Week 2: Introduction - Digital logic and Boolean algebra

- Binary can be considered as truth values:
  0 = False, 1 = True
- Boolean Algebra: the laws of algebra for truth values
  – consists of operations on truth values, the result of which is another truth value
  – fundamental for the design of logic circuits and for tests in computer programming
- CPU hardware consists of circuits built from logic gates
  – logic gates perform Boolean Algebra

Boolean algebra

Boolean algebra is appropriate for computing, since it is based on the notion of true or false, which a computer can easily handle as 0 and 1. It consists of operands that are either true or false, and operations to combine those operands. Usually, true is represented as 1, and false as 0. The following shows the most common operators:

AND operator

- returns true when both operands are true. Its algebraic symbol is . (period), and in programming it is a && (double ampersand). A table of results with this operator is as follows:
  – written as x.y or sometimes x∧y or as xy

AND	. or &&
0 . 0	= 0
0 . 1	= 0
1 . 0	= 0
1 . 1	= 1

OR operator

- returns true when *either* operand is true. In algebra it is denoted as + (plus), while in computer programming it is a || (double pipe).
  – written (confusingly!) as x+y or sometimes x∨y

OR	+ or ll
0 + 0	= 0
0 + 1	= 1
1 + 0	= 1
1 + 1	= 1

NOT operator

- this is a *unary operator* (it only takes in one term) that returns the opposite value; that is, it returns true if the operand is false, and false if the operand is true. Its algebraic symbol is - (minus), and in programming it is a ! (exclamation point).

NOT	- or ~ or !
0	1
1	0

(written as /x or -x or sometimes !x or $\bar{x}$)

XOR operator

- XOR (exclusive OR) is similar to OR, except it is true when only one, and exclusively one, input is true
  – written as $x \oplus y$

XOR	⊕
0 . 0	0
0 . 1	1
1 . 0	1
1 . 1	0

Operator precedence

Similarly to normal arithmetic, to make sure that operations are performed in the right order, these operators are given a precedence. For example, we know that in normal arithmetic 2 + 3 * 4 is interpreted as 2 + (3 * 4) = 14 , rather than (2 + 3) * 4 = 20, since the multiplication 'binds stronger' than the sum. If we need to change the precedence of the evaluation, we use brackets.

When a Boolean operator has a higher precedence than another, it also means that it binds stronger, with similar consequences. As it may be seen in the table, the highest precedence is the

(.)	*highest precedence*
NOT	
AND	
OR	*lowest precedence*

Therefore:

-a + b . c + d = (-a) + (b . c) + d

Note that since the () have higher precedence that any other operator, you can use () to change the precedence of evaluation to whatever is required. For example the expression above may be changed to:

(-a) + b . (c + d)

if so desired; the brackets forcing the evaluation the new way.

These Boolean operations have particular characteristics that are different to the ones we are used to. This table shows a few:

0 + X = X	0 . X = 0	X = -(-X)
1 + X = 1	1 . X = X	
X + X = X	X . X = X	
X + -X = 1	X . -X = 0	

Since these operators work on binary data, it is often possible to prove an identity such as the ones above by going through all the possible situations that may occur, verifying that the equality holds true at all times. This is called a *truth table* and they are used very often. As an example, let us prove the identity X + X = 1 using a truth table:

Truth table for X + -X = 1

X	**- X**	**X + - X**
0	1	1
1	0	1