

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 1.4

Logic Operations

This set of notes briefly describes a generalization to truth tables, then introduces Boolean logic operations as well as notational conventions and tools that we use to express general functions on bits. We illustrate how logic operations enable us to express functions such as overflow conditions concisely, then show by construction that a small number of logic operations suffices to describe any operation on any number of bits. We close by discussing a few implications and examples.

Truth Tables

You have seen the basic form of truth tables in the textbook and in class. Over the semester, we will introduce several extensions to the basic concept, mostly with the goal of reducing the amount of writing necessary when using truth tables. For example, the truth table to the right uses two generalizations to show the carry out C (also the unsigned overflow indicator) and the sum S produced by adding two 2-bit unsigned numbers. First, rather than writing each input bit separately, we have grouped pairs of input bits into the numbers A and B. Second, we have defined multiple output columns so as to include both bits of S as well as C in the same table. Finally, we have grouped the two bits of S into one column.

inputs		outputs	
A	B	C	S
00	00	0	00
00	01	0	01
00	10	0	10
00	11	0	11
01	00	0	01
01	01	0	10
01	10	0	11
01	11	1	00
10	00	0	10
10	01	0	11
10	10	1	00
10	11	1	01
11	00	0	11
11	01	1	00
11	10	1	01
11	11	1	10

Keep in mind as you write truth tables that only rarely does an operation correspond to a simple and familiar process such as addition of base 2 numbers. We had to choose the unsigned and 2's complement representations carefully to allow ourselves to take advantage of a familiar process. In general, for each line of a truth table for an operation, you may need to make use of the input representation to identify the input values, calculate the operation's result as a value, and then translate the value back into the correct bit pattern using the output representation. Signed magnitude addition, for example, does not always correspond to base 2 addition: when the signs of the two input operands differ, one should instead use base 2 subtraction. For other operations or representations, base 2 arithmetic may have no relevance at all.

Boolean Logic Operations

In the middle of the 19^{th} century, George Boole introduced a set of logic operations that are today known as **Boolean logic** (also as **Boolean algebra**). These operations today form one of the lowest abstraction levels in digital systems, and an understanding of their meaning and use is critical to the effective development of both hardware and software.

You have probably seen these functions many times already in your education—perhaps first in set-theoretic form as Venn diagrams. However, given the use of common English words *with different meanings* to name some of the functions, and the sometimes confusing associations made even by engineering educators, we want to provide you with a concise set of definitions that generalizes correctly to more than two operands. You may have learned these functions based on truth values (true and false), but we define them based on bits, with 1 representing true and 0 representing false.

Table 1 on the next page lists logic operations. The first column in the table lists the name of each function. The second provides a fairly complete set of the notations that you are likely to encounter for each function, including both the forms used in engineering and those used in mathematics. The third column defines the function's value for two or more input operands (except for NOT, which operates on a single value). The last column shows the form generally used in logic schematics/diagrams and mentions the important features used in distinguishing each function (in pictorial form usually called a **gate**, in reference to common physical implementations) from the others.

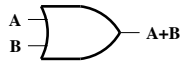
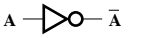
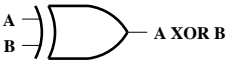
Function	Notation	Explanation	Schematic
AND	A AND B AB $A \cdot B$ $A \times B$ $A \wedge B$	the "all" function: result is 1 iff **all** input operands are equal to 1	 flat input, round output
OR	A OR B $A + B$ $A \vee B$	the "any" function: result is 1 iff **any** input operand is equal to 1	 round input, pointed output
NOT	NOT A A' $\overline{A}$ $\neg A$	logical complement/negation: NOT 0 is 1, and NOT 1 is 0	 triangle and circle
XOR exclusive OR	A XOR B $A \oplus B$	the "odd" function: result is 1 iff an **odd** number of input operands are equal to 1	 OR with two lines on input side
English "or"	A, B, or C	the "one of" function: result is 1 iff exactly **one of** the input operands is equal to 1	(not used)

Table 1: Boolean logic operations, notation, definitions, and symbols.

The first function of importance is **AND**. Think of **AND** as the "all" function: given a set of input values as operands, AND evaluates to 1 if and only if *all* of the input values are 1. The first notation line simply uses the name of the function. In Boolean algebra, AND is typically represented as multiplication, and the middle three forms reflect various ways in which we write multiplication. The last notational variant is from mathematics, where the AND function is formally called **conjunction**.

The next function of importance is **OR**. Think of **OR** as the "any" function: given a set of input values as operands, OR evaluates to 1 if and only if *any* of the input values is 1. The actual number of input values equal to 1 only matters in the sense of whether it is at least one. The notation for OR is organized in the same way as for AND, with the function name at the top, the algebraic variant that we will use in class—in this case addition—in the middle, and the mathematics variant, in this case called **disjunction**, at the bottom.

The definition of Boolean OR is not the same as our use of the word "or" in English. For example, if you are fortunate enough to enjoy a meal on a plane, you might be offered several choices: "Would you like the chicken, the beef, or the vegetarian lasagna today?" Unacceptable answers to this English question include: "Yes," "Chicken and lasagna," and any other combination that involves more than a single choice!

You may have noticed that we might have instead mentioned that AND evaluates to 0 if any input value is 0, and that OR evaluates to 0 if all input values are 0. These relationships reflect a mathematical duality underlying Boolean logic that has important practical value in terms of making it easier for humans to digest complex logic expressions. We will talk more about duality later in the course, but you should learn some of the practical value now: if you are trying to evaluate an AND function, look for an input with value 0; if you are trying to evaluate an OR function, look for an input with value 1. If you find such an input, you know the function's value without calculating any other input values.

We next consider the **logical complement** function, **NOT**. The NOT function is also called **negation**. Unlike our first two functions, NOT accepts only a single operand, and reverses its value, turning 0 into 1 and 1 into 0. The notation follows the same pattern: a version using the function name at the top, followed by two variants used in Boolean algebra, and finally the version frequently used in mathematics. For the NOT gate, or **inverter**, the circle is actually the important part: the triangle by itself merely copies the input. You will see the small circle added to other gates on both inputs and outputs; in both cases the circle implies a NOT function.

Last among the Boolean logic functions, we have the **XOR**, or **exclusive OR** function. Think of XOR as the "odd" function: given a set of input values as operands, XOR evaluates to 1 if and only if *an odd number* of the input values are 1. Only two variants of XOR notation are given: the first using the function name, and the second used with Boolean algebra. Mathematics rarely uses this function.

Finally, we have included the meaning of the word "or" in English as a separate function entry to enable you to compare that meaning with the Boolean logic functions easily. Note that many people refer to English' use of the word "or" as "exclusive" because one true value excludes all others from being true. Do not let this human language ambiguity confuse you about XOR! For all logic design purposes, *XOR is the odd function.*

The truth table to the right provides values illustrating these functions operating on three inputs. The AND, OR, and XOR functions are all associative—$(A \text{ op } B) \text{ op } C = A \text{ op } (B \text{ op } C)$—and commutative—$A \text{ op } B = B \text{ op } A$, as you may have already realized from their definitions.

inputs			outputs			
A	B	C	ABC	$A+B+C$	$\overline{A}$	$A \oplus B \oplus C$
0	0	0	0	0	1	0
0	0	1	0	1	1	1
0	1	0	0	1	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	0	0
1	1	0	0	1	0	0
1	1	1	1	1	0	1

Overflow as Logic Expressions

In the last set of notes, we discussed overflow conditions for unsigned and 2's complement representations. Let's use Boolean logic to express these conditions.

We begin with addition of two 1-bit unsigned numbers. Call the two input bits A_0 and B_0. If you write a truth table for this operation, you'll notice that overflow occurs only when all (two) bits are 1. If either bit is 0, the sum can't exceed 1, so overflow cannot occur. In other words, overflow in this case can be written using an AND operation:

$$A_0 B_0$$

The truth table for adding two 2-bit unsigned numbers is four times as large, and seeing the structure may be difficult. One way of writing the expression for overflow of 2-bit unsigned addition is as follows:

$$A_1 B_1 + (A_1 + B_1) A_0 B_0$$

This expression is slightly trickier to understand. Think about the place value of the bits. If both of the most significant bits—those with place value 2—are 1, we have an overflow, just as in the case of 1-bit addition. The $A_1 B_1$ term represents this case. We also have an overflow if one or both (the OR) of the most significant bits are 1 and the sum of the two next significant bits—in this case those with place value 1—generates a carry.

The truth table for adding two 3-bit unsigned numbers is probably not something that you want to write out. Fortunately, a pattern should start to become clear with the following expression:

$$A_2 B_2 + (A_2 + B_2) A_1 B_1 + (A_2 + B_2)(A_1 + B_1) A_0 B_0$$

In the 2-bit case, we mentioned the "most significant bit" and the "next most significant bit" to help you see the pattern. The same reasoning describes the first two product terms in our overflow expression for 3-bit unsigned addition (but the place values are 4 for the most significant bit and 2 for the next most significant bit). The last term represents the overflow case in which the two least significant bits generate a carry which then propagates up through all of the other bits because at least one of the two bits in every position is a 1.

The overflow condition for addition of two N-bit 2's complement numbers can be written fairly concisely in terms of the first bits of the two numbers and the first bit of the sum. Recall that overflow in this case depends only on whether the three numbers are negative or non-negative, which is given by the most significant bit. Given the bit names as shown to the right, we can write the overflow condition as follows:

$$\begin{array}{rr} & A_{N-1}A_{N-2}\ldots A_2A_1A_0 \\ + & B_{N-1}B_{N-2}\ldots B_2B_1B_0 \\ \hline & S_{N-1}\,S_{N-2}\ldots S_2\,S_1\,S_0 \end{array}$$

$$A_{N-1}\ B_{N-1}\ \overline{S_{N-1}} + \overline{A_{N-1}}\ \overline{B_{N-1}}\ S_{N-1}$$

The overflow condition does of course depend on all of the bits in the two numbers being added. In the expression above, we have simplified the form by using S_{N-1}. But S_{N-1} depends on the bits A_{N-1} and B_{N-1} as well as the carry out of bit $(N-2)$.

Later in this set of notes, we present a technique with which you can derive an expression for an arbitrary Boolean logic function. As an exercise, after you have finished reading these notes, try using that technique to derive an overflow expression for addition of two N-bit 2's complement numbers based on A_{N-1}, B_{N-1}, and the carry out of bit $(N-2)$ (and into bit $(N-1)$), which we might call C_{N-1}. You might then calculate C_{N-1} in terms of the rest of the bits of A and B using the expressions for unsigned overflow just discussed. In the next month or so, you will learn how to derive more compact expressions yourself from truth tables or other representations of Boolean logic functions.

Logical Completeness

Why do we feel that such a short list of functions is enough? If you think about the number of possible functions on N bits, you might think that we need many more functions to be able to manipulate bits. With 10 bits, for example, there are 2^{1024} such functions. Obviously, some of them have never been used in any computer system, but maybe we should define at least a few more logic operations? In fact, we do not even need XOR. The functions AND, OR, and NOT are sufficient, even if we only allow two input operands for AND and OR!

The theorem below captures this idea, called **logical completeness**. In this case, we claim that the set of functions {AND, OR, NOT} is sufficient to express any operation on any finite number of variables, where each variable is a bit.

Theorem: Given enough 2-input AND, 2-input OR, and 1-input NOT functions, one can express any Boolean logic function on any finite number of variables.

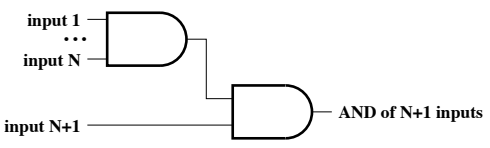
The proof of our theorem is **by construction**. In other words, we show a systematic approach for transforming an arbitrary Boolean logic function on an arbitrary number of variables into a form that uses only AND, OR, and NOT functions on one or two operands. As a first step, we remove the restriction on the number of inputs for the AND and OR functions. For this purpose, we state and prove two **lemmas**, which are simpler theorems used to support the proof of a main theorem.

Lemma 1: Given enough 2-input AND functions, one can express an AND function on any finite number of variables.

Proof: We prove the Lemma **by induction**.[1] Denote the number of inputs to a particular AND function by N.

The base case is $N = 2$. Such an AND function is given.

To complete the proof, we need only show that, given any number of AND functions with up to N inputs, we can express an AND function with $N+1$ inputs. To do so, we need merely use one 2-input AND function to join together the result of an N-input AND function with an additional input, as illustrated to the right.



[1] We assume that you have seen proof by induction previously.

Lemma 2: Given enough 2-input OR functions, one can express an OR function on any finite number of variables.

Proof: The proof of Lemma 2 is identical in structure to that of Lemma 1, but uses OR functions instead of AND functions.

Let's now consider a small subset of functions on N variables. For any such function, you can write out the truth table for the function. The output of a logic function is just a bit, either a 0 or a 1. Let's consider the set of functions on N variables that produce a 1 for exactly one combination of the N variables. In other words, if you were to write out the truth table for such a function, exactly one row in the truth table would have output value 1, while all other rows had output value 0.

Lemma 3: Given enough AND functions and 1-input NOT functions, one can express any Boolean logic function that produces a 1 for exactly one combination of any finite number of variables.

Proof: The proof of Lemma 3 is by construction. Let N be the number of variables on which the function operates. We construct a **minterm** on these N variables, which is an AND operation on each variable or its complement. The minterm is specified by looking at the unique combination of variable values that produces a 1 result for the function. Each variable that must be a 1 is included as itself, while each variable that must be a 0 is included as the variable's complement (using a NOT function). The resulting minterm produces the desired function exactly. When the variables all match the values for which the function should produce 1, the inputs to the AND function are all 1, and the function produces 1. When any variable does not match the value for which the function should produce 1, that variable (or its complement) acts as a 0 input to the AND function, and the function produces a 0, as desired.

The table below shows all eight minterms for three variables.

inputs			outputs							
A	B	C	$\overline{A}\,\overline{B}\,\overline{C}$	$\overline{A}\,\overline{B}\,C$	$\overline{A}\,B\,\overline{C}$	$\overline{A}\,B\,C$	$A\,\overline{B}\,\overline{C}$	$A\,\overline{B}\,C$	$A\,B\,\overline{C}$	$A\,B\,C$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

We are now ready to prove our theorem.

Proof (of Theorem): Any given function on N variables produces the value 1 for some set of combinations of inputs. Let's say that M such combinations produce 1. Note that $M \leq 2^N$. For each combination that produces 1, we can use Lemma 1 to construct an N-input AND function. Then, using Lemma 3, we can use as many as M NOT functions and the N-input AND function to construct a minterm for that input combination. Finally, using Lemma 2, we can construct an M-input OR function and OR together all of the minterms. The result of the OR is the desired function. If the function should produce a 1 for some combination of inputs, that combination's minterm provides a 1 input to the OR, which in turn produces a 1. If a combination should produce a 0, its minterm does not appear in the OR; all other minterms produce 0 for that combination, and thus all inputs to the OR are 0 in such cases, and the OR produces 0, as desired.

The construction that we used to prove logical completeness does not necessarily help with efficient design of logic functions. Think about some of the expressions that we discussed earlier in these notes for overflow conditions. How many minterms do you need for N-bit unsigned overflow? A single Boolean logic function can be expressed in many different ways, and learning how to develop an efficient implementation of a function as well as how to determine whether two logic expressions are identical without actually writing out truth tables are important engineering skills that you will start to learn in the coming months.

Implications of Logical Completeness

If logical completeness doesn't really help us to engineer logic functions, why is the idea important? Think back to the layers of abstraction and the implementation of bits from the first couple of lectures. Voltages are real numbers, not bits. *The device layer implementations of Boolean logic functions must abstract away the analog properties of the physical system.* Without such abstraction, we must think carefully about analog issues such as noise every time we make use of a bit! Logical completeness assures us that no matter what we want to do with bits, implementating a handful of operations correctly is enough to guarantee that we never have to worry.

A second important value of logical completeness is as a tool in screening potential new technologies for computers. If a new technology does not allow implementation of a logically complete set of functions, the new technology is extremely unlikely to be successful in replacing the current one.

That said, {AND, OR, and NOT} is not the only logically complete set of functions. In fact, our current complementary metal-oxide semiconductor (CMOS) technology, on which most of the computer industry is now built, does not directly implement these functions, as you will see later in our class.

The functions that are implemented directly in CMOS are NAND and NOR, which are abbreviations for AND followed by NOT and OR followed by NOT, respectively. Truth tables for the two are shown to the right.

inputs		outputs	
		$\overline{AB}$	$\overline{A+B}$
A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Either of these functions by itself forms a logically complete set. That is, both the set {NAND} and the set {NOR} are logically complete. For now, we leave the proof of this claim to you. Remember that all you need to show is that you can implement any set known to be logically complete, so in order to prove that {NAND} is logically complete (for example), you need only show that you can implement AND, OR, and NOT using only NAND.

Examples and a Generalization

Let's use our construction to solve a few examples. We begin with the functions that we illustrated with the first truth table from this set of notes, the carry out C and sum S of two 2-bit unsigned numbers. Since each output bit requires a separate expression, we now write S_1S_0 for the two bits of the sum. We also need to be able to make use of the individual bits of the input values, so we write these as A_1A_0 and B_1B_0, as shown on the left below. Using our construction from the logical completeness theorem, we obtain the equations on the right. You should verify these expressions yourself.

inputs				outputs		
A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

$$C = \overline{A_1}\ A_0\ B_1\ B_0 + A_1\ \overline{A_0}\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ B_1\ B_0 + A_1\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ \overline{B_0} + A_1\ A_0\ B_1\ B_0$$

$$S_1 = \overline{A_1}\ \overline{A_0}\ B_1\ \overline{B_0} + \overline{A_1}\ \overline{A_0}\ B_1\ B_0 + \overline{A_1}\ A_0\ \overline{B_1}\ B_0 + \overline{A_1}\ A_0\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1}\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1}\ B_0 + A_1\ A_0\ \overline{B_1}\ \overline{B_0} + A_1\ A_0\ B_1\ B_0$$

$$S_0 = \overline{A_1}\ \overline{A_0}\ \overline{B_1}\ B_0 + \overline{A_1}\ \overline{A_0}\ B_1\ B_0 + \overline{A_1}\ A_0\ \overline{B_1}\ \overline{B_0} + \overline{A_1}\ A_0\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1}\ B_0 + A_1\ \overline{A_0}\ B_1\ B_0 + A_1\ A_0\ \overline{B_1}\ \overline{B_0} + A_1\ A_0\ B_1\ \overline{B_0}$$

Now let's consider a new function. Given an 8-bit 2's complement number, $A = A_7A_6A_5A_4A_3A_2A_1A_0$, we want to compare it with the value -1. We know that we can construct this function using AND, OR, and NOT, but how? We start by writing the representation for -1, which is 11111111. If the number A matches that representation, we want to produce a 1. If the number A differs in any bit, we want to produce a 0. The desired function has exactly one combination of inputs that produces a 1, so in fact we need only one minterm! In this case, we can compare with -1 by calculating the expression:

$$A_7 \cdot A_6 \cdot A_5 \cdot A_4 \cdot A_3 \cdot A_2 \cdot A_1 \cdot A_0$$

Here we have explicitly included multiplication symbols to avoid confusion with our notation for groups of bits, as we used when naming the individual bits of A.

In closing, we briefly introduce a generalization of logic operations to groups of bits. Our representations for integers, real numbers, and characters from human languages all use more than one bit to represent a given value. When we use computers, we often make use of multiple bits in groups in this way. A **byte**, for example, today means an ordered group of eight bits. We can extend our logic functions to operate on such groups by pairing bits from each of two groups and performing the logic operation on each pair. For example, given $A = A_7A_6A_5A_4A_3A_2A_1A_0 = 01010101$ and $B = B_7B_6B_5B_4B_3B_2B_1B_0 = 11110000$, we calculate A AND B by computing the AND of each pair of bits, A_7 AND B_7, A_6 AND B_6, and so forth, to produce the result 01010000, as shown to the right. In the same way, we can extend other logic operations, such as OR, NOT, and XOR, to operate on bits of groups.

$$\begin{array}{rr} & A\ 01010101 \\ \text{AND} & B\ 11110000 \\ \hline & 01010000 \end{array}$$