In lectures 1 and 2, we looked at representations of numbers. For the case of integers, we saw that we could perform addition of two numbers using a binary representation and using the same algorithm that you used in grade school. I also argued that if you represent negative numbers using twos complement, then you can do addition with negative numbers as well. In lecture 4, we will see how to implement the addition algorithm using circuits. Before we can do so, we need to review some basic logic and use it to build up circuits.

Truth tables

Let A and B be two binary valued variables, that is, A, B each can take values in $\{0, 1\}$. If we associate the value 0 with FALSE and the value 1 with TRUE, then we can make logical expressions using such binary valued variables. We make such expressions using the binary operators AND, OR, and the unary operator NOT. Let $A \cdot B$ denote AND(A, B). Let $A + B$ denote $OR(A, B)$. Let $\overline{A}$ denotes NOT(A). The AND and OR operators are defined by the following truth table.

A	B	$A \cdot B$	$A + B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

The unary operator NOT is defined by the truth table:

A	$\overline{A}$
0	1
1	0

Since a truth table of two input variables has four rows, it follows that there are $2^4 = 16$ possible output functions (that is, possible columns in the truth table). The operators AND and OR are just two of these 16 possible output functions. Three other output functions that are commonly used are NAND (not and), NOR (not or), and XOR (exclusive or). These are defined as follows.

A	B	$A \cdot B$	$A + B$	$\overline{A \cdot B}$	$\overline{A + B}$	$A \oplus B$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

To justify why the symbols "+" and "·" are used to represent OR and AND operators, I would have to spend some time talking about *boolean algebra*. This might be of interest to some of you, but it won't take us in the direction we want to go in this course. Instead, I will give you a brief introduction to the topic and then focus on *how to use* the logical expressions to build up digital circuits in a computer.

Laws of Boolean Algebra

identity	$A + 0 = A$	$A \cdot 1 = A$
inverse	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
one and zero	$A + 1 = 1$	$A \cdot 0 = 0$
commutative	$A + B = B + A,$	$A \cdot B = B \cdot A$
associative	$(A + B) + C = A + (B + C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
distributive law:	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$(A \cdot B) + C = (A + C) \cdot (B + C)$
De Morgan	$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$

These laws capture the logical reasoning that you carry out in your head when you evaluate the truth of expressions formed using OR, NOT, AND. But they are more than this. The laws also give a set of rules for automatically evaluating and re-writing such expressions. For example, the commutative law allows you to swap the order of two terms; De Morgan's Laws allows you to swap the order of the NOT with either of the OR or AND operators, etc.

I encourage you to memorize the names of the laws. Although I will not examine you on this, these names are not just used in Boolean Algebra; they are used in Algebra in many other situations. At the very least, you should convince yourself that you *already* know the laws of Boolean Algebra since you use them everyday in reasoning about the world. However, you should understand what each of the laws says, and you should make sure that you agree with each of the laws. *In particular, note that most of these laws correspond to the rule so addition and multiplication with numbers, but not all do. The second distributive law generally does not.*

Example

On the last page I wrote truth tables for basic binary and unary operators. We can also write truth tables for expressions that are built out of these operators. Here is an example:

$$Y \;=\; \overline{A \cdot B \cdot C} \cdot (A \cdot B + A \cdot C)$$

The last column $\overline{Y}$ not necessary here, but I'll discuss it on the next page.

A	B	C	$A \cdot B \cdot C$	$\overline{A \cdot B \cdot C}$	$A \cdot B$	$A \cdot C$	$A \cdot B + A \cdot C$	Y	$\overline{Y}$
0	0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	0	0	1
0	1	0	0	1	0	0	0	0	1
0	1	1	0	1	0	0	0	0	1
1	0	0	0	1	0	0	0	0	1
1	0	1	0	1	0	1	1	1	0
1	1	0	0	1	1	0	1	1	0
1	1	1	1	0	1	1	1	0	1

Sum-of-products and product-of-sums (two level logic)

Logical expressions can get very complicated. If a logical expression has n different variables then there are 2^n combinations of values of these variables, and hence 2^n rows in the the truth table. However, once the expression is evaluated, there are simple ways to rewrite the expression, as we show next.

Suppose that Y were some horribly long expression with the A, B, C variables, and that we computed the values of Y for the various combinations of A, B, C. We think of Y as being the *output* and A, B, C as being the *input* variables. We can write Y in two simple ways. The first is called a *sum of products*. For the example on the previous page:

$$Y = A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}$$

since "$\cdot$" is a product and "+" is a sum. The two terms correspond to the two 1's in the Y column of the above table.

The second is called a *product-of-sums*. To compute the product-of-sums, we use a trick. First, we write $\overline{Y}$ as a sum-of-products. For the example on the previous page, see the rightmost column:

$$\overline{Y} = \overline{A} \cdot \overline{B} \cdot \overline{C} \; + \; \overline{A} \cdot \overline{B} \cdot C \; + \; \overline{A} \cdot B \cdot \overline{C} \; + \; \overline{A} \cdot B \cdot C \; + \; A \cdot \overline{B} \cdot \overline{C} \; + \; A \cdot B \cdot C$$

and then we negate both sides:

$$\overline{\overline{Y}} = (\overline{\overline{A} \cdot \overline{B} \cdot \overline{C}}) \cdot (\overline{\overline{A} \cdot \overline{B} \cdot C}) \cdot (\overline{\overline{A} \cdot B \cdot \overline{C}}) \cdot (\overline{\overline{A} \cdot B \cdot C}) \cdot (\overline{A \cdot \overline{B} \cdot \overline{C}}) \cdot (\overline{A \cdot B \cdot C})$$

We then apply De Morgan's laws within each term. Here you must convince yourself that de Morgan's laws hold for n variables, not just two variables. In the case below, we have three variables.

$$Y = (A + B + C) \cdot (A + B + \overline{C}) \cdot (A + \overline{B} + C) \cdot (A + \overline{B} + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{B} + \overline{C})$$

If we have n input variables, then writing our output as a sum-of-products or product-of-sums might involve as many as 2^n terms, one for each row. However, it has the advantage that it only involves two levels of binary operations (first OR then AND, or vice-versa).

Don't cares

If we have m input variables and n output variables, then the truth table has 2^m rows. However, sometimes we don't need to write all the rows because certain combinations of the inputs give the same output. For example, take

$$Y = A \cdot \overline{B} \cdot C + \overline{A}$$

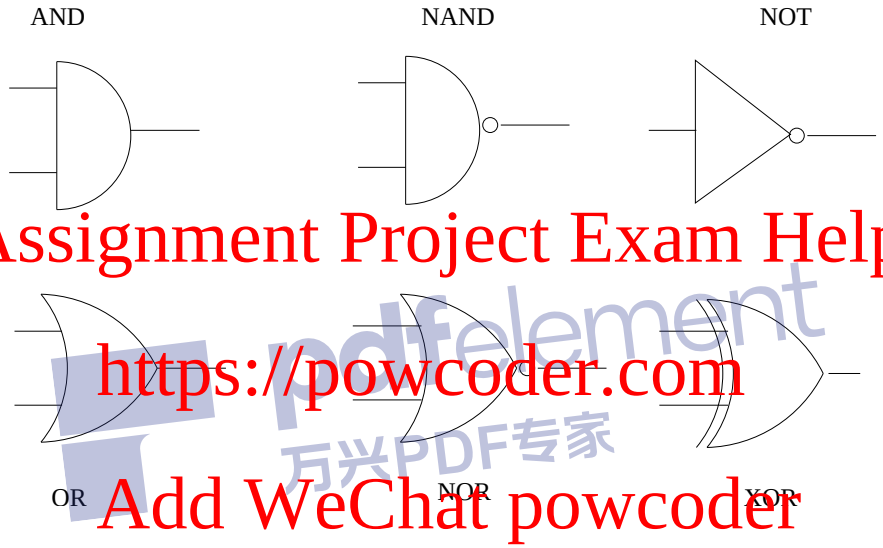
The second expression is really the sum of four expressions which have all combinations of B and C. Alternatively, the second expression can be thought of as saying $\overline{A}$ and "I don't care what B and C are". We denote such "don't care's" in the truth table with x symbols.

A	B	C	Y
0	x	x	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Logic Gates

Computers solve for expressions "$Y = ...$" using electronic circuits, called logic gates, which are composed of resistors and *transistors,* and other elements. Transistors are basic circuits typically made from silicon and other elements. (To understand how they work, you would need to spend a few semesters in the physics department.) Transistors were invented in the 1940s at Bell Labs, and the inventors won the Nobel Prize for it in 1956.

We will not discuss how transistors work. Rather we will work at the *gate level.* A gate implements one of the binary operators defined above, or the unary operation NOT. Examples of the gates we will use are shown below. These are standard symbols and you will need to memorize them.



The inputs and outputs to gates are wires which have one of two voltages (often called low and high, or 0 and 1). Often we will put more than two inputs into an AND or OR gate. This is allowed because of the associative law of Boolean algebra. (The underlying physical circuit would have to be changed, but we don't concern ourselves with this lower level.) Also note that the commutative law of Boolean algebra says that the ordering of the wires into a gate doesn't matter.

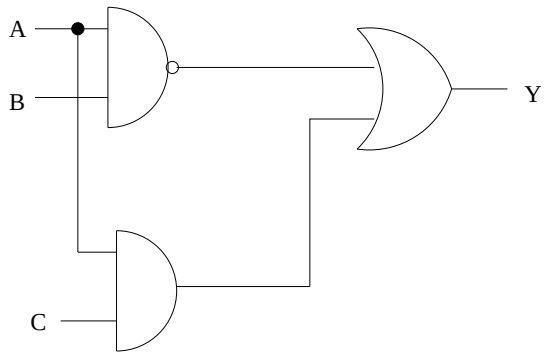
Combinational (or Combinatorial) Circuits

Example 1

Consider a circuit that computes the value of the following expression, where A, B, C are three input variables and Y is the output variable:

$$Y = \overline{A \cdot B} + A \cdot C$$

Whatever binary values are given to the input variables, the output Y will correspond to the truth value of the logical expression.



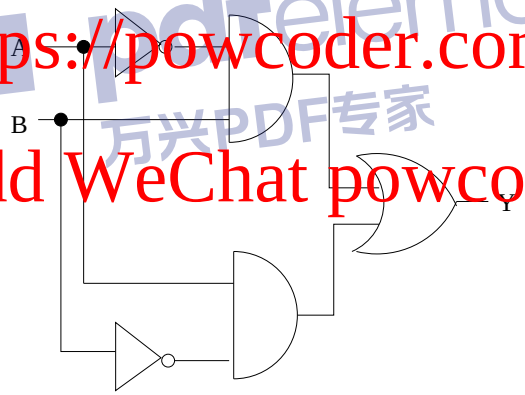
The black dot in this figure indicates that input wire A branches into two input wires.

Example 2

We can write the XOR gate using a sum-of-products

$$Y = A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

and the circuit is shown below. (We could have also used a product-of-sums.)



Example 3

Consider a circuit,

$$Y = \overline{S} \cdot A + S \cdot B$$

This circuit is called a multiplexor or selector. What does this circuit do? If S is true, then Y is given the value B whereas if S is false then Y is given the value A. Such a circuit implements the familiar statement:

```
if   S
then Y := B
else Y := A
```