectified later through reconfiguration.

Statistical Logic Synthesis. The continuous miniaturization of semiconductor devices imposes serious threats to circuit design robust against process variations and environmental fluctuations. Various uncertainties appear in both pre- and post-design phases. How to synthesize a robust circuit optimal in a statistical sense with respect to design constraints is an important challenge that needs to be addressed.

Physically Aware Logic Synthesis. Logic synthesis and physical design are traditionally separated to enable a divide-and-conquer approach to VLSI design automation. This separation becomes problematic when interconnect becomes the dominating factor of circuit delays. Lacking wiring information, logic synthesis cannot produce accurate timing estimation and precise timing optimization; lacking logic information, physical design cannot exploit logic flexibility and has limited optimization power. Therefore, before timing constraints are met, often several iterations of logic synthesis and physical design are performed in order to reach timing closure. Unfortunately there is no guarantee that the process will converge. This phenomenon leads to a serious design closure problem, which slows down design cycles and therefore time to market. Even though there are approaches to timing closure, such as gain-based synthesis, incremental placement and resynthesis, etc., there is still plenty of room for improvement.

Logic Synthesis for Emerging Technologies. As the miniaturization of electronic devices approaches physical limits, Moore's Law is expected to be broken sooner or later. Alternatives to silicon-based computation devices are actively being researched. For the next computation model, we might need very different logic synthesis tools, perhaps even beyond propositional logic and Boolean algebra.

6.8 EXERCISES

6.1. (Commutativity between Cofactor and Boolean Operations) Given two Boolean functions f and g and a Boolean variable v, prove or disprove the following equalities:

(a) $(\neg f)_v = \neg(f_v)$

(b) $(f \langle op \rangle g)_v = (f_v) \langle op \rangle (g_v)$ for $\langle op \rangle = \{\wedge, \oplus\}$

6.2. (Boolean Difference) Let $f(x, y, z) = h(g(x, y, z), y, z)$. Prove or disprove the following equalities:

(a) $$\frac{\partial^2 f(x,y,z)}{\partial x \partial y} = \frac{\partial^2 f(x,y,z)}{\partial y \partial x}$$

(b) $$\frac{\partial f(x,y,z)}{\partial x} = \frac{\partial h(u,y,z)}{\partial u} \frac{\partial g(x,y,z)}{\partial x}$$

(c) $\dfrac{\partial f(x,y,z)}{\partial y} = \dfrac{\partial h(u,y,z)}{\partial u}\dfrac{\partial g(x,y,z)}{\partial y} \oplus \dfrac{\partial^2 h(u,y,z)}{\partial u \partial y}\dfrac{\partial g(x,y,z)}{\partial y}\dfrac{\partial y}{\partial y}$

6.3. (Quantified Boolean Formula) For Boolean functions f and g, show that

(a) $\neg(\exists x. f(x,y)) = \forall x. \neg f(x,y)$

(b) $\neg(\forall x. f(x,y)) = \exists x. \neg f(x,y)$

(c) $\exists x. (f(x,y) \wedge g(x,y)) \neq (\exists x. f(x,y)) \wedge (\exists x. g(x,y))$

(d) $\neg \forall x, \exists z. (f(x,y) \wedge g(x,z)) = \exists x. (\neg f(x,y) \vee \forall z. \neg g(x,z))$

6.4. (Boolean Function Bi-decomposition) For a given Boolean function $f(X_A, X_B)$ with non-empty variable sets X_A and X_B, with $X_A \cap X_B = \emptyset$, what is the condition on $f(X_A, X_B)$ such that the rewriting $f(X_A, X_B) = f_A(X_A) \wedge f_B(X_B)$ is possible for some $f_A(X_A)$ and $f_B(X_B)$ to exist? (Express the condition with a quantified Boolean formula.)

6.5. (Characteristic Functions) Let $\boldsymbol{f} : \mathbb{B}^3 \rightarrow \mathbb{B}^2$ be the vector (f_1, f_2) of Boolean functions with $f_1 = x_1 \vee \neg x_1 x_2$ and $f_2 = x_3 \wedge (x_1 \vee \neg x_1 x_2)$; let $\chi_S = x_1 \vee x_2$ be a characteristic function representing a set $S \subseteq \mathbb{B}^3$.

(a) Write down the characteristic function $Img_{\boldsymbol{f}}(S)$ (in terms of a quantified Boolean formula) of the *image* of S under the mapping of $\boldsymbol{f}$, that is, the set $\{q \in \mathbb{B}^2 \mid q = \boldsymbol{f}(p), p \in S\}$.

(b) Perform quantifier elimination to obtain a quantifier-free formula equivalent to $Img_{\boldsymbol{f}}(S)$ in (a).

(c) Justify that the formula in (b) indeed represents the image of S under $\boldsymbol{f}$ by enumerating all the truth assignments of (x_1, x_2, x_3) and the corresponding valuations of χ_S and $\boldsymbol{f}$.

6.6. (BDD APPLY) Let F and G be the ROBDDs of Boolean functions $f = abc$ and $g = bd + b'd$, respectively, under the variable ordering $index(a) < index(b) < index(c) < index(d)$.

(a) Draw F and G.

(b) Derive the ROBDD of $F \cdot G$ using the BddApply procedure.

(c) Derive the ROBDD of $F + G$ using the BddApply procedure.

(d) Derive the ROBDD of $F \oplus G$ using the BddApply procedure.

6.7. (ROBDD Variable Ordering) Let F be the ROBDD of an arbitrary Boolean function $f(a, b, c, d, e)$ under variable ordering $index(a) < index(b) < index(c) < index(d) < index(e)$.

Show that the new ROBDD $F^\dagger$ under variable ordering
$index(a) < index(b) < index(d) < index(c) < index(e)$,
must have the same BDD structure as F except for the nodes controlled by variables c and d.

6.8. (ROBDD Variable Ordering) Consider the Boolean function
$f = a_1b_1 + a_2b_2 + \cdots + a_nb_n$.

(a) Show that the ROBDD under variable ordering
$index(a_1) < index(b_1) < \cdots < index(a_n) < index(b_n)$
has $2n + 2$ nodes.

(b) Show that the ROBDD under variable ordering
$index(a_1) < \cdots < index(a_n) < index(b_1) < \cdots < index(b_n)$
has 2^{n+1} nodes.

6.9. (ROBDDs of Symmetric Functions) Totally symmetric functions are characterized by the fact that the value of each such function is determined by the number of variables which are 1 under a truth assignment; it does not matter which particular variables are. For example, functions $f_1 = x_1 \wedge \cdots \wedge x_n$, $f_2 = x_1 \vee \cdots \vee x_n$, and $f_3 = x_1 \oplus \cdots \oplus x_n$ are totally symmetric. A totally symmetric function on n variables can be described by a set $S \subseteq \{0, 1, \ldots, n\}$ such that for a minterm $a \in \mathbb{B}^n$, $f(a) = 1$ iff the number of 1's in a is a member of S. Prove that the ROBDD of any n-ary totally symmetric function has at most $O(n^2)$ nodes under any variable ordering.

6.10. (Circuit-to-CNF Conversion) Convert each of the following circuits to a CNF formula representing the consistency condition. In each case, list the truth assignments to the input/output variables that make the CNF true.

(a) An inverter with input a and output b.
(b) An OR2 gate with inputs a, b and output c.
(c) An XOR gate with inputs a, b and output c.

6.11. (Global Function Derivation) Consider the AIG of Figure 6.16. Derive the global function of x_7 (in terms of primary inputs x_1, x_2, x_3) using the following two methods.

(a) Existentially quantify out the intermediate variables x_4, x_5, x_6 from its corresponding consistency CNF formula and then perform a positive cofactor with respect to the variable x_7.
(b) Derive the global function of x_7 by recursively substituting intermediate variables with their local functions.

Verify that the above two methods yield the same result. Explain why these two approaches are equivalent.

6.12. (SOP and Tautology) Show that the tautology checking of any SOP formula with at most 2 literals in each product term can be done with time complexity polynomial in the formula size.

(Remark: The dual problem is the 2SAT problem in computer science, which is checkable in polynomial time.)

6.13. (Prime and Irredundant Cubes) Let

$$C = \{a'c'd',\ abd',\ a'b'd',\ a'bc',\ ab'c',\ a'b'c,\ abc,\ a'bd\}$$

be a cover of a completely specified function f.

(a) For each cube in C, determine whether it is prime and/or irredundant.

(b) Can we delete all the redundant cubes at once without affecting the function of f? Which redundant cubes can we delete from C if we successively delete removable cubes from left to right? How about from right to left? (Assume the cubes listed in C is ordered.)

6.14. (Quine-McCluskey Two-Level Logic Minimization) Given function

$$f = w'x'y'z' + wx'z' + wxz + w'x'z$$

with don't care set

$$d = w'xyz' + wx'yz + w'xyz$$

minimize f using the Quine-McCluskey procedure.

6.15. (Column Covering) Column covering is an essential computation step in Quine-McCluskey procedure. It can be solved in different ways.

(a) Show that the column covering problem can be formulated as a CNF satisfiability problem. Give an algorithm that performs such conversion. (The so-derived covering need not be minimum.)

(b) Show that the *minimum* column covering problem can be formulated in term of ROBDD. Give a polynomial-time algorithm solving the problem.

6.16. (Number of Prime Implicants) Show that

$$\frac{C^{n}_{\lceil \frac{n-2}{3} \rceil} 2^{n-\lceil \frac{n-2}{3} \rceil}}{n - \lceil \frac{n-2}{3} \rceil}$$

is a lower bound on the number of prime implicants for any n-ary Boolean function.

6.17. (Node Value and Elimination) Recall that the value of a node represents the saved literal count due to the existence of the node rather than collapsing it into its fanouts. Given the Boolean network of Figure 6.47, what are the values of nodes 5 and 6? What is the new value of node 6 after collapsing node 5 into its fanouts? (Here we treat Boolean formulas as polynomials in an algebraic sense, and assume that Boolean simplifications, such as $x \wedge \neg x = 0$, $x \vee \neg x = 1$, $x \wedge \mathrm{x} = x$, and $x \vee x = x$, are not involved.)

6.18. (Algebraic Division) Prove that algebraic division produces a unique quotient and remainder. (Note that by definition the remainder is made as few cubes as possible.)

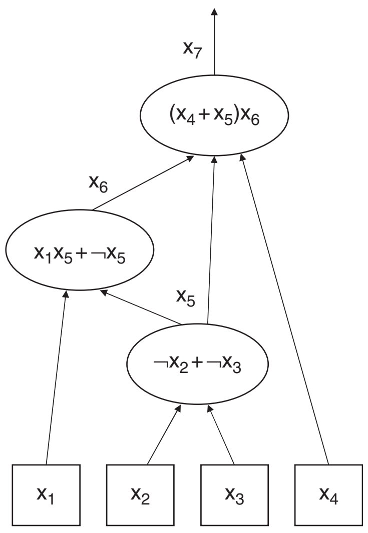


FIGURE 6.47

Boolean network.

6.19. (Kernels and Cokernels) Let expression
$F = aefh + aegh + aei + befh + begh + bei + cdefh + cdegh + cdei.$
Apply KERNEL1(0, F) to compute the kernels and corresponding cokernels of F. Identify which kernels are of level 0.

6.20. (Factoring) Continuing Exercise 6.19, apply GFACTOR to factor the function F. Use different level-0 kernels as divisors. What is the best factoring for F? For an arbitrary expression, can GFACTOR always produce a minimum-literal factoring with some proper level-0 kernels as divisors?

6.21. (Common Divisor Extraction) Let expressions
$F = ac + ad + bc + bd + adf + aef + ag + bcdf + bcef + bcg$, and
$G = ag + bcg + bcf + bcg + bdf + bdg + bef + beg$

(a) Iteratively reexpress F and G in terms of a common expression that yields the most reduction in literal count until no more common expressions exist. (A common expression can be a cube-free expression or a cube.)

(b) Extract an optimal common divisor of F and G by finding rectangles in the cokernel-cube matrix.

6.22. (Kernel Intersection) For two expressions F and G, suppose any kernel k_f of F and any k_g of G have at most one term in common. Show that F and G have no common algebraic divisor with more than one term.

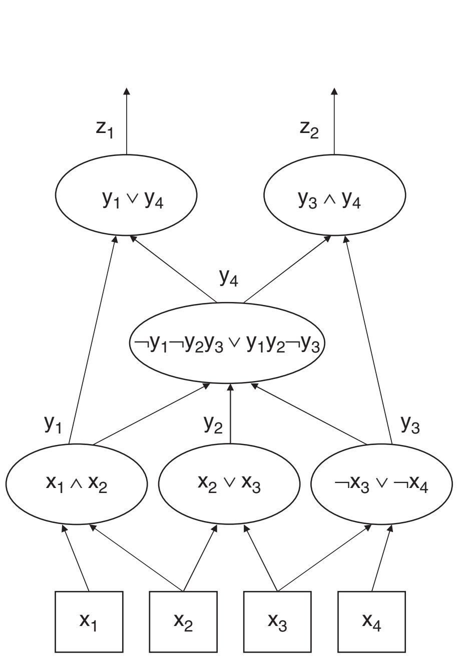


FIGURE 6.48

Boolean network.

6.23. (SDC and ODC) Consider the Boolean network of Figure 6.48.

(a) Write down a Boolean formula representing the SDC of the entire circuit. That is, it represents the inconsistency condition of the circuit.

(b) Write down a Boolean formula for the satisfiability don't cares SDC_4 of node 4 (with output y_4). Since SDC_4 is induced by the transitive fanins of node 4, the formula should depend on variables $x_1, \dots, x_4, y_1, \dots, y_3$. How can you make SDC_4 refer only to y_1, y_2, y_3 such that we can minimize node 4 directly?

(c) Compute the observability don't cares ODC_4 of node 4.

6.24. (Don't Cares in Local Variables) Continuing Exercise 6.23, suppose the XDC for z_1 is $\neg x_1 \neg x_2 \neg x_3 \neg x_4$ and that for z_2 is $x_1 x_2 x_3 x_4$.

(a) Compute the don't cares D_4 of node 4 in terms of its local input variables y_1, y_2, and y_3. (Note that in general the computation of ODC may be affected by XDC especially when there exist different XDCs for different primary outputs.)

(b) Based on the computed don't cares, what is the best implementable function for node 4 (in terms of the literal count and cube count)?

6.25. **(Complete Flexibility)** Continuing Exercise 6.24, let $Y = \{y_1, y_2, y_3\}$ and $Z = \{z_1, z_2\}$.

(a) Suppose the XDC for z_1 is $\neg x_1 \neg x_2 \neg x_3 \neg x_4$ and that for z_2 is $x_1 x_2 x_3 x_4$. Write down the specification relation $S(X,Z)$.
(b) What is the influence relation $I_4(X, y_4, Z)$ of node 4?
(c) What is the environment relation $E_4(X, Y)$ of node 4?
(d) What is the complete flexibility $CF_4(Y, y_4)$ of node 4?
(e) Is the previously computed don't care set D_4 of node 4 subsumed by CF_4?

6.26. **(Static Timing Analysis)** Given the circuit of Figure 6.49 with gate delays shown, assume the arrival times for the primary inputs are 0 except for input *b* with arrival time 1ns, and the required times for the primary output are 8ns. Compute the ***arrival time, required time***, and ***slack*** of every net. Identify the critical path(s).

6.27. **(Time Slack and Critical Path)** Prove or disprove the following statement: The most critical path (with the smallest slack) must be a thorough path all the way from some primary input to some primary output.

6.28. **(Arrival/Required Time Computation)** Given a black box that computes arrival times for a Boolean network with specified gate delays and input arrival times, devise a way of reusing this black box to compute required times for a Boolean network.

6.29. **(Tree Mapping)** Decompose the subject DAG of Figure 6.50 into trees and perform dynamic programming to find optimum tree

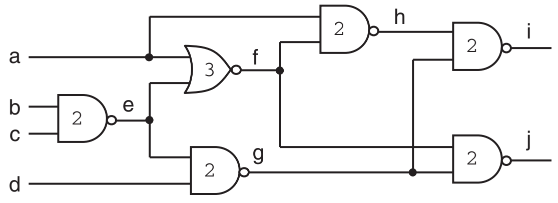


FIGURE 6.49

Circuit for timing analysis.

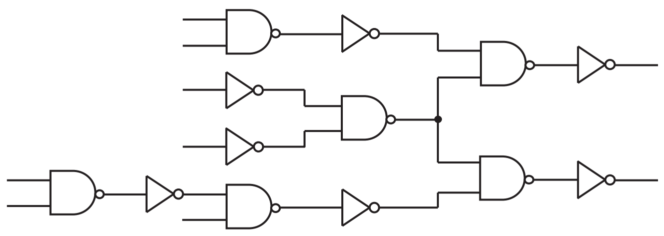


FIGURE 6.50

Subject graph.

mappings with respect to the pattern graphs of Figure 6.26. What is the optimum solution that you can get among different decomposition approaches?

6.30. **(DAG Mapping as SAT Solving)** Formulate the DAG mapping *feasibility* problem as a satisfiability problem. For the subject graph of Figure 6.50 and the pattern graphs of Figure 6.26, what is the CNF formula representing feasible DAG mappings?

ACKNOWLEDGMENTS

The authors are grateful to Professor Robert Brayton and Dr. Alan Mishchenko of the University of California at Berkeley, and Professor Jianwen Zhu of the University of Toronto for valuable feedback on the manuscript.

REFERENCES

R6.0 Books

[Brayton 1984] R. K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer, 1984.

[Brown 2003] F. M. Brown, *Boolean Reasoning: The Logic of Boolean Equations,* Dover, 2003.

[Devadas 1994] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis,* McGraw-Hill, 1994.

[Garey 1979] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, 1979.

[Hassoun 2002] S. Hassoun and T. Sasao, *Logic Synthesis and Verification,* Kluwer, 2002.

[Kohavi 1978] Z. Kohavi, *Switching and Finite Automata Theory,* McGraw-Hill, 1978.

[McGeer 1991] P. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Design,* Kluwer, 1991.

[Mo 2004] F. Mo and R. K. Brayton, *Regular Fabrics in Deep Sub-Micron Integrated-Circuit Design,* Kluwer, 2004.

[Minato 1996] S. Minato, *Binary Decision Diagrams and Applications to VLSI CAD,* Kluwer, 1996.

[Papadimitriou 1993] C. Papadimitriou, *Computational Complexity,* Addison Wesley, 1993.

[Sapatnekar 2004] S. Sapatnekar, *Timing,* Springer, 2004.

[Scholl 2001] C. Scholl, *Functional Decomposition with Applications to FPGA Synthesis,* Kluwer, 2001.

[Sutherland 1999] I. Sutherland, R. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits,* Margan Kaufmann, 1999.

[Villa 1997] T. Villa, T. Kam, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Logic Optimization,* Kluwer, 1997.

R6.1 Introduction

[ABC 2005] Berkeley Logic Synthesis and Verification Group, ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/~alanmi/abc/, 2005.

[Brayton 1987] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, MIS: Multiple-level interactive logic optimization system, *IEEE Trans. on Computer-Aided Design*, 6(6), pp. 1062–1081, November 1987.

[Gao 2002] M. Gao, J.-H. R. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, T. Villa, and R. K. Brayton, Optimization of multi-valued multilevel networks, in *Proc. IEEE Int. Symp. on Multiple-Valued Logic*, pp. 168–177, May 2002.

[Rudell 1987] R. Rudell and A. Sangiovanni-Vincentelli, Multiple-valued minimization for PLA optimization, *IEEE Trans. on Computer-Aided Design*, 6(5), pp. 727–751, September 1987.

[Sentovich 1992] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, SIS: A system for sequential circuit synthesis, *Memo. UCB/ERL M92/41*, 1992.

R6.2 Data Structures for Boolean Representation and Reasoning

[ABC 2005] Berkeley Logic Synthesis and Verification Group, ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/∼alanmi/abc/, 2005.

[Akers 1978] S. B. Akers, Binary decision diagrams, *IEEE Trans. on Computers*, C-27(6), pp. 509–516, June 1978.

[Biere 2007] A. Biere, The AIGER and-inverter graph (AIG) format, http://fmv.jku.at/aiger/, 2007.

[Bryant 1986] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Trans. on Computers*, C-35(8), pp. 677–691, August 1986.

[Bryant 1991] R. E. Bryant, On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication, *IEEE Trans. on Computers*, C-40(2), pp. 205–213, February 1991.

[Bryant 1992] R. E. Bryant, Symbolic Boolean manipulation with ordered binary decision diagrams, *ACM Computg Surveys*, 24(3), pp. 293–318, September 1992.

[Kautz 1970] W. Kautz, The necessity of closed circuit loops in minimal combinational circuits, *IEEE Trans. on Computers*, C-19(2), pp. 162–164, February 1970.

[Kuehlmann 1997] A. Kuehlmann and F. Krohm, Equivalence checking using cuts and heaps, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 263–268, June 1997.

[Lee 1959] C. Y. Lee, Representation of switching circuits by binary-decision programs, *Bell Systems Techncal J.*, 38(4), pp. 985–999, July 1959.

[Madre 1988] J-C. Madre and J-P. Billon, Proving circuit correctness using formal comparison between expected and extracted behaviour, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 205–210, June 1988.

[Rudell 1990] R. L. Rudell, K. S. Brace, and R. E. Bryant, Efficient implementation of a BDD package, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 40–45, June 1990.

[Rudell 1993] R. Rudell, Dynamic variable ordering for binary decision diagrams, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 42–47, November 1993.

[Rudell 1987] R. Rudell and A. Sangiovanni-Vincentelli, Multiple-valued minimization for PLA optimization, *IEEE Trans. on Computer-Aided Design*, 6(5), pp. 727–751, September 1987.

[Tseitin 1970] G. S. Tseitin, On the complexity of derivation in propositional calculus, *Studies in Constructive Mathematics and Mathematical Logic*, Part II, (A. O. Slisenko, editors), pp. 115–125. Consultants Bureau, New York, 1970.

R6.3 Combinational Logic Minimization

[ABC 2005] Berkeley Logic Synthesis and Verification Group, ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/∼alanmi/abc/, 2005.

[Bartlett 1986] K. Bartlett, W. Cohen, A. J. De Geus, and G. D. Hachtel, Synthesis of multilevel logic under timing constraints, *IEEE Trans. on Computer-Aided Design*, CAD-5(4), pp. 582–595, October 1986.

[Bjesse 2004] P. Bjesse and A. Boralv, DAG-aware circuit compression for formal verification, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 42–49, November 2004.

[Bostick 1987] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, The Boulder optimal logic design system, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 62–65, November 1987.

[Brayton 1982] R. K. Brayton and C. McMullen, The decomposition and factorization of Boolean expressions, in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 49–54, May 1982.

[Brayton 1984] R. K. Brayton and C. McMullen, Synthesis and optimization of multistage logic, in *Proc. IEEE Int. Conf. on Computer Design*, pp. 23–28, October 1984.

[Brayton 1987] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, MIS: Multiple-level interactive logic optimization system, *IEEE Trans. on Computer-Aided Design*, 6(6), pp. 1062–1081, November 1987.

[Brayton 1990] R. K. Brayton, G. D. Hachtel, and A. Sangiovanni-Vincentelli, Multilevel logic synthesis,*Proceedgs of the IEEE*, 78(2), pp. 264–300, February 1990.

[Coudert 1994] O. Coudert, Two-level logic minimization: An overview, *Integrato*,17(2), pp. 97–140, October 1994.

[Coudert 1995] O. Coudert, Doing two-level logic minimization 100 times faster, in *Proc. ACM/SIAM Symp. on Discrete Algorithms*, pp. 112–121, January 1995.

[Dagenais 1986] M. Dagenais, V. K. Agarwal, and N. Rumin, McBOOLE: A procedure for exact Boolean minimization, *IEEE Trans. on Computer-Aided Design*, CAD-5(1), pp. 229–237, January 1986.

[Darringer 1981] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan, Logic synthesis through local transformations, *IBM J. of Research and Development*, 25(4), pp. 272–280, July 1981.

[Darringer 1984] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, LSS: A system for production logic synthesis, *IBM J. of Research and Development*, 28(5), pp. 537–545, September 1984.

[Devadas 1989] S. Devadas, A. R. Wang, A. R. Newton, and A. Sangiovanni-Vincentelli, Boolean decomposition in multilevel logic optimization, *IEEE J. of Solid State Circuits*, 24(2), pp. 399–408, April 1989.

[Fleisher 1975] H. Fleisher and L. I. Maissel, An introduction to array logic, *IBM J. of Research and Development*, 19(3), pp. 98–109, March 1975.

[Hong 1974] S. J. Hong, R. G. Cain, and D. L. Ostapko, MINI: A heuristic approach for logic minimization, *IBM J. of Research and Development*, 18(4), pp. 443–458, September 1974.

[Jiang 2004] J.-H. R. Jiang and R. K. Brayton, Functional dependency for verification reduction, in *Proc. Int. Conf. on Computer Aided Verification*, pp. 268–280, July 2004.

[Jiang 2006] J.-H. R. Jiang and R. K. Brayton, Retiming and resynthesis: A complexity perspective, *IEEE Trans. on Computer-Aided Design*, 25(12), pp. 2674–2686, December 2006.

[Keutzer 1987] K. Keutzer, DAGON: Technology mapping and local optimization, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 341–347, June 1987.

[Kuehlmann 1997] A. Kuehlmann and F. Krohm, Equivalence checking using cuts and heaps, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 263–268, June 1997.

[Lee 2007] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko, Scalable exploration of functional dependency by interpolation and incremental SAT solving, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 227–233, November 2007.

[Leiserson 1983] C. Leiserson and J. Saxe, Optimizing synchronous systems, *J. of VLSI and Computer Systems*, 1(1), pp. 41–67, Spring 1983.

[Leiserson 1991] C. Leiserson and J. Saxe, Retiming synchronous circuitry, *Algorithmica*, 6(1), pp. 5–35, December 1991.

[Ling 2007] A. Ling, J. Zhu, and S. Brown, BddCut: Towards scalable symbolic cut enumeration, in *Proc. Asia and South Pacific Design Automation Conf.*, pp. 408–413, January 2007.

[Malik 1991] S. Malik, E. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, Retiming and resynthesis: Optimizing sequential networks with combinational techniques, *IEEE Transactions on Computer-Aided Design*, 10(1), pp. 74–84, 1991.

[McCluskey 1956] E. J. McCluskey, Minimization of Boolean functions, *Bell Systems Technical J.*, 35(6), pp. 1417–1444, November 1956.

[McGeer 1987] P. C. McGeer and R. K. Brayton, Efficient, stable algebraic operations on logic expressions, in *Proc. IFIP Int. Conf. on Very Large Scale Integration*, August 1987.

[Mishchenko 2002] A. Mishchenko and R. K. Brayton, Simplification of non-deterministic multi-valued networks, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 557–562, November 2002.

[Mishchenko 2005] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, T. Villa, and N. Yevtushenko, Efficient solution of language equations using partitioned representations, in *Proc. Design Automation and Test in Europe*, pp. 418–423, March 2005.

[Mishchenko 2006a] A. Mishchenko and R. K. Brayton, A theory of non-deterministic networks, *IEEE Trans. on Computer-Aided Design*, 25(6), pp. 977–999, June 2006.

[Mishchenko 2006b] A. Mishchenko, S. Chatterjee, and R. K. Brayton, DAG-aware AIG rewriting: A fresh look at combinational logic synthesis, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 532–536, June 2006.

[Mishchenko 2007a] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, and S. Jang, SAT-based logic optimization and resynthesis, in *Proc. Int. Workshop on Logic Synthesis*, pp. 358–364, May 2007.

[Mishchenko 2007b] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, Combinational and sequential mapping with priority cuts, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 354–361, November 2007.

[Rudell 1987] R. Rudell and A. Sangiovanni-Vincentelli, Multiple-valued minimization for PLA optimization, *IEEE Trans. on Computer-Aided Design*, 6(5), pp. 727–751, September 1987.

[Rudell 1989] R. Rudell, *Logic Synthesis for VLSI Design,* Ph.D. dissertation, University of California, Berkeley, 1989.

[Yevtushenko 2001] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli, Solution of parallel language equations for logic synthesis, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 103–110, November 2001.

R6.4 Technology Mapping

[Aho 1976] A. Aho and S. Johnson, Optimal code generation for expression trees, *J. of the ACM*, 23(2), pp. 488–501, July 1976.

[Chatterjee 2006] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, Reducing structural bias in technology mapping, *IEEE Trans. on Computer-Aided Design*, 25(12), pp. 2894–2903, December 2006.

[Keutzer 1987] K. Keutzer, DAGON: Technology mapping and local optimization, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 341–347, June 1987.

[Kravets 2001] V. Kravets, *Constructive Multilevel Synthesis by Way of Functional Properties,* Ph.D. dissertation, University of Michigan, Ann Arbor, 2001.

[Lehman 1997] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, Logic decomposition during technology mapping, *IEEE Trans. on Computer-Aided Design*, 16(8), pp. 813–834, August 1997.

R6.5 Timing Analysis

[Chen 1991] H.-C. Chen and D. H. Du, Path sensitization in critical path problem, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 208–211, November 1991.

[Devadas 1992] S. Devadas, K. Keutzer, S. Malik, and A. Wang, Computation of floating mode delay in combinational logic circuits: Practice and implementation, in *Proc. Int. Symp. on Logic Synthesis and Microprocessor Architecture*, pp. 68–75, July 1992.

[Guerra E Silva 2002] L. Guerra E Silva, J. Marques-Silva, L. Silveira, and K. Sakallah, Satisfiability models and algorithms for circuit delay computation, *ACM Trans. on Design Automation of Electronic Systems*, 7(1), pp. 137–158, January 2002.

[McGeer 1989] P. McGeer and R. K. Brayton, Efficient algorithms for computing the longest viable path in a combinational network, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 561–567, June 1989.

[McGeer 1991] P. McGeer, A. Saldanha, P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, Timing analysis and delay-fault test generation using path-recursive functions, in *Proc. IEEE/ACM Int. Conf. on Computer Aided-Design*, pp. 180–183, November 1991.

R6.6 Timing Optimization

[Chaudhary 1992] K. Chaudhary and M. Pedram, A near optimal algorithm for technology mapping minimizing area under delay constraints, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 492–498, June 1992.

[Leiserson 1983] C. Leiserson and J. Saxe, Optimizing synchronous systems, *J. of VLSI and Computer Systems*, 1(1), pp. 41–67, Spring 1983.

[Leiserson 1991] C. Leiserson and J. Saxe, Retiming synchronous circuitry, *Algorithmica*, 6(1), pp. 5–35, December 1991.

[Rudell 1989] R. Rudell, *Logic Synthesis for VLSI Design,* Ph.D. dissertation, University of California, Berkeley, 1989.

[Singh 1992] K. Singh, *Performance Optimization of Digital Circuits,* Ph.D. dissertation, University of California, Berkeley, 1992.

[Touati 1990] H. Touati, *Performance-Oriented Technology Mapping,* Ph.D. dissertation, University of California, Berkeley, 1990.

R6.7 Trends in Logic Synthesis

[Brayton 2007] R. K. Brayton, The synergy between logic synthesis and equivalence checking, in *Proc. Formal Methods in Computer Aided Design,* (Tutorial), November 2007.

[Jiang 2007] J.-H. R. Jiang and W.-L. Hung, Inductive equivalence checking under retiming and resynthesis, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 326–333, November 2007.