BOOLEAN FUNCTIONS

INTRODUCTION

Boolean algebra is a part of *discrete mathematics*. The distinguishing feature of discrete mathematics is that value domains have only a finite number of elements; that is, continuous infinite domains such as the real numbers are not present. This area of mathematics is appropriate for modern computation, which is based on discrete-valued domains and discrete approximations to real numbers when required.

The direct coupling between the elegant, lean formalism of Boolean algebra and its many uses in the form of implementation tools and application products is a beautiful example of the synergy possible between mathematics and the engineering disciplines.

HISTORY OF MATHEMATICAL DEVELOPMENT

Boolean algebra began with the work of George Boole in 1854. Today, Boolean algebra has a general mathematical definition based on the definition given by E. V. Huntington in 1904. A more specific, limited definition used in engineering and computing applications is given in this article. This engineering definition is based in part on the *switching algebra* work done in 1938 by C. E. Shannon. Switching algebra is specific to two-valued switching elements that correspond closely to practical electrical switching devices.

DEFINITION OF BOOLEAN ALGEBRA

Boolean algebra is defined as the set B, three operations on the elements of B, and a set of axioms or postulates. B contains two elements: $B = \{0, 1\}$. These elements are also called the *truth values* where 0=*False*, and 1=*True* so B is also equal to $\{False, True\}$. The primitive operations are *and*, *or*, and *not*. These three functions are defined by both their syntax (how they are written) and their semantics (what they mean):

1. *and* is written as an infix $\wedge$ and defined by the equations $0 \wedge 0 = 0,\ 0 \wedge 1 = 0,\ 1 \wedge 0 = 0,$ and $1 \wedge 1 = 1$.
2. *or* is written as an infix and defined by the equations $0 \vee 0 = 0,\ 0 \vee 1 = 1,\ 1 \vee 0 = 1,$ and $1 \vee 1 = 1$.
3. *not* is written as an overbar and defined by the equations $\overline{0} = 1$ and $\overline{1} = 0$.

LANGUAGE OF BOOLEAN ALGEBRA STATEMENTS

A language exists that consists of legal Boolean expressions. The simplest forms of this language are written as combinations of the primitive truth values, the primitive operations, and parentheses. Occurrences of primitive functions in Boolean expressions require one (*not*) or two (*and* and *or*) truth values as arguments. When subexpressions are not simple truth values, they are enclosed in parentheses to indicate the proper grouping of the overall expression. Examples of simple Boolean expressions are $1,\ 1 \wedge 0,\ (1 \wedge \overline{0}) \vee 0$ and $(1 \wedge \overline{(}(\overline{0} \vee 1) \wedge 1)) \vee 0$. This concept of language is made more formal by the statement that a legal expression, ε, can be 0, 1, $\varepsilon\prime \wedge \varepsilon\prime\prime,\ \varepsilon\prime \vee \varepsilon\prime\prime$, or $\overline{\varepsilon}\prime$, where $\varepsilon\prime$ and $\varepsilon\prime\prime$ are also legal expressions.

Boolean expressions are made more general and useful by adding *variables*. Boolean-valued variables can take on (be bound to) either truth value. Variables are written as lowercase letters; they can be used wherever a truth value or a subexpression that reduces to a truth value could be used in an expression. Examples of Boolean expressions that contain variables are x, $x \vee y$, and $(a \wedge \overline{b})$.

Boolean expressions that include variables have multiple *readings*. Each unique variable in an expression can be interpreted as being either 0 or 1, consistently throughout the expression. Each unique variable thereby generates two versions of the expression. Each expression with n unique variables can be read as 2^n separate forms. For example, the expression $(a \wedge \overline{b}) \vee (\overline{a} \wedge b)$ has two variables and therefore four readings: when a and b are both 0: $(0 \wedge \overline{0}) \vee (\overline{0} \wedge 0)$; when a is 0 and b is 1: $(0 \wedge \overline{1}) \vee (\overline{0} \wedge 1)$; when a is 1 and b is 0: $(1 \wedge \overline{0}) \vee (\overline{1} \wedge 0)$; and when a and b are both 1: $(1 \wedge \overline{1}) \vee (\overline{1} \wedge 1)$.

This concept of variables is made more formal by the statement that a legal expression ε can be ν, 0, 1, $\varepsilon\prime \wedge \varepsilon'$, $\varepsilon\prime \vee \varepsilon'$, or $\overline{\varepsilon}\prime$, where ν is any variable and $\varepsilon\prime$ and $\varepsilon\prime\prime$ are also legal expressions.

AXIOMS AND THEOREMS

Boolean algebra has a set of *axioms* (or *postulates*) and *theorems*. Both axioms and theorems are given below as equations, which state that one Boolean expression is equal to another Boolean expression. Equality (=), in these equations, is a meta-symbol (equality played the same metarole in the definitions of *not*, *and*, and *or*). Equality between two Boolean algebra expressions means that both reduce to the same truth value for all 2^n readings generated by the n variables appearing in either or both of the two expressions.

Axioms:

A1. Closure: The set B is closed under the operations $\wedge$ and $\vee$.

A2. Commutative: $x \wedge y = y \wedge x$ and $x \vee y = y \vee x$.

A3. Distributive: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.

A4. Identities: There are two identity elements in B, *0* and *1*, such that $1 \wedge x = x$ and $0 \vee x = x$.

A5. Inverses: For each element in B, x, there is an inverse element in B, $\overline{x}$.

Theorems:

T1. Idempotent: $x \wedge x = x$ and $x \vee x = x$.

T2. Boundness: $x \wedge 0 = 0$ and $x \vee 1 = 1$.

T3. Involution (or double-negation): $(\overline{\overline{x}}) = x$.

T4. Complementarity: $x \wedge \overline{x} = 0$ and $x \vee \overline{x} = 1$.

T5. Associative: $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ and $(x \vee y) \vee z = x \vee (y \vee z)$.

T6. Logical Adjacency: $(\bar{a} \wedge b) \vee (a \wedge b) = b$ and $(\bar{a} \vee b) \wedge (a \vee b) = b$.

T7. Absorption: $x \wedge (x \vee y) = x$ and $x \wedge (x \wedge y) = x$.

T8. DeMorgan's Law: $\bar{(}x \wedge y) = (\bar{x} \vee \bar{y})$and $\bar{(}x \vee y) = (\bar{x} \wedge \bar{y})$.

Boolean algebra exhibits a fundamental *duality* in its axioms and theorems. *Dual forms* can be derived from each other by replacing all instances of *and* by *or*, *or* by *and*, 1 by 0, and 0 by 1. Thus, either form of any of the above axioms and theorems can be used to derive the other form.

PROOFS

Proofs exist for all of the theorems listed above. Three proofs are given below as examples of different proof styles. First we prove the *or* form of T2—the Boundness Theorem $(x \vee 1 = 1)$. Each step is justified with a specific axiom:

$$\begin{array}{ll} x \vee 1 & \{\text{given}\} \\ = 1 \wedge (x \vee 1) & \{\text{by A4}\} \\ = (x \vee \bar{x}) \wedge (x \vee 1) & \{\text{A5}\} \\ = x \vee (\bar{x} \wedge 1) & \{\text{A3}\} \\ = x \vee \bar{x} & \{\text{A4}\} \\ = 1 & \{\text{A5}\} \end{array}$$

Thus, in a series of five transformations, each justified by an axiom, we prove that $x \vee 1 = 1$. Proofs can be less formal. T3($(\bar{\bar{x}}) = x$) is proven in a more casual style: From $x \vee \bar{x} = 1$ and $x \wedge \bar{x} = 0$, and noting that the complement of $\bar{x}$ is denoted by $(\bar{\bar{x}})$, we get $\bar{x} \vee (\bar{\bar{x}}) = 1$, $\bar{x} \wedge (\bar{\bar{x}}) = 0$, and therefore, $(\bar{\bar{x}}) = x$.

Theorems can be proven by exhaustive consideration of cases over all combinations of argument values, as in the following proof of the first form of T6—the Logical Adjacency theorem, $(\bar{a} \wedge b) \vee (a \wedge b) = b$. The first two columns of Table 1 present all combinations of the variables found in the equation to be proven: a and b. The remaining four columns of the table establish the values of the various subexpressions of the theorem until the values for $(\bar{a} \wedge b) \vee (a \wedge b)$ are known. Then it can be observed that the values for $(\bar{a} \wedge b) \vee (a \wedge b)$ are the same as the values for b, and consequently, $(\bar{a} \wedge b) \vee (a \wedge b) = b$. Exhaustive consideration of cases is possible because the discrete-valued arguments have only a finite number of combinations.

Many axioms and theorems can be generalized to n variables as in the following restatement of DeMorgan's Law for n variables: $(\bar{x}_1 \vee x_2 \vee \ldots \vee x_n) = \bar{x}_1 \wedge \bar{x}_2 \wedge \ldots \wedge \bar{x}_n$ and $(\bar{x}_1 \wedge x_2 \wedge \ldots \wedge x_n) = \bar{x}_1 \vee \bar{x}_2 \vee \ldots \vee \bar{x}_n$.

REDUCTION

Boolean expressions can be reduced to truth values. *Reduction* is carried out by replacing the inner-most expressions with their equivalent truth values until the process terminates with a single truth value. Reduction is denoted by the symbol. For example, the expression on the left is reduced to the truth value on the right in a series of five reductions: $(1 \vee 0) \wedge \bar{(}0 \vee \bar{1}) \Rightarrow 1 \wedge \bar{(}0 \vee \bar{1}) \Rightarrow 1 \wedge \bar{(}0 \vee 0) \Rightarrow 1 \wedge \bar{0} \Rightarrow 1 \wedge 1 \Rightarrow 1$. Other orders of reduction are possible; they all produce the same result [for example, the first reduction could have been $(1 \vee 0) \wedge \bar{(}0 \vee \bar{1}) \Rightarrow (1 \vee 0) \vee \bar{(}0 \vee 0)$].

When variables are present in an expression, reduction can continue at least as long as a value is known for each variable. Thus, each of the 2^n readings of an expression with n variables can be completely reduced to a value in B. For example, when $x = 0$ and $y = 1$, the expression $(x \vee 1) \wedge y$ can be reduced as follows: $(x \vee 1) \wedge y \Rightarrow (0 \vee 1) \wedge y \Rightarrow 1 \wedge 1 \Rightarrow 1$.

Whenever values remain unknown for one or more variables, the expression cannot always be reduced to a value in B. In such cases, we say the expression has been *simplified* but not necessarily reduced. For example, when $x = 0$, but we have no known value for y, y, $(x \vee 1) \wedge y$ can be simplified: $(x \vee 1) \wedge y \Rightarrow (0 \vee 1) \wedge y \Rightarrow 1 \wedge y$.

This notion of reduction/simpification is strengthened by using the axioms and theorems of Boolean algebra as additional rules for reduction or simplification. Our example expression, $(x \vee 1) \wedge y$, can be simplified even when both x and y remain unknown by use of T2 and then A4: $(x \vee 1) \wedge y \Rightarrow_{T2} 1 \wedge y \Rightarrow_{A4} y$.

A Boolean expression is a *tautology (contradiction)* if it can be reduced to 1(0). $(x \vee 1) \wedge y$ is neither a tautology nor a contradiction because its value depends on the value of y. not b or ((not a and b) or (a and b)) is a tautology because of the reduction: not b or ((not a and b) or (a and b)) ⇒/T6 not b or b ⇒/T4 1.

CLOSED AND COMPLETE

Boolean algebra is *complete* in the sense that every Boolean function can be expressed in terms of the primitive operations: *not, and*, and *or* (this fact is demonstrated later in this article in the material on *canonical representations*). Other choices for the primitive functions also produce functional completeness; for example, both $\{not, and\}$ and $\{not, or\}$ are functionally complete. $\{nor\}$ and $\{nand\}$ are *minimal complete sets*, where *nor* and *nand* are negated versions of *or* and *and*, respectively [e.g., *nand* returns 0(1) when returns 1(0)]. No other complete set is minimal.

Boolean algebra is also closed. *Closure* means that every legal expression, when fully reduced, evaluates to a value in the primitive domain. Closure is asserted in A1 and follows immediately from the definitions of *and*, *or*, and *not*: All applications of these operations to values in B result in some value in B. As all other functions can be expressed as combinations the three primitives, they all must also preserve closure.

ADDITIONAL SYNTAXES AND OMITTED PARENTHESES

To this point only, the overbar, $\wedge$, and $\vee$ have been used to denote *not*, *and*, and *or*, respectively. However, several parallel syntaxes have evolved for Boolean expressions. *not* can be written in any of the following forms: $\bar{a}$, $not(a)$, $a\prime$, and $\sim a$. *and* can be written in any of the following forms: $a \wedge b$, $a\, and\, b$, $a * b$, and ab. *or* can be written

Table 1.

a	b	$\overline{a}$	$\overline{a}\wedge b$	$a\wedge b$	$(\overline{a}\wedge b)\vee(a\wedge b)$
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	0	0	0
1	1	0	0	1	1

Table 2. The 16 Functions of $B^2 \to B$

ab	0	$\wedge$	$\overline{a}\to b$	a	$\overline{b}\to a$	b	*xor*	$\vee$	$\overline{\vee}$	*xnor*	$\overline{b}$	$b\to a$	$\overline{a}$	$a\to b$	$\overline{\wedge}$	1
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 3. An Example Function of Three Variables

Arguments: (a, b, c)	Result: f(a,b,c)
000	0
001	1
010	1
011	1
100	0
101	1
110	1
111	0

Table 4. Behavior of Switching Logic

x	y	$x\vee y$	$x\wedge y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 5. Switching Algebra is Distributive

x	y	z	$y+z$	$x*(y+z)$	$x*y$	$x*z$	$(x*y)+(x*z)$	$x+y$	$x+z$	$y*z$	$x(y*z)$	$(x+y)*(x*z)$
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	1	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0	0
0	1	1	1	0	0	0	0	1	1	1	1	1
1	0	0	0	0	0	0	0	1	1	0	1	1
1	0	1	1	1	0	1	1	1	1	0	1	1
1	1	0	1	1	1	0	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1

in any of the following forms: $a \vee b$ *a or b* or $a + b$.

Also to this point, subexpressions have been fully parenthesized. A less rigid syntax can be used in which some parentheses can be omitted. Where parentheses are omitted, it is necessary to define the order in which operations are applied, as though full parenthesization were present. For example, should the expression $a \wedge b \vee \overline{c} \wedge d$ be interpreted as $a \wedge ((b \vee \overline{c}) \wedge d)$ or some other of the possible parenthesized interpretations? The question is important, as different interpretations may reduce to different truth values [consider $0 \wedge 0 \vee 1 \wedge 1$: $(0 \wedge (0 \vee 1)) \wedge 1 \Rightarrow 0$, but $((0 \wedge 0) \vee 1) \wedge 1 \Rightarrow 1$]. Within the context of any set of parentheses, this ambiguity is resolved by applying all *not's* first, then applying all instances of *and*, then applying all instances of *or*, and then applying any other defined functions. Within equal precedences (e.g., several *and*'s in a row with no parentheses), the left-most operation is taken first. For the above example, the implied parenthesization is $(a \wedge b) \vee ((\overline{c}) \wedge d)$.

DEFINITION OF FUNCTIONS

New functions can be defined, based on the primitive functions, variables, parentheses, and other defined functions. For example, the standard *exclusive or* function (*xor*) is defined as $xor(x, y) = (\bar{x} \wedge y) \vee (x \wedge \bar{y})$. Equality (=) takes on a second role here: It defines a new function with a new name (e.g., *xor*), a specification of the number and name of arguments (formal parameters) the new function takes (e.g., two arguments, named x and y), and a Boolean expression that defines the new function in terms of the arguments and other known entities [e.g., $(\bar{x} \wedge y) \vee (x \wedge \bar{y})$]. The defined functions of two arguments can be written as infix functions, for example, $(x\ xor\ \bar{y})\ and\ z$. Functions of one or more than two arguments are written in prefix form with parentheses and commas, for example, $foo(x, y, z)$.

Replacement of functions by their definitions, with proper replacement of formal parameters by actual parameters, provides additional rules for reduction/simplification. The following example uses the definition of *xor* given above to simplify a given expression. Although the resultant expression is "larger," it is simpler in the sense that its continued simplification requires only the use of the primitive operators.

$xor(\bar{a}, b) \wedge (c \vee d)$	{given}
$((\bar{a} \wedge b) \vee (a \wedge \bar{b})) \wedge (c \vee d)$	definition of *xor*: *xor's* formal parameters, x and y, are (consistently) replaced by the actual parameters $\bar{a}$ and b

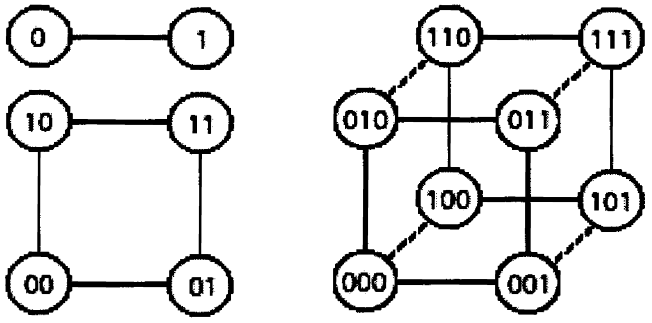

Figure 1. B^1, B^2, and B^3 and cubes.

CARTESIAN PRODUCTS AND BOOLEAN CUBES

Boolean functions take one or more values from B and produce a value, also in B. There are two useful ways to think about the arguments of functions of more than one argument: The arguments can be thought of as being separately taken or sampled from B, or one can think of a collection of arguments being sampled from a more complex domain, a Cartesian product on B, denoted B^n. A *Cartesian product* B^n is an ordered set of all combinations of independent samplings from B. Cartesian products are written as ordered, comma-separated lists enclosed by angle brackets. B^2 is the set of ordered pairs taken from B or $B^2 = \{<0, 0>, <0, 1>, <1, 0>, <1, 1>\}$. A function such as *nor* can be interpreted as taking two independent arguments from the domain B or as taking one argument from the Cartesian product domain B^2. This second interpretation is described by the following type definition $nor :: B^2 \to B$, which means that "*nor* is a function of type B^2 to B" or that *nor* takes one argument from B^2 and returns a result in B.

B^3 is the Cartesian product domain formed by taking all combinations of three values from $B^2 = \{<0, 0>, <0, 1>, <1, 0>, <1, 1>\}$

$B : B^3 = \{<0, 0, 0>, <0, 0, 1>, <0, 1, 0>, <0, 1, 1>, .$
$<1, 0, 0>, <1, 0, 1>, <1, 1, 0>, <1, 1, 1>\}$

For many purposes, $B^1 = \{<0>, <1>\}$ is taken to be the same domain as $B = \{0, 1\}$. The size of b^n (its *cardinality*, written as $|B^n|$) is 2^n. This exponential growth in the size of the argument domains is a fundamental source of difficulty in dealing with Boolean functions: It is generally too difficult to individually reason about (or compute) all possible inputs to a function.

The domains B^n are also called *cubes*, as they form *binary hypercubes* of degree n. They can be drawn as a labeled node for each value in the domain and an edge connecting all values, which differ in only one position in their labels, i.e, a *dimension* of the cube. The cubes for B^1, B^2, and B^3 are illustrated in Fig. 1 . A B^n cube has n variables; each variable corresponds to a *dimension* or *coordinate* of the cube and to a position in all of the node labels. Each edge in a cube corresponds to a change in value of one variable.

Subcubes are k-cubes imbedded in n-cubes, where $k \leq n$. Each node of the k-cube is mapped to a node of the n-cube. The selected n-cube nodes for all k-cube nodes share a common value for the n-k dimensions of address, which the K-cube lacks. Each pair of n- and k-cube nodes share the same value for the other k dimension. The choice of which dimension is common is free, as is the value of the $n - k$ common variables. Thus, there are $2^{n-k}\binom{n}{k}$ embeddings of a k-cube in an n-cube. In Fig. 2 , a 2-cube is embedded in a 3-cube with the right-most dimension of the 3-cube being fixed at 1. The embedded 2-cube is in bold.

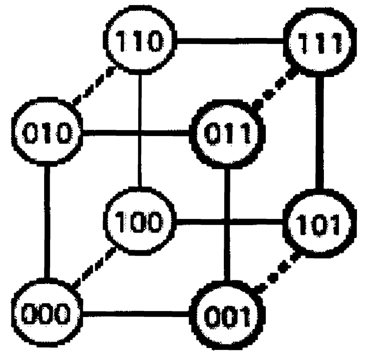

Figure 2. A 2-cube embedded in a 3-cube.

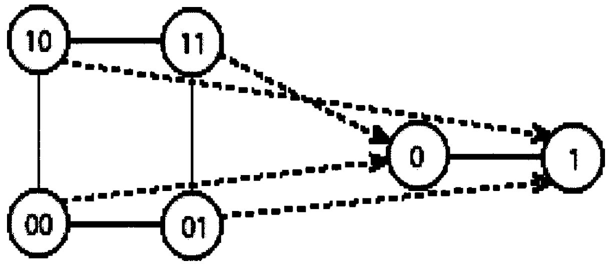

Figure 3. Cube-to-cube functional mapping of *xor*.

FUNCTIONS OF $B^N \to B$

There are four functions of type $B^1 \to B$: *not* (described above), the identity function ($identity\, 0 = 0,\ identity\, 1 = 1$), the everywhere zero function ($zero\, 0 = 0,\ zero\, 1 = 0$), and the everywhere one function ($one\, 0 = 1,\ one\, 1 = 1$). There are 16 functions of type $B^2 \to B$, as shown in Table 2 .

There are $2^{(2^n)}$ distinct functions from $B^n \to B$: B^n has 2^n unique combinations of argument values; each of the 2^n unique inputs to a $B^n \to B$ function can result in either truth value, so there are $2^{(2^n)}$ unique functions in $B^n \to B$. The population of functions grows exponentially with the number of arguments ($B^3 \to B$ has 256 functions; $B^4 \to B$ has 65,536 functions); this enormous population of functions is a second fundamental source of difficulty in dealing with Boolean functions in automated systems.

Functions are mappings from one domain to another. Such mappings can be conceptualized and drawn as directed arcs from one cube $((B^n))$ to another $((B^1))$. The mapping for the exclusive or function ($xor(x, y) = \bar{x}y \vee x\bar{y}$) is illustrated in Fig. 3 .

To this point, only functions that produce a single truth value have been considered (functions in $T^n \to T$). It is common, however, to group functions of this type to produce one function that produces an m-bit vector. These bundled functions are considered to be functions of type $T^n \to T^m$.

TRUTH TABLES

There are numerous ways to encode the mapping represented by a Boolean function. The most straightforward method is the truth table. A *truth table* is a tabular list of all the arguments of a function in one column with the associated results of the function listed in a second column, each row constituting of one argument and the associated result of the function. An example of a truth table for an arbitrary function in is given in Table 3 .

The truth table in Table 3 completely specifies the behavior or mapping of a particular $B^3 \to B$ function, but it does not indicate how the function is specified in Boolean algebra. Although it is commonly convenient to work with functions in various tabular and graphical forms, it is sometimes necessary to translate such representations to algebraic forms. This process is straightforward and leads to two canonical representations of Boolean functions.

CANONICAL REPRESENTATIONS (SOP AND POS)

The first *canonical representation* is the *canonical sum-of-products* (SOP) form: A *literal* is a variable (a) or its negation ($\bar{a}$). Boolean expressions constructed with only *and* and literals are called *products-of-literals* (or just *products)*. Examples of products are $a \wedge \bar{b}$ and $\bar{a} \wedge c$. *Sums-of-products-of-literals* (or just *sums-of-products)* are Boolean expressions constructed with only *or* and products. An example of an SOP is $(a \wedge \bar{b}) \vee (\bar{a} \wedge c)$. A product that contains n distinct literals (i.e., literals made from each n different argument) for a function with n arguments is called a *minterm*. The function in Table 3 has eight minterms ($\bar{a} \wedge \bar{b} \wedge \bar{c}$ through $a \wedge b \wedge c$). The function returns *True* for only five of those minterms ($\bar{a} \wedge \bar{b} \wedge c, \bar{a} \wedge b \wedge \bar{c}, \bar{a} \wedge \bar{b} \wedge c, a \wedge \bar{b} \wedge c$ and $a \wedge b \wedge \bar{c}$). We refer to these five points as the *true minterms*. The sum of all true minterms must be true if the function is True and must be False otherwise. Thus, the sum of all true minterms is always a correct algebraic expression for the function. The function in Table 3 can be written as

$$f(a, b, c) =$$
$$(\bar{a} \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$$

This SOP formulation gives a unique, canonical expression for each function, if the products are written in a standard order. The standard order uses the base-two representation of the natural numbers, where negative literals are taken as 0s and positive literals are taken as 1s. Thus, in the SOP form above, $\bar{a} \wedge b \wedge \bar{c}(010)$ precedes $\bar{a} \wedge b \wedge c(011)$ in the standard ordering of SOPs.

A dual representation of functions is called the *product-of-sums (-of-literals)* (POS). *Sums* are formed from literals and *or*, and *products-of-sums* are formed from *sums* and *and*. *Sums* of n distinct literals are called *maxterms*. Each *maxterm* is used to rule out the points of the function, for which the function returns 0. If a zero of the function occurs at maxterm $x \wedge \bar{y} \wedge \bar{z}$, then the sum $\bar{x} \vee y \vee z$ rules out that zero-producing point (i.e., the sum $\bar{x} \vee y \vee z$ is True when and only when the argument of the function does not occur at $x \wedge \bar{y} \wedge \bar{z}$). If all zero points are ruled out for a given set of arguments, then the result produced by the product is *True*. A POS representation for the function in Table 3 is

$$f(a, b, c) = (a \vee b \vee c) \wedge (\bar{a} \vee b \vee c) \wedge (\bar{a} \vee \bar{b} \vee \bar{c})$$

MINIMIZATION

The *SOP* forms introduced above are canonical representations of Boolean functions, but they may not be minimal representations. Boolean functions may have more than one *SOP* representation. This multiplicity of *SOP* representations is introduced in two ways: 1) trivially, by allowing different orderings of the terms; and 2) substantially, by using product terms with fewer than n literals. Each product with $m < n$ literals covers or replaces 2^{n-m} minterms. Of all possible *SOP* representations for a function, a minimal *SOP* representation requires the fewest logic operations. More than one minimal *SOP* form for a function may exist. Consider the function in Table 3 . It has the canonical SOP representation:

1. $(\overline{a} \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge b \wedge c) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c})$ but this function is also correctly expressed by other forms, including:
2. $(\overline{b} \wedge c) \vee (\overline{a} \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge b \wedge c) \vee (a \wedge b \wedge \overline{c})$
3. $(\overline{a} \wedge c) \vee (\overline{a} \wedge b \wedge \overline{c}) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c})$
4. $(\overline{b} \wedge c) \vee (b \wedge \overline{c}) \vee (\overline{a} \wedge c)$
5. $(\overline{b} \wedge c) \vee (b \wedge \overline{c}) \vee (\overline{a} \wedge b)$

These five expressions have 14, 10, 10, 5, and 5 *and* or *or* operators, respectively. Expressions 4 and 5 have the smallest number of operators possible. Either one of them is a suitable minimal SOP. Dual results apply to POS forms.

The reduced expressions for the function in Table 3 can be derived systematically by repeated application of logical adjacency. If two minterms differ in only one literal (as with $\overline{a} \wedge \overline{b} \wedge c$ and $\overline{a} \wedge b \wedge c$), then they can be removed and replaced by the single product that has all the shared literals from the two minterms but drops the single literal at which the minterms differ (resulting in $\overline{a} \wedge c$ for the example). This algebraic manipulation can be applied as well to products that are not minterms (i.e., that have fewer than n literals). The process can be continued until more reductions are available. Depending on choices among several simultaneously available logical adjacencies, this method may or may not result in a minimal expression. With complete backtracking at all choice points, the minimal expression can be found, but there are better algorithms for this purpose.

The process of finding a minimal SOP form for any function is an important problem. For small functions, straightforward graphical techniques can be used (e.g., Karnaugh Maps); for functions with more arguments, more sophisticated algorithms are required. The worst-case time required to find a minimal expression for a function grows exponentially with the number of arguments ($O(2^n)$); consequently, for large functions, it may be necessary to give up on finding a truly minimal expression and be satisfied with a relatively small expression, found in more modest time.

All methods for finding minimal *SOPs* share four common concepts. These are concepts: *implicants, prime implicants, essential prime implicants,* and *minimal covers.*

Implicants (I) are those products that are true only where the function is true. The implicants of the function in Table 3 are all five of the n-literal true minterms identified from Table 3 's true rows and from, $b \wedge \overline{c}$, $b \wedge \overline{c}$, $\overline{a} \wedge c$, and $\overline{a} \wedge b$. No other products cover only true results in the function.

Prime implicants (PI) are those implicants that are not completely covered by a larger implicant. Among the implicants of the example, it can be observed that $\overline{a} \wedge c$ covers $\overline{a} \wedge \overline{b} \wedge c$. Therefore, $\overline{a} \wedge \overline{b} \wedge c$ is not a PI. The PIs for the function given in Table 3 are $\overline{b} \wedge c$, $b \wedge \overline{c}$, $\overline{a} \wedge c$, and $\overline{a} \wedge b$.

Essential prime implicants (EPIs) are those PIs that cover true minterms that are not covered by other PIs. In the example, it can be observed that $\overline{b} \wedge c$ and $b \wedge \overline{c}$ are essential, as only they cover the minterms $a \wedge \overline{b} \wedge c$ and $a \wedge b \wedge \overline{c}$, respectively.

The *minimal cover* is the set of EPIs combined with the smallest number of nonessential prime implicants (N-EPIs are those prime implicants that are not essential) required to cover all true minterms of the function. In the case of the function in Table 3 , the EPIs ($\overline{b} \wedge c$ and $b \wedge c$) are sufficient to cover only four true minterms, so N-EPI(s) are required to cover the remaining true minterm of the function $\overline{a} \wedge b \wedge c$. We chose $\overline{a} \wedge c$, but $\overline{a} \wedge b$ would have done as well. Thus, a minimal (but not unique) SOP form for the example function is $(\overline{b} \wedge c) \vee (b \wedge \overline{c}) \vee (\overline{a} \wedge c)$.

Minimization algorithms generally start from the truth table or SOP form of a function, then build the set of all implicants, then filter out the non-PIs, then divide the PIs into EPIs and N-EPIs, and finally make the choices among the N-EPIs to provide a complete minimal cover.

CYCLES OF CANONICAL REPRESENTATIONS

Any function can be written as an SOP; any function can be written as a POS. Six other, related canonical forms can be reached by the application of DeMorgan's rule to the *SOP* and *POS* forms. These forms are illustrated in Fig. 4 , where two rings of forms are shown, one including the *SOP* and the other including the *POS*. Whereas *SOP* means "*or's* of *and's*", a form like $SO\overline{S}$ means "*or's* of *nor's*". Translation from one form in a ring to another is accomplished by application of DeMorgan's Law: The dotted lines correspond to the transformations accomplished by application of DeMorgan's Law to the inner expressions; the solid lines correspond to the transformations accomplished by application of DeMorgan's Law to the outer expressions. The ability to translate standard positive two-level logic (*SOP*) to a doubly negative form ($\overline{P}O\overline{S}$) is important, as most implementation technologies naturally produce negated logic forms.

For some new example function, the four SOP-related forms are:

$$\begin{aligned}
SOP &: (a \wedge b \wedge c) \vee (\overline{a} \wedge b \wedge c) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c}) \\
SO\overline{S} &: \overline{(}\overline{a} \vee \overline{b} \vee \overline{c}) \vee \overline{(}a \vee \overline{b} \vee \overline{c}) \vee \overline{(}a \vee b \vee c) \\
\overline{P}O\overline{S} &: \underline{\overline{(}\overline{a} \vee \overline{b} \vee \overline{c})} \wedge (a \vee \overline{b} \vee \overline{c}) \wedge (a \vee b \vee c) \\
\overline{P}OP &: \overline{(}a \wedge b \wedge c) \wedge \overline{(}\overline{a} \wedge b \wedge c) \wedge \overline{(}\overline{a} \wedge \overline{b} \wedge \overline{c})
\end{aligned}$$

Algebraic conversion between SOP and POS can be performed by multiplying out the operations, but this process takes $O(2^n)$ steps for n variables. An example SOP form is

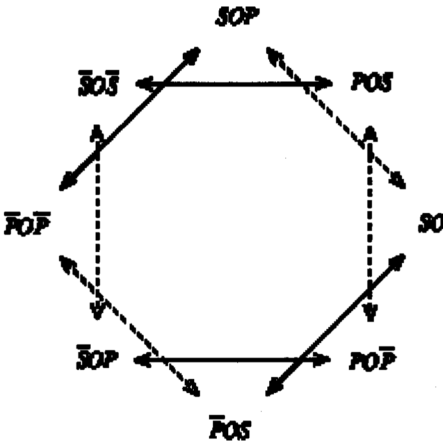

Figure 4. SOP and POS rings of canonical forms.

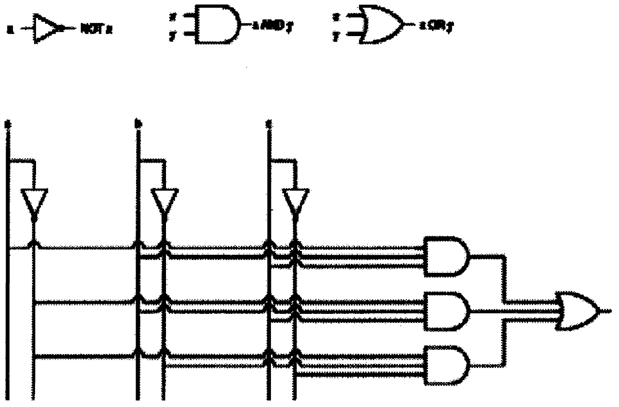

Figure 5. Logic gate schematic diagrams.

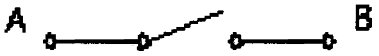

Figure 6. A simple switch.

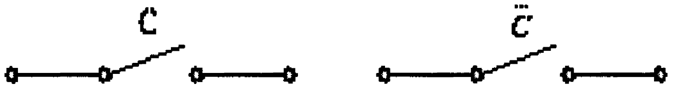

Figure 7. Positive and negative logic switches.

transformed to its POS form in the following steps:

$$(a \wedge b) \vee (\bar{a} \wedge c)$$
$$\Rightarrow (a \vee (\bar{a} \wedge c)) \wedge (b \vee (\bar{a} \wedge c))$$
$$\Rightarrow (a \vee \bar{a}) \wedge (a \wedge c) \wedge (b \vee \bar{a}) \wedge (b \vee c)$$
$$\Rightarrow (a \vee c) \wedge (\bar{a} \vee b) \wedge (b \vee c)$$

NESTED LOGIC FORMS

SOP and POS are two-level logic forms in the sense that the inner level of logic must produce a result for the outer level of logic to consume but that no deeper nesting of logic is required (the negations of the literals are not counted). Often, more deeply nested logic is more efficient in representing a function in terms of logic gate count. The study of these nested forms of logic is less structured than the study of two-level logics, but the area is important in practice. Nested logic forms can be found by factoring two-level forms. For the example in Table 3 (with SOP form $(\bar{a} \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$), a factor of $\bar{a}$ can be extracted from the first three products and a factor of a can be extracted from the last two products, yielding $(\bar{a} \wedge ((\bar{b} \wedge c) \vee (\bar{b} \wedge c) \vee (b \wedge c))) \vee (a \wedge ((\bar{b} \wedge c) \vee (b \wedge \bar{c})))$. This factored form requires three levels of logic but only 11 logic gates compared with 14 for the SOP form.

In many applications of Boolean functions, there may be combinations of input arguments that never occur or for which the results of the function do not matter. These input combinations are called *don't-cares*. Truth table specifications of functions with don't-cares include 0s, 1s, and don't-cares (which are commonly written as Xs). In the minimization problem, don't-cares can be treated as 0s or 1s, which are chosen such that the resultant expression is maximally reduced.

LOGIC GATES

Boolean expressions can be drawn in a graphical circuit form. Figure 5 illustrates the basic logic gate types and a logic diagram for the function in Table 3 . Three input *nand*/*nor* gates are equivalent to *(x nand y) nand z* or *(x nor y) nor z*. Negations on inputs of gates can be also be drawn as small circles right at any input(s) of any gate type.

SWITCHING CIRCUITS

Switching circuits are closely related to Boolean algebras. A simple switch is shown in Fig. 6 . It has the fundamental property that it is either *on (closed)* or *off (open)*. When it is on, A and B are connected together and current can flow. This switch is a two-state logical device.

A single switch is denoted by a letter or variable. When $c = 1$ ($c = 0$), the circuit is closed (open). The negation of this circuit will be closed (open) when $c = 0$ ($c = 1$) (Fig. 7).

If two switches d and e are connected in series, the circuit is closed if and only if both switches are closed. The transmission function of two switches in series may therefore be written as $f \wedge g$. Similarly the parallel connection of two switches (Fig. 8) is closed if and only if one or both of them is closed. The transmission function for two switches in parallel is $f \vee g$.

In summary, the rules of the two binary operators $\vee$ and $\wedge$ corresponding to the circuit operations described above are shown in Table 4 .

We now show that the switching algebra is a Boolean algebra. *Closure* for the three primitive operations on the set $B = \{0, 1\}$ is obvious, because the result of each operation is either 0 or 1 and $0, 1 \in B$. From Table 4 , the commutative laws are obvious as well. The identity elements are verified as $1 \wedge 1 = 1$, $1 \wedge 0 = 0$, $0 \vee 0 = 0$, and $0 \vee 1 = 1$. From $\bar{0} = 1$ and $\bar{1} = 0$, the inverse property is satisfied. Table 5 shows that the distributive laws are satisfied (column 5 = column 8 and column 12 = column 13). Notice that Table 5 uses "*"

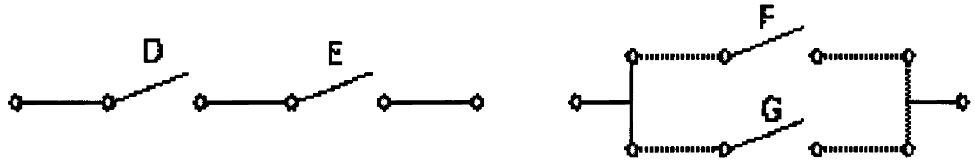

Figure 8. Series and parallel switches.

for *and* and "+" for *or*.

Therefore, the set $B = \{0, 1\}$ with the defined switching operations form a Boolean algebra.

COMPLEMENTS AND RESTRICTIONS

Given a Boolean function $f :: B^n \to B$, its *complement* function, denoted by $f'(x)$ or $\overline{f}(x)$, is defined as $\overline{f}(x) = 0$, *iff* $f(x) = 1$, and $\overline{f}(x) = 1$ *iff* $f(x) = 0$.

For an n-dimensional Boolean space B^{n} with variable set $\{x_1, \ldots, x_n\}$, the function *restriction* (also called *cofactor)* of a Boolean function $\{x_1, \ldots, x_n\}$ with respect to a variable x_i is defined to be $f_i(x_1, \ldots, x_{i+1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. Similarly, we can define $f_{\bar{i}}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. Of course f_{i} and $f_{\bar{i}}$ are of type $B^{n-1} \to B$. Similarly we recursively define $f_{x_i x_j} = (f_{x_i})_{x_j}$. For the function $f = x^1 x^2 \vee x^3 x^4$, we have $f_{x^1} = x^2 \vee x^3 x^4$ $f_{\overline{x}^1} = x^3 x^4$, $f_{\overline{x}^1 x^3} = x^4$.

A variable x_{i} is a *supporting variable* of the function f if $f_{x_i} \neq x_{\overline{x}_i}$. The supporting variable set, denoted by *sub(f)*, consists of all supporting variables of *f*. The *dimension* of a function *f* is the number of supporting variables.

DATA STRUCTURES FOR BOOLEAN FUNCTIONS

Automated algorithmic processing of Boolean functions is challenging in large part because of the exponential size of the domains (B^{n}) and the number of functions (2^{2^n}). Truth tables quickly become impractical for many common problems. Considerable research has been undertaken to find more efficient data structures to represent and manipulate Boolean functions. In general, all such representations have bad cases that suffer exponential growth. But for many practical problems, significantly more efficient representations have been identified. Chief among these new representations are the ordered binary decision diagram (OBDD) and its many close relatives reported in the literature, for example, factored forms such as the Shannon expansion and forms based on *exclusive or* such as Reed–Muller expansions.

ALGORITHMS FOR BOOLEAN FUNCTIONS

The most commonly studied algorithms for Boolean functions are for the minimization problem, which accept a truth table, SOP, or other representation of a function and compute either a minimal or some nearly minimal expression for that function. Practical solutions to this problems have been important, as the cost of implementation has been dependent on progress on this research front. Other important algorithmic problems are *complementation*, *tautology checking*, computing *restrictions* of functions, and *solving* Boolean equations. Algorithms for *conversion* of data from one Boolean-based representation to another are also of practical importance.

CLASSIFICATION OF BOOLEAN FUNCTIONS

The classification of Boolean functions by various properties inherent in their mappings has been important in a variety of advanced research topics. Several forms of *symmetry* can be used to classify functions and can lead to practical advantages. A concept of *threshold functions* can serve as primitive functions for the implementation of all other functions. In some possible technologies, an advantage exist to implementation in these threshold functions and, hence, to in understanding the cost of implementation of other functions in one or more threshold functions.

STANDARD BOOLEAN FUNCTIONS

Many standard applications of digitial logic (such as processors, memories, and telecommunications gear) can be largely described as being composed of customized versions of a fairly modest set of higher level components. Key examples of these higher level constructs are *multiplexers, demultiplexers, adders, decoders, encoders*, and *priority chains*. Each of these higher level constructs has one or several straightforward implementations in terms of the standard Boolean gates. These constructs allow the level of abstraction of design to be raised without any compromise to the formal basis of Boolean algebra or to the practicality of the resultant designs.

IMPLEMENTATIONS OF BOOLEAN FUNCTIONS

The primitive Boolean functions can be implemented in a remarkably wide range of technologies from the obvious electronic technologies to pneumatic solutions. The simple rules of composition of the logical primitives are mirrored by equally simple "wiring" rules in any implementation technology. As a consequence, it is straightforward to design and often practical to implement quite complicated designs in a wide variety of technologies. By far the most common basis for implementation of digital systems today is CMOS. CMOS is capable of implementing systems consisting of millions of interconnected logic gates.

COMPUTER AIDED DESIGN

It is now commonplace to develop systems—based on these foundations of Boolean algebra—that are far too complicated for “hand” or “paper” design. It is necessary to build and use computer aided design (CAD) tools to allow correct completion of most practical systems. Many approaches to CAD and the design of CAD tools exist, but the two most important characteristics of these systems are that 1) they should allow the production of provably correct systems, and 2) they should allow the designers to work at higher levels of abstraction than logic gates, with mostly automatic generation (synthesis) of the lower layer representations of the system until a realizable implementation in the primitive gates is obtained.

STATE MACHINES

This short article has left most of the literature and applications of Boolean algebra and the switching algebra undiscussed. However, perhaps the most important single omission is the idea of *state machines*. State machines have internal memory that represents the current state of some computational process. Logic, based on Boolean algebra, is used to compute a next state and some set of outputs from the state machine, based on the current state and/or the current inputs. The idea of internal state that is preserved from one time period until the next is an entirely new element to the discussion thus far in this article. The internal state corresponds to a feedback mechanism, or to recursive equations that track the state-by-state course of these machines over computational time. This seemingly simple addition to the ideas of Boolean algebra is essential for most practical applications. Although state (and the memory mechanisms that record this changing state) is essential to most practical applications, it introduces semantics and mathematics that go far beyond this introductory article.

BIBLIOGRAPHY

Reading List

The following textbooks contain a variety of levels of useful overviews of, and introductions to, Boolean functions and their applications:

Friedman, A. D., Menon, P. R. *Theory and Design of Switching Circuits*, Computer Science Press: Woodland Hills, CA, 1975.

Gajski, D. D. *Principles of Digital Design*, Prentice-Hall: Upper Saddle River, NJ, 1997.

Katz, R. H. *Contemporary Logic Design* Benjamin Cummings: Redwood City, CA, 1994.

Mano, M. M. *Computer Engineering: Hardware Design*, Prentice-Hall: Englewood Cliffs, NJ, 1988.

Roth, C. H. *Fundamentals of Logic Design*, 4th ed., West: St. Paul, MN, 1994.

Wakerly, J. F. *Digital Design: Principles and Practices*, 2nd ed., Prentice-Hall: Englewood Cliffs, 1994.

CARL D. MCCROSKY
YUKE WANG
Department of Electrical and Computer Engineering, University of Saskatchewan
Department of Computer Science, University of Texas at Dallas