Boolean Algebra and the Application of Truth Tables

Boolean Algebra applies algebra to an environment where there are only 2 values, true and false. This may also be referred to as Binary Algebra or Logical Algebra. Common applications of Boolean Algebra are logic gates (for work in circuitry), computer science, set theory and statistics.

The concept of Boolean Algebra was first explored by George Boole. He published two books on the subject, *The Mathematical Analysis of Logic (1847)* and then later *An Investigation of the Laws of Thought (1854)*.

Logical statements in Boolean Algebra are often analyzed using a **Truth Table**. A truth table represents all possible outcomes when performing a specific logical operation. Typically, each value to be operated on is represented in column form with the answer being on the right side of the table.

Unary Operations are operators that operate on a single statement. An example would be the NOT operator (One's Complement in Computer Science). Unary operations may be represented by a truth table with 2 columns and 2 rows, where the answer is displayed in the second column.

Binary Operations require two statements to operate. These include AND, OR, XOR, etc. Binary operations are represented by a truth table with 3 columns and 4 rows, where the answer is displayed in the third column.

The symbols used in mathematics to represent logical operations are not always consistent between sources and can be difficult to represent on a computer. Instead, this text will use the word abbreviations as shown below.

AND	Yields true if all conditions are true
OR	Yields true if any one condition is true
XOR (Exclusive or)	Yields true if exactly one of two conditions are true
NOT	Converts true to false and false to true
NOR (Not or)	Yields false if any condition is true
XNOR (Exclusive not or)	EXCLUSIVE NOT OR
NAND (Not and)	Yields true if any condition is false

Figure 1: Common Binary Operations

The composite Unary and Binary Truth Tables below show the answers to the different operators.

	NOT
T	F
F	T

Figure 2: Unary Operations Truth Table

		AND	OR	XOR	NOR	XNOR	NAND
T	T	T	T	F	F	T	F
T	F	F	T	T	F	F	T
F	T	F	T	T	F	F	T
F	F	F	F	F	T	T	T

Figure 3: Binary Operations Truth Table

Applications in Computer Science

Boolean Algebra lends well to concepts in computer science. Computers count using base 2 (binary) notation. The smallest value in computer science is called a bit, which has only two possible values, 0 or 1. 0 is typically false and 1 is true. More precisely any number that is not zero is true. When 4 bits are placed together, this is called a **Nibble** and has 16 discrete values. A **Byte** contains 8 bits and has 256 discrete values. Working with the individual bits is often called bitwise operations.

	Number of placeholders (Bits)	Number of discrete values
Bit	1	2
Nibble	4	16
Byte	8	256
Half Word	Word / 2	???
Word	???	???
Double Word	Word * 2	???

Figure 4: Storage Units in Computer Science

A **Word** is the register size for a particular processor. It will vary in length based on the architecture used. Common values are 8, 16, 32, and 64 bits. To avoid confusion when using the term word, consider prepending it with the size, such as XX-bit word (eg 32-bit word). It is also acceptable to say half word or double word to indicate one half the number of bits or double the number of bits of a word.

Truth tables are also used in computer science. Traditionally, truth tables are displayed with sets of values existing in column form (reading left to right). Often in computer science it is easier to display a concept by representing the truth table in row form with the answer at the bottom, especially for bit shift operations.

Bitwise operations in a program source file are not typically represented with the same symbols used when applying Boolean Algebra to other disciplines. C and many other programing languages do not have a single operator to express NOR, XNOR, or NAND. However, these operators may be represented by combining others in specific patterns.

Bitwise AND	& (Ampersand)
Bitwise Inclusive-OR	| (OEM Pipe)
Bitwise XOR	^ (Carrot)
One's Complement Bitwise Not	~ (tilde)
NOR	~(a | b)
XNOR	~(a ^ b) OR a == b
NAND	~(a & b)
Left Shift	<< X
Right Shift	>> X

Figure 5: Bitwise Operations using C language notation

In addition to the typical Boolean operators, Computer Science also uses a bitwise left shift and bitwise right shift.

Binary Numbers
Binary Numbers are values represented in base-2 notation as opposed to the typical base-10 notation (decimal). Other formats also exist such as octal and hexadecimal. The binary number 0110 represents decimal 6.

As an example, convert the binary number 0101 into decimal form:

4^{th} digit		3^{rd} digit		2^{nd} digit		1^{st} digit	Equals
0000	+	0**1**00	+	00**0**0	+	000**1**	Binary 0101
0 ^ 3 = 0	+	2 ^ 2 = 4	+	0 ^ 1 = 0	+	2 ^ 0 = 1	Decimal 5

Negative Binary Numbers
In computer science, the terms signed or unsigned indicate if a storage unit can hold negative numbers. The term **Unsigned** means only positive numbers are allowed. The term **Signed** means either positive or negative numbers are allowed. Signed and unsigned numbers may both store the number 0.

The most common method for representing a signed number is using **Twos Complement** notation. In twos complement notation, the leftmost bit in the sequence is the sign bit. In this case, if the leftmost bit is 1, the number is negative. If this bit is 0, the number is positive.

If the number is negative, the rest of the number is represented by its complement (opposite value). As an example, the signed byte 1111 1111 would represent the number -1. The high order bit (furthest to the left), represents the sign, which is negative in the case of a 1 bit. Then the remaining 7 bits when taking the complement (~) represent 000 0000 or zero.

As an exercise, what number does 1111 1101 represent when stored in a signed byte, assuming twos complement notation.

1. Check the highest order bit to determine sign (positive/negative). In this case 1XXX XXXX means a negative number.
2. Because the number is negative, perform a Bitwise NOT operation on remaining digits: X111 1101 becomes X000 0010.
3. Convert the binary number into decimal notation, in this case 2 to the first power equals 2.

While twos complement is the most common method, other less common options for representing signed numbers exist including Signed Magnitude and Ones Complement.

A signed value can effective store half the number of positive values as an unsigned bit, allocating space for both negative and positive values.

Storage Type	Unsigned Range	Signed Range
Bit	0 to 1	N/A
Byte	0 to 255	-128 to 127
16-bit Word	0 to 65,536	-32,767 to 32,767
32-bit Word	0 to 4,294,967,296	-2,147,483,648 to 2,147,483,647

Figure 6: Range of values allowed (assuming Twos Complement storage for signed values)

Bitwise Shift Operations
Two bitwise shift operations are available, left shift (<<) and right shift (>>).

Left Shift is represented by << X, where X is the number of placeholders to shift all bits by. 0s are always shifted into the lower order bits. Left shift effectively raises the value of the number by a power of 2 each time it is shifted. (EG << 4 would raise the number by X * 2 ^ 4.)

Right Shift is represented by >> X, where X is the number of placeholders to shift all bits by to the right. Right shift effectively divides the value of the number by a power of 2 each time shifted. (EG >> 3 would lower the number by X / 2 ^ 4). Which bits are shifted into the higher order bits is dependent on whether the number is a signed or unsigned number.

For an unsigned number, a 0 will always be right-shifted in. For a signed number, if the number is positive, 0 bits will be shifted in. For a negative number, the bit shifted in will be machine-dependent with two typical choices, arithmetic right shift and logical right shift.

A **Logical Right Shift** occurs if 0s are shifted in during right shift operations on a negative value. An **Arithmetic Right Shift** occurs if 1s are shifted in during right shift operations on a negative value. Another way to look at it is arithmetic shifts preserve the sign of the number while logical shifts do not.

For an example, perform a 1010 0001 >> 2 (right shift the byte by 2)

1. Arithmetic Right Shift: 1110 1000
2. Logical Right Shift: 0010 1000

Further Reading
- **Boolean Functions:** http://mathworld.wolfram.com/BooleanFunction.html
- **Logic Gates:** https://www.electronics-tutorials.ws/boolean/bool_7.html
- **Bitwise Operations in C:** https://en.wikipedia.org/wiki/Bitwise_operations_in_C
- **Endianness:** https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/