Boolean Algebra - Basics

Instructor: Priyank Kalla
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT 84112
Email: kalla@ece.utah.edu

In this lecture, basic terms and definitions corresponding to Boolean algebra are defined. Various characteristics of Boolean functions are analyzed and operations upon them are discussed. Boolean algebra concepts are explained both from a mathematical (functional) as well as from a set theory perspective.

I. Boolean Algebra and Applications

An algebraic system is the combination of a set and one or more operations. A Boolean algebra is defined by the set $B \supseteq \mathbf{B} \equiv \{0, 1\}$ and by two operations, denoted by + and $\cdot$ which satisfy the *commutative* and *distributive* laws and whose identity elements are 0 and 1, respectively. Any element $a \subseteq B$ has a complement, denoted by a', such that $a + a' = 1$ and $a \cdot a' = 0$. These axioms, which define a Boolean algebra, are often referred to as Huntington's postulates [1].

We often use formulae to describe functions, but we have to keep in mind that the two are distinct. Many Boolean formulae can describe a Boolean function. The formula $f = a + b$ is equivalent to $f = a + a'b$, which can have many equivalent formulae. These are formulae defining a function; then what is the function? From the last lecture, a function is a particular mapping between domain and co-domain points. The formula represents the mapping of the function.

Definition I.1: Given a Boolean algebra B, **Boolean formula** on n variables $x_1 \ldots x_n$ is defined recursively as:

- Elements of B are Boolean formulae.
- $x_1 \ldots x_n$ are Boolean formulae.
- If g and h are Boolean formulae then $g \cdot h, g + h, \overline{g}$ or g' are also Boolean formulae, and
- Any string that can be derived by applying the above rules is also a Boolean formula.

There are many examples of Boolean algebraic systems, for example set theory, propositional calculus, arithmetic Boolean algebra [2], etc. In this chapter we consider only binary Boolean algebra, where $B = \mathbf{B} = \{0, 1\}$ and the operations + and $\cdot$ are *disjunction* and *conjunction*, respectively. The multi-dimensional space spanned by n binary-valued Boolean variables is often referred to as n-dimensional cube. Shown below is an example of a 3-dimensional Boolean function modeled on the 3-D cube, along with its 3-var K-map. Take a look at this carefully (see slides).

Definition I.2: A **completely specified Boolean function**, F, of n variables is a mapping $f : B^n \rightarrow B$, where $B = \{0, 1\}$. We model B^n as a binary n-cube. Vertex v in the binary n-cube for which $F(v) = 1$, is a member of the set called the ON-set of F. If $F(v) = 0$ then v is a member of the set called the OFF-set of F.

An n-input, m-output Boolean function is a mapping $f : B^n \rightarrow B^m$. It can be considered as an array of m scalar functions over the same domain.

An *incompletely specified* Boolean function is defined over a subset of B^n. The points where the function is not defined are called *don't care* conditions. They are related to input patterns that can never occur and to those for which the output is not observed. Incompletely specified functions are represented as $f : B^n \rightarrow \{0, 1, *\}^m$, where the symbol $*$ denotes the *don't care* condition.

Definition I.3: A **binary variable** is a symbol representing a single coordinate of the Boolean space B^n.

Definition I.4: A **literal** is a Boolean variable or its complement.

Definition I.5: A **cube** is defined as a product of literals. It denotes a point, or a set of points, in the Boolean space.

Let $f(x_1, \ldots, x_n)$ be a Boolean function of n variables. The set $\{x_1, x_2, \ldots, x_n\}$ is called the *support* of the function f.

A function can be represented as a sum of products of n literals, called the **minterms** of the function; alternatively product of sums and *maxterms* of the function. Operations on Boolean functions over the same domain can be viewed as operations on the set of their minterms. In particular, sum and product of two functions are the union ($\cup$) and intersection ($\cap$) of their minterm sets, respectively. Implication between two functions corresponds to the containment ($\subseteq$) of their minterm sets. The **cardinality** of a set is the number of elements it contains. In switching theory terminology, the cardinality of a Boolean function corresponds to the number of minterms contained in the function.

II. Operations on Boolean Functions

Definition II.1: The **cofactor** of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to variable x_i (the positive cofactor) is $f_{x_i} = f(x_1, \ldots, 1, \ldots, x_n)$. The cofactor of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to variable $\overline{x_i}$ (the negative cofactor) is $f_{\overline{x}_i} = f(x_1, \ldots, 0, \ldots, x_n)$.

The Boole's expansion (also called the Shannon's expansion) of a function over a variable is given as follows [2] [3]: Let $f : B^n \rightarrow B$. Then,

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i \cdot f_{x_i} + \overline{x}_i \cdot f_{\overline{x}_i} \tag{1}$$

Significance of Boole's expansion : This is the most fundamental operation on Boolean functions - and its significance comes from the fact that any Boolean function can be *decomposed* into its co-factors, and *re-composed* to form the original function. Why would anyone want to do this decomposition? When a Boolean function becomes too large to manipulate, you can decompose it using the above expansion, perform the manipulations on the co-factors, and then recompose the original function. The co-factors of a function are smaller in cardinality than the function itself and that is how we can break a big function down into smaller pieces and operate on those smaller pieces easily. Another point: The expansion can be performed recursively over the variables – so if you are writing a computer program and using Shannon's decomposition, a recursive subroutine is the best option. On top of that, by decomposing w.r.t. a variable, the complexity of the problems reduces exponentially! (Remember exponential behaviour of Boolean functions?) See the example below:

Definition II.2: A function $f(x_1, \ldots, x_i, \ldots, x_n)$ is positive (negative) **unate in variable** x_i if $f_{x_i} \supseteq f_{\overline{x}_i}$ ($f_{\overline{x}_i} \supseteq f_{x_i}$). Otherwise it is **binate** in that variable.

For a function f positive unate in x_i, the set of minterms of f_{x_i} includes the set of minterms of $f_{\overline{x}_i}$; the opposite follows for negative unate functions. A function is unate if it is (positive/negative) unate in all variables in its support. Otherwise it is binate. Unateness is similar to monotony, and unate functions are also called monotonic functions. An example of monotonic function $f = a + b + c'$ shown below:

Characteristics of unate functions:

- If f is +ve unate in x then the Boole/Shannon expansion leads to $f = x \cdot f_x + f_{x'}$.
- Note that f_x (also $f_{x'}$) has one less variable - it does not contain x in its support.
- If f is -ve unate in x, then $f = f_x + x' \cdot f_{x'}$. Can you prove the above?
- Note $\overline{f}_x = \overline{f_x}$. Can you prove this too?
- We will use the above properties when we will discuss the satisfiability problem. Remind me about it if I forget...

Unate functions have some very interesting properties which can be exploited to solve many complicated problems easily. For example, suppose you are searching for an assignment to the input variables of f that would excite f to **1**. In other words, we are searching for ANY ON-SET minterm of f. Suppose, further, that f is +ve unate in x. If f is too large to handle, you can use the Shannon's expansion, and search for the cubes in the cofactors f_x and f'_x. But is it necessary to search in $f_{x'}$? Remember that because of unateness of f w.r.t. x, $f_x \supset f_{x'}$; thus all the cubes in $f_{x'}$ are already in f_x. Furthermore, because f is +ve unate in x, we can ALWAYS find an ON-SET minterm that contains x in TRUE form. This means we can ALWAYS find an ON-SET minterm of f in $x \cdot f_x$. Thus, a search for any ON-SET minterm of f can just ignore the contained co-factor! *Mutatis mutandis* for f -ve unate in x. This is a very important property that can simplify testing/satisfiability. See the example below:

Unfortunately, most Boolean functions that we encounter in life: full adders, multipliers, etc. are not unate. They are highly binate, which means that these properties cannot be exploited to efficiently solve most of these problems. Bad news? Not really! While most functions are binate, the recursive Shannon's expansion on binate functions quickly leads us to *unate cofactors*. So if we cannot exploit unate characteristics directly on a binate function, we can always expand it using Shannon's decomposition untill we find unate cofactors and then solve the problem efficiently. See the example below:

Let f and g be two functions with support variables $\{x_i, i = 1, 2, \ldots, n\}$. Let $\odot$ be an arbitrary binary operator, representing a Boolean function of two arguments. The orthonormal expansion of $f \odot g$ with respect to x_i is given as [4] [3]:

$$f \odot g = x_i(f_{x_i} \odot g_{x_i}) + \overline{x}_i(f_{\overline{x}_i} \odot g_{\overline{x}_i}), \forall i = 1, 2, \ldots, n \tag{2}$$

Definition II.3: The **Boolean difference** or **Boolean derivative** of f with respect to x is $\delta f/\delta x$ or $f_x \oplus f'_x$.

Boolean difference of f w.r.t. x means whether or not f is sensitive to changes in x. When the Boolean difference is zero, it means f does not depend on x. In other words, any change in x does not change f. This is yet another important property that we will utilize time and again when studying Testing.

Definition II.4: The **consensus** or **universal abstraction** of f with respect to x is $f_x \cdot f'_x$. This represents the component of the function independent of x.

Definition II.5: The **smoothing** or **existential abstraction** of f with respect to x is $f_x + f'_x$. This drops the dependency of the function f on the variable x.

In order to understand what the above two really mean, follow the example below:

REFERENCES

[1] F. Hill and G. Peterson, *Introduction to Switching Theory and Digital Design*, John Wiley and Sons, 1981, 1981.
[2] F. Brown, *Boolean Reasoning*, Kluwer Academic Publishers, 1990.
[3] G.D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1996.
[4] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 94.