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 "$\cdot$" 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.

The 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. At the very least, you should convince yourself that you *already* know the laws (since you use them 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.

Example

Write a truth table for following expression:

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

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*:

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

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 as follows. (See the rightmost column of the above table.)

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

and then we negate both sides and apply De Morgan's laws, which gives

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

and applying De Morgan's laws to each term on the right hand side giving

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

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

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

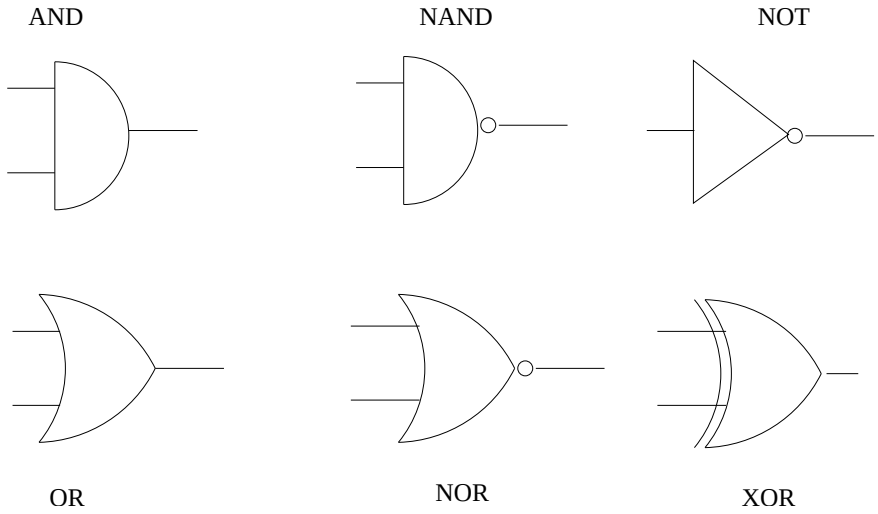
The second expression is really the sum of four expressions which have all combinations of B and C. Alternatively, the second expression can be thought of as saying $\overline{A}$ and "I don't care what B and C are". We denote such "don't care's" in the truth table with x symbols.

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 expressions such as above using electronic circuits, composed of resistors and *transistors,* and other elements. Transistors are simple circuits typically made from silicon and other elements. Transistors are ON/OFF switches, but unlike the mechanical switches you are used to, these switches are turned ON vs. OFF by electricity. That is, one current is used to turn another current ON or OFF. 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 in this course. We will stop one level higher, at the *gate level*. Gates are circuits make out of transitors. 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.



The inputs and outputs to gates are wires. These voltages can have one of two values (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.

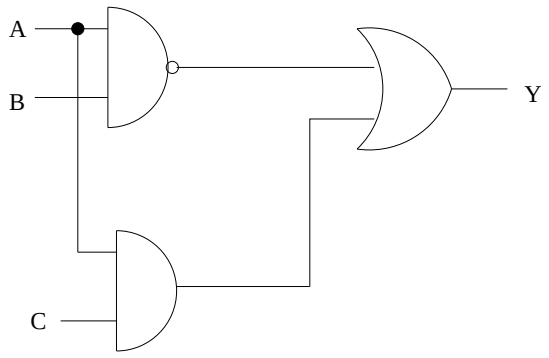
Combinational Circuits

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 put on the input wires to the circuit, 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.

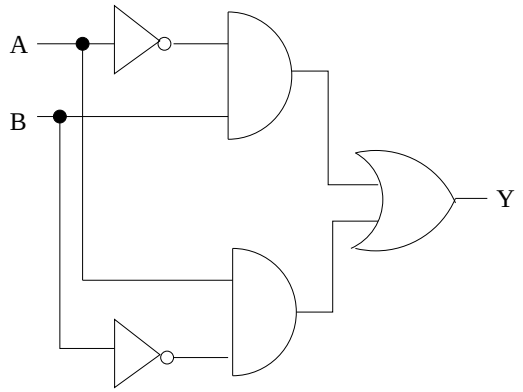


Example 2

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.) Indeed, sums of products can be used to define any of the 16 possible logical functions of two variables A and B.



Example 3

Consider a circuit,

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

This circuit is called a multiplexor. 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
```

Read-only memory (ROM) using combinational logic circuits)

The truth tables are defined by "input variables" and "output variables", and we have been thinking of them as evaluating logical expressions. Another way to think of a combinational circuit is as a *Read Only Memory* (ROM). The inputs encode a memory address. The outputs encode the value stored at the address. We say such a memory is read-only because the gates of the circuit are fixed.

For example, suppose the memory address is specified by the two input variables A_1, A_0, and the contents at each address are specified by the bits $Y_2Y_1Y_0$. Here is a truth table:

A_1	A_0	Y_2	Y_1	Y_0
0	0	0	1	1
0	1	0	0	1
1	0	0	0	0
1	1	1	0	0

and here is the corresponding memory. (Note that the address is not stored in the memory!)

input (address)	output (contents of memory)
00	011
01	001
10	000
11	100

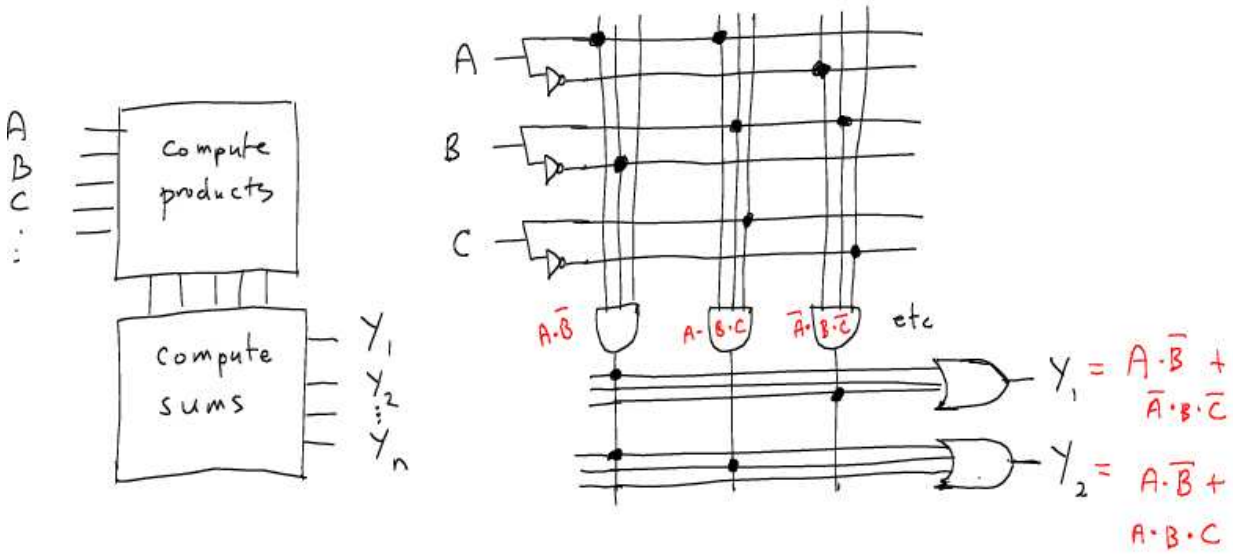
What is new about this concept? You now have to think of the input and output variables in a different way than you did before. Before, you thought of the input variables as having TRUE or FALSE values e.g. you did not associate any particular significance to their order. Thinking about the circuit as a ROM is quite different. You no longer think of each of the input variables as TRUE or FALSE. Rather, you think of the input variables as defining a binary number, namely an address. The order of the input variables now has a significance, since it defines an ordering on the address space. Similarly, the output variables are a string of bits whose order may matter: it might represent a number.

Programmable Logic Array (PLA)

One interesting electronic device that is based on a sum-of-products is the *programmable logic array* (PLA). Such a device that has a predetermined number of input variables, say 12 $\{A_0, A_2, ...A_{11}\}$, and a predetermined number of output variables, say 10 $\{Y_0, Y_2, ...Y_9\}$.

Suppose you are a company making some device whose electronics requires that it compute several (say up to 10) logical expressions, each of which have several (say up to 12) variables. The device might need to evaluate these expressions millions of times. You could design and build your own circuits to solve your problem. Or, you could have another company (say Texas Instruments) produce such circuits for you. What do they do? Rather than building this specialized circuit for you, they take a general circuit called a *progammable logic array*, and they modify it. Here's how.

First, they translate your expressions into sum-of-products expressions. In general, with 3 input variables, there will be a maximum of $2^3 = 8$ terms in the sum(-of-products) for each variable for each output variable Y_i.



Then, they take an intact circuit which consists of two parts. The top part has a set of rows (wires) which carry variables $A, \overline{A}, B, \overline{B}, etc.$ There are also a set of vertical wires that carry product terms. Initially, all rows are connected to all columns, so there are black dots at each intersection. In order to make the circuit compute the product terms that we'll need, many of the black dots must be removed. (Intuitively, what happens here is that certain wires are cut. Technically, one burns out a fuse.)

If there are m input variables, then the columns are grouped into m-tuples. Each m tuple carries one variable. (Each product term has at most m variables.) The m-tuples (or less than m) are fed into AND gates i.e. to form the products you need.

Below the AND gates, the products are appropriately OR-ed together to give the sums we want. See figure. Again, originally all rows are connected to all columns below the AND gates, and so in the figure all intersections would have black dots. To compute the sums of products we want, some of the black dots need to be removed i.e. wires need to be cut (or fuses broken).

Companies like Texas Instruments and National Semiconductor manufacture such circuits. They mass produce the . A user just needs to specify what expression he/she wants to compute. This can be programmed relatively cheaply by cutting out wires. Hence, the term *programmable logic array*. The "array" aspect comes about when you have multiple output variables. In this case, the circuit can be packed nicely as a 2D array.