CS 2742 (Logic in Computer Science)
Lecture 10

Antonina Kolokolova

January 28, 2013

3.1 Complete set of connectives

From what we have done it is easy to see that any possible truth table (we will start saying "boolean function" soon) can be represented by a formula written with just $\wedge, \vee$ and $\neg$. Such set of connectives is called *complete.*

A stronger result tells us that even $\vee$ is not necessary in this set (homework exercise). But $\neg$ is necessary, because every formula without $\neg$ is true when all of its variables are set to true. So contradiction cannot be described by such a formula, and any truth table with a 0 in the last column when all of its inputs are true cannot be represented by a formula using only $\vee$ and $\wedge$.

We have seen that there are two connectives that allow us to construct formulas representing all possible truth tables. Is there a single connective that can be used to construct formulas with all possible truth tables? The answer is yes, and such connective is called NAND (stands for "not-and") in digital circuit design, or Sheffer's stroke (written as $|$). It is defined so $p|q$ is false when both p and q are true, and true otherwise.

p	q	$p\|q$
1	1	0
1	0	1
0	1	1
0	0	1

You can see here that it has values exactly opposite of the values of $p \wedge q$, hence the name NAND. To show that it is complete, let us look at how to simulate $\neg$ and $\wedge$ with it. Once we have both $\neg$ and $\wedge$, we can rewrite any formula to an equivalent one which only uses $|$ as a connective using ideas from the problem in your homework assignment.

To show how to represent $\neg p$, note that the first row of the truth table has 0 for the value of the Sheffer's stoke, and the last row has a 1. So $\neg p \iff p|p$. Similarly, $p \wedge q \iff \neg(p|q) \iff (p|q)|(p|q)$, since Sheffer's stroke is the negation of $\wedge$ and we just showed how to do $\neg$.

4 Boolean functions and circuits

The propositional logic is a special case of Boolean algebra. In Boolean algebra 0 corresponds to false (F), 1 to true (T), + is $\vee$ and $\cdot$ it $\wedge$; here, $\neg$ corresponds to a unary $-$ of arithmetic, and $\neg a$ is written as $\bar{a}$. For this to be a proper Boolean algebra, the identities such as commutativity, associativity, distributivity, identity have to hold for + and $\cdot$. We will skip the more general definition, and say for now that the rules are that $\bar{0} = 1, \bar{1} = 0$, $0 + a = a, a + 1 = 1, 0 \cdot a = 0$ and $1 \cdot a = a$ (note that there are more general Boolean algebras which we will define later). Note also that logic identities hold for Boolean algebras as well, and in fact can be derived from the axioms of Boolean algebra.

Just as arithmetic functions take as arguments some list of numbers often represented by variables, and output a number as an answer, Boolean function on n variables takes n inputs which have values 0 or 1, and produces a 0 or a 1 as an output. It is easy to see that, with that notational substitution, Boolean algebra and propositional logic are very much related: a propositional formula represents a boolean function when the formula is true on its variables iff the function on the corresponding values of its arguments outputs 1.

The easiest way to describe a Boolean function is to give its truth table (from which a CNF or DNF formula representing this function can be constructed). A boolean function is fully described by its truth table: there can be several formulas descibing the same function,but they must be logically equivalent. Usually when writing a truth table for a Boolean function we write 0s and 1s rather than Fs and Ts. Although we often can describe a function by a formula much smaller than its truth table, there are functions that cannot be described by anything smaller than the table itself. You will see a proof of this later if you take an advanced course on theory of computation.

Example 1. Consider a Boolean function $Majority(x, y, z)$ which outputs 1 (true) if at least two of its inputs x, y, z are 1s. It has the following truth table:

x	y	z	$Majority(x, y, z)$
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	0

This can be generalized by defining a Boolean function $Majority(x_1, x_2, \ldots, x_n)$ for every value of n; such a function outputs 1 if more than half of its inputs are 1. In that case, for every n there would be a different truth table describing the function.

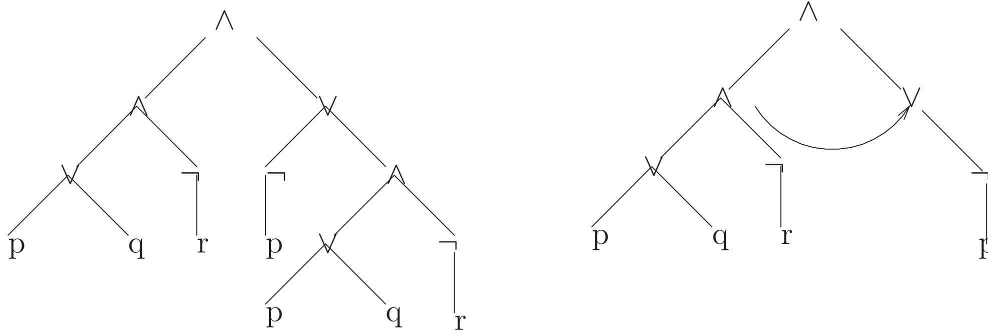
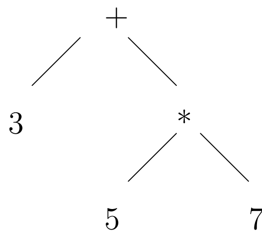


Figure 1: $(p \vee q) \wedge \neg r) \wedge (\neg p \vee ((p \vee q) \wedge \neg r))$. The first is the parse tree for the formula and the second is a corresponding circuit.

4.1 Formulas and circuits

It is often useful to draw a formula (or an arithmetic expression) as a diagram to show the order of evaluation of its parts. For example, the following diagram shows how to evaluate $3 + 5 * 7$:

Example 2.



Essentially this diagram tells us in which order to evaluate parts of the original arithmetic expression. Here, first we (recursively) have to evaluate the expresions on both sides of the $+$, and then apply the arithmetic operation $+$ to the two results.
Similarly, we can have a diagram that shows the order of evaluation of a propositional formula (just change numbers to propositional variables and $+, *$ to $\vee, \wedge, \neg$.)

We call this representation a *tree* or *parse tree* of an arithmetic expression. To construct a parse tree of a formula of the form $A \circ B$, where A and B are formulas themselves and $\circ$ is $\wedge, \vee, \rightarrow$ or $\leftrightarrow$, put the symbol for $\circ$ on top, and the trees for A and B under it; connect $\circ$ to the top symbols of trees for A and B. If the connective is $\neg$, put only one line to the tree for the expression being negated. When run out of connectives,put variables. To evaluate a formula like this, start by evaluating, for every connective, its subformulas and then applying the connective to the result

Example 3. Consider a propositional formula $(p \vee q) \wedge \neg r) \wedge (\neg p \vee ((p \vee q) \wedge \neg r))$. On inputs $(0, 1, 1)$ (that is, $p = 0, q = 1, r = 1$) this formula evaluates to 0: $(p \vee q)$ is 1, $((p \vee q) \wedge \neg r)$ is 0, the same expression in the second part is also 0, and $\neg p \vee \ldots$ is 1, but because the left side of the $\wedge$ was 0 the whole formula evaluates to 0.

Note that in this formula there is a redundancy: there are two identical subformulas (subtrees) computing $(p \vee q) \wedge \neg r$. We can combine them to obtain a *circuit.* A circuit has an additional property that computed values of subformulas can be reused without recomputation. This often makes circuits much more concise than formulas.

Figure 2: Types of gates in a digital circuit.

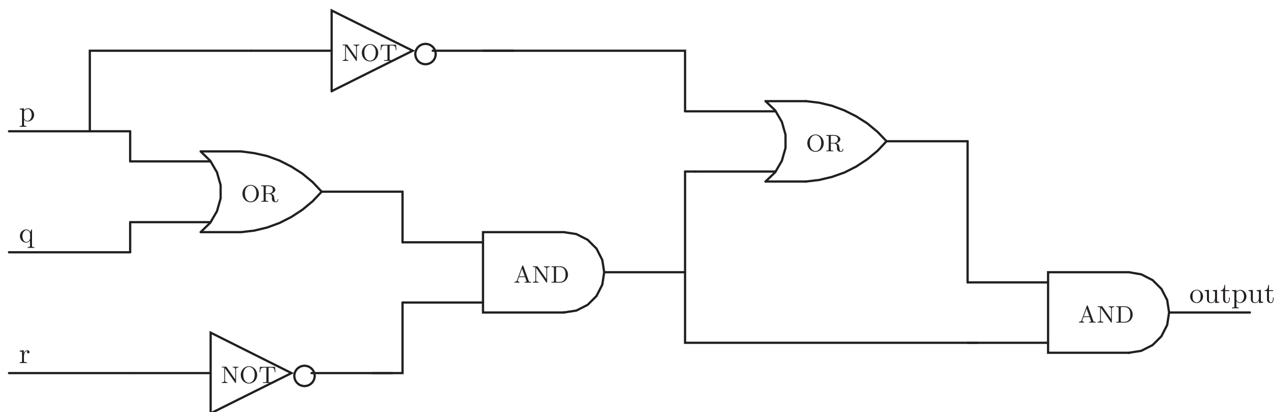
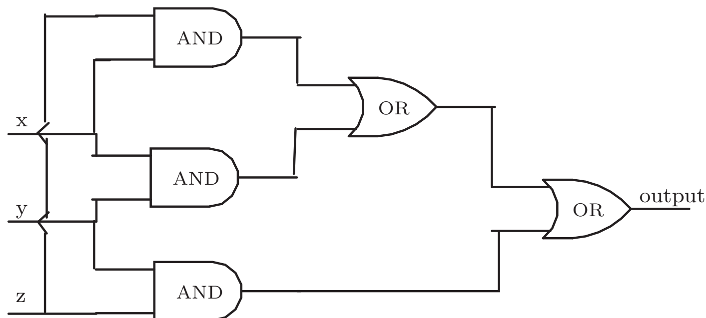


Figure 3: A digital circuit for the formula from example 3.

With this we arrive to the notion of circuits corresponding to the circuits in computer hardware. A hardware circuit consists of elements (called gates) performing exactly the operations of conjunction, disjunction and negation. The following figure shows the notation used in digital circuits and computer architecture literature to draw circuits.

A node in the tree or circuit diagram is usually called the *gate*. In a Boolean circuit, there are $\vee, \wedge, \neg$ gates, and also *input* gates and one *output* gate. The figure 4.1 shows a circuit for the formula from example 3 written as a digital circuit.

Example 4.
Recall the Boolean function $Majority(x, y, z)$ which evaluates to 1 if and only if at least two of its inputs are 1. It can be represented by the formula $(x \wedge y) \vee (y \wedge z) \vee (x \wedge z)$. The following circuit computes majority of its three inputs.



So how do we construct a circuit for a given function? The most natural way (although not giving the best circuit in many cases) is to start with a truth table of a function, write a CNF or a DNF formula for it, then simplify it to make it smaller, and finally encode it as

a circuit. Can we do better then that? Sometimes, yes, but in general we do not have an algorithm that, given a truth table, constructs us the best possible circuit (well, short of a very brute-force method of trying all possible circuits up to the size of the truth table to see if any of them work). This is an open problem in computer science, to design such an algorithm. However, we know that for most Boolean functions the best circuit is of the size comparable with the size of its truth table (think about a "random-looking" column in a truth table: how would you describe it other than writing it out?).

4.2 Doing arithmetic with Boolean circuits

Binary notation: every natural number can be written using only 0s and 1s. There is a convention for defining negative numbers, rational numbers, reals and so on (for example, real numbers are infinite strings of 0s and 1s). Just as a number 209 in decimal notation is $2 \cdot 10^2 + 0 \cdot 10^1 + 9 \cdot 10^0$, in binary notation it is 11010001.

How to convert from decimal into binary? If the number is even, put a 0, if it is odd, put a 1, then divide by 2 (rounding down) and repeat until got to 1. Digits of a number in binary are called bits.

What does it have to do with Boolean circuits? Boolean circuits can compute bits of numbers.

Example 5. Think of adding numbers $3_{10} = 11_2$ and $5_{10} = 101_2$ (here, the subscript denotes which base we are using, 10 or 2).

$$\begin{array}{r} 11 \\ 011 \\ 101 \\ \hline 1000 \end{array}$$

Here, the top row keeps track of the carry as we are adding bits from right to left. This tells us that to compute a bit in the middle we need three inputs: the two corresponding bits of the numbers and a carry from the previous step. This also tells us that here we need two outputs of the circuits (in terms of Boolean functions, we can think about each output as computed separately; however, in real life we reuse parts of circuits to produce several outputs.

A construction of a circuit for addition consists of three parts. First, we construct a *half-adder* which adds two bits without considering the carry and outputs a sum and a carry from that addition. Such a circuit could be used to add two one-bit numbers and is used to add the two lowest bits of the numbers. Then we will use a half-adder to construct a circuit *full adder* which adds two bits taking into account the carry. And then a number of such full adders are connected recursively (that is, with an output of one feeding as an input do the next) to compute the sum of an n-bit number. See figure 5 for the circuit diagrams.

In the same fashion any arithmetic function can be evaluated using Boolean circuits to evaluate bits of the resulting number.

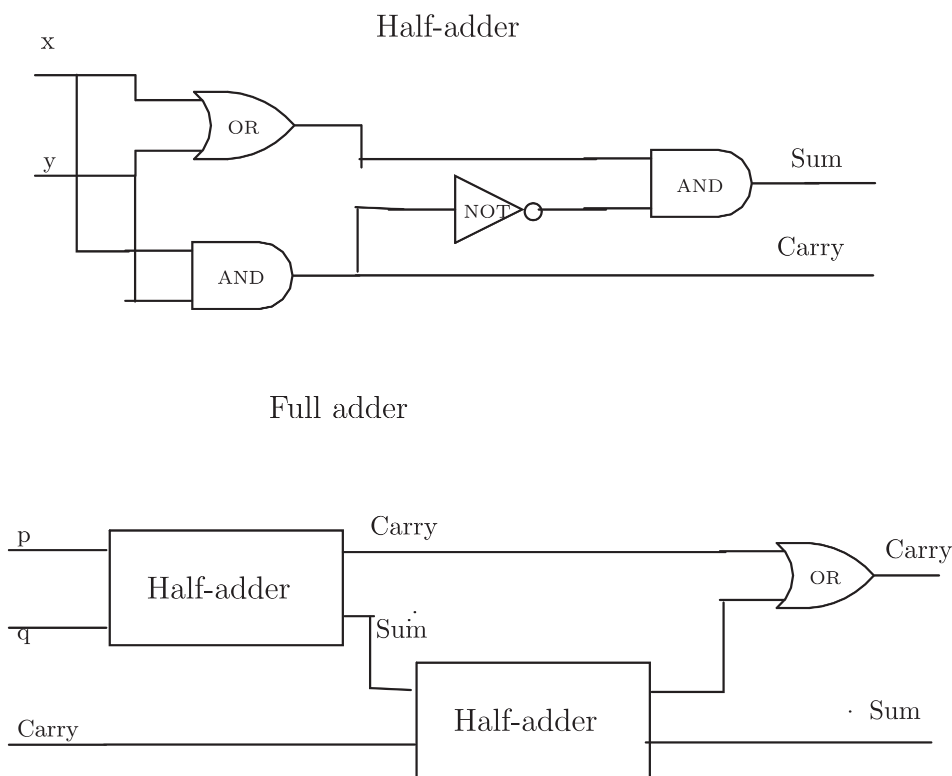


Figure 4: A half-adder and a full adder used to add numbers in binary