Lecture 1, 17th September 2012

Electronic Methods in the Physical Laboratory

Junior Honours – University of Edinburgh

Dr Stephan Eisenhardt, rm 5418, S.Eisenhardt@ed.ac.uk

Course Layout (15 lectures á ~50 minutes)

Digital Electronics (3 lectures)
Analogue Electronics – linear (6 lectures)
Analogue Electronics – non-linear (4 lectures)
Analogue meets Digital (1 lecture)
Digital Signal Processing – using LabVIEW (1 lecture)

Text Books (for occasional consultation in the library)

The Art of Electronics
P. Horowitz & W. Hill (CUP, New York, 1989) TK7815 Hor.
Student Manual for the Art of Electronics
T.C. Hayes & P. Horowitz (CUP, New York, 1989) NLS: SPR3.89.534

Course Structure

period 1: Mon 17.09. - Fri 19.10. (5 weeks: 15 lectures, 5 lab sessions)
period 2: Mon 22.10. - Fri 02.11. (2 weeks: design exercise part 1)
period 3: Mon 05.11. - Fri 23.11. (3 weeks: 3 lab sessions)
period 4: Mon 26.11. - Fri 07.12. (2 weeks: design exercise part 2)

Lectures – period 1

Monday, Thursday, Friday 14:00
Location: JCMB – LTC

Laboratory – period 1 & 3 (each student attends **<u>one</u>** tutored session per week)

seat limit: 20 students per session

Tutored sessions:	period 1	period 3
Monday, Tuesday, Thursday Friday	15:00–17:00	14:00-17:00

Location: JCMB - Lab 3301
weight: 60% of total assessment of course

Untutored lab access:

Further lab access is possible, but *without* tutoring:

Tuesdays:	09:00-14:00
Wednesdays:	14:00-17:00
Thursdays:	09:00-14:00

Design Exercise: (a number of design problems)

Issue of first sheet: week 5 (to be worked on in period 2)
Issue of second sheet: week 9 (to be worked on in period 4)
submit solutions by 07.12.2011: 17:00
weight: 40% of total assessment of course
(Design Exercise 3 can be solved either using Labview or JAVA, using skills from Computational Physics course)

Website

Lecture notes and design exercises and further teaching material becomes available from: http://www2.ph.ed.ac.uk/~eisenhar/teaching/courses.shtml

Digital Electronics 1: Binary numbers & combinational logic

The course begins with digital electronics because we find that the digital circuits are easier for students to build in the lab. We will progress onto analogue electronics next week and then finish the course with converting analogue signals into digital signals.

Analogue versus digital: (H&H 8.02 p.472)

The world's representation turns increasingly digital. Since microchips began to appear in the 1960s digital electronics has been playing a larger and larger part in people's lives. With the advent of the CD in the (fabulous) 1980s, first sound went digital, followed by still images and finally video; at the start of the 21st century, *digital memory chips* are the most used media (e.g. in MP3-players, USB-keys, mobile phones and digital cameras).

Digital technology stores information in discrete values, represented by **binary states**. Example pairs of binary states are:

in Boolean Logic:	True / False
in Binary Numbers:	1 / 0
represented as Voltages or Currents:	High / Low (or Low / High!)
in Gene activation patterns:	activator / repressor

One binary value is called a **bi**nary dig**it** or **bit**. Storing information as bits has significant *advantages*, for suppressing noise and for subsequent processing. The disadvantage is a *loss of subtlety*. Digital information:

- yields a *smaller dynamic range* than analogue signals,
  this is constrained by the size of one bit
- is *ideal for counting* / integer data
- is *more robust* than analogue information and can be transmitted without signal loss,
  only the "1" and "0" states have to be recognized, analogue noise and dispersion can be effectively suppressed
- needs *smaller bandwidth* for transmission of analogue-to-digital converted data
- and *allows logic combination* of signals and makes computers possible:
  digital signals are intimately tied to the way computers work.

For a physicist, digital electronics is important in processing signals on all scales – from a single parameter temperature sensor in the table-top setup to a 10-million channel particle detector at the Large Hadron Collider. Digital electronics allows to **trigger** (i.e. to react with a defined delay in time) on the occurrence of pre-defined signal conditions. E.g. the observation of a signal corresponding to the presence of a muon can be registered as a "1". Two "1"s recorded simultaneously in different detectors can be used in coincidence counting.

Physicists employ computers to process and analyse large amounts of data at increasing speed. For that the readings of analogue sensors first have to be converted into binary form in a **digitization procedure**, introducing an *additional loss of fidelity* on top of the analogue noise. Any sensory readout can be converted into digital values, and these days is done so basically everywhere by default, e.g. astronomic and medical imaging, remote sensing, any form of recording from microscopy (whether light/ electron/ atomic force), take your pick ...

Lecture 1, 17th September 2012

Digital signals:

Anything to which there is a yes/no answer can be very easily represented in binary form. The assignment of the meaning of these states is by *convention*, i.e. can be freely defined.

Any more complex information, e.g. rational numbers, is less straight forward to represent in digital form. Using various conventional notations, sequences of binary states (binary numbers) can be used to represent such information. The length of the bit sequence determines the *precision* and *range* of the representation. E.g. to represent a floating point (rational) number, a single (16-bit), double (32-bit) or long (64-bit) **word** of bits can be used. Once numbers, or other data, have been represented in binary code they can be manipulated using Boolean logic (AND, NOT, OR).

Number Codes: (H&H 8.03 p.473)

To get a flavour for the issues involved we will revise the method to convert between *base 10* (decimal) and *base 2* (binary) unsigned integer numbers. We then will think about the best way to code a minus sign to represent signed integer numbers.
More complex codes, e.g. for floating point numbers, can be found in the literature.

A base 10 integer number is made up of units, tens, hundreds etc. We need to be able to *convert* this into a base 2 number made up of units, twos, fours, etc. Let's use the example decimal number: $213_{10} = 2 \times 10^2 + 1 \times 10^1 + 3 \times 10^0$

The conversion to binary is carried out by repeatedly dividing the decimal number by 2. When only using integer numbers, after each division there is a remainder 'r' which is either a "1" or a "0". The series of remainders makes up the binary representation of the number, starting with the smallest digit.

Decimal to Binary – example conversion

213 / 2 =	106	r	1
106 / 2 =	53	r	0
53 / 2 =	26	r	1
26 / 2 =	13	r	0
13 / 2 =	6	r	1
6 / 2 =	3	r	0
3 / 2 =	1	r	1
1 / 2 =	0	r	1

$$213_{10} = 11010101_2$$

The opposite direction, the binary to digital conversion, can be carried out by adding up the units, twos, fours etc. that make up the number.

Binary to Digital – example conversion

$$\begin{aligned} 101_2 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 4 + 0 + 1 = 5_{10} \end{aligned}$$

How can you represent a minus sign?

An obvious approach would be to *dedicate one bit* of the binary number, e.g. its *most significant bit* (MSB), to indicate its sign, like we do in writing. This is called **sign magnitude** representation and is regularly used to represent floating point numbers (coded as sign-exponent-mantissa). But computational handling of integers in this approach is awkward: binary addition and subtraction would need different procedures and the sign bits extra treatment, there also would be two zeros “+0” and “-0”. There are more efficient solutions, allowing for much simpler circuits to perform basic operations.

The first step in the right direction to compute with signed integers is the **offset binary** representation. Here one *subtracts the half the maximal numbers* which could be presented as unsigned integer. With this the binary number sequence increments continuously from the most negative to the most positive number, the MSB indicates the sign and the problem with the two zeros is gone. This works well with A/D and D/A conversions.

But signed integer computation becomes easy once in addition the *MSB gets inverted*. Then positive numbers are represented again as simple unsigned binaries. And a positive number can be turned negative by first complementing each bit of the number (“*1’s complement*”) and then adding a binary 1 (“*2’s complement*”), hence the name **2’s complement** representation. In this representation binary adding a positive number and its negative counterpart of the same value yields zero, with zero been represented by “0” bits only. Binary addition, binary subtraction (i.e. adding the 2’s complement of the subtrahend) and binary multiplication yield in this representation the correct numerical results, using circuitry designed for positive integers alone!

See the following table for the relation of the three representations. But here we will not take this topic further.

Integer	sign-magnitude	offset-binary	2’s complement
+7	0111	1111	0111
+6	0110	1110	0110
+5	0101	1101	0101
+4	0100	1100	0100
+3	0011	1011	0011
+2	0010	1010	0010
+1	0001	1001	0001
0	0000	1000	0000
-1	1001	0111	1111
-2	1010	0110	1110
-3	1011	0101	1101
-4	1100	0100	1100
-5	1101	0011	1011
-6	1110	0010	1010
-7	1111	0001	1001
-8	-	0000	1000
(-0)	1000	-	-

Combinational logic: (H&H 8.04 p.478)
So we can record any data (logic states, decimal numbers, etc) in binary states, in form of "1"s and "0"s – now we want to manipulate them. This is typically carried out using digital circuitry, called **logic gates**. You will be building electronic circuits using logic gates in the lab.

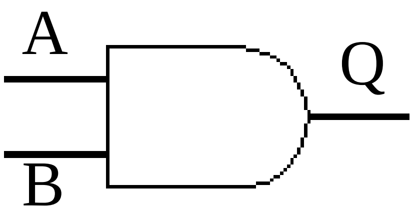
The binary states are combined and manipulated using Boolean Logic, using operations like NOT, AND, NAND, OR, XOR and NOR. Each of these operations is available as a distinct logic gate, typically implemented in **integrated circuit chips**. Using these as building blocks one can put together the circuitry to implement *any logic algorithm*, for example, one to add binary numbers.

The subset of logic gates {AND, OR, NOT} is called the **fundamental logic functions**. All other logic circuits can be built using only these three logic functions.

The logic gates NAND and NOR are called **universal logic functions**. It is possible to implement every other logic function just using NAND gates – and the same holds for NOR gates. Theoretically an entire PC CPU could be built from just NAND gates (or NOR gates), practically this is not very efficient.
In the lab you are provided with NAND gates (and NOT gates, to ease some tasks).

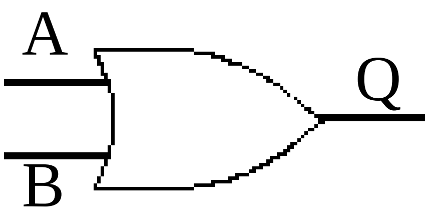
Logic gates:
Each logic gate has *one or more inputs* and exactly *one output* and is represented by a specific symbol. We use the convention to label the input wires A, B, C, etc, and the output wire Q. The logic function of the gate is defined in the **truth table**, where the output Q is defined for all possible combinations of inputs.
Below, for the fundamental and universal logic gates, you find: the name, the way they are written as an arithmetic operation, the conventional symbol and the truth table.

AND (A.B)



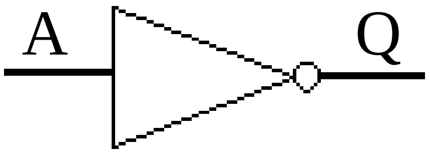
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR (A + B)



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

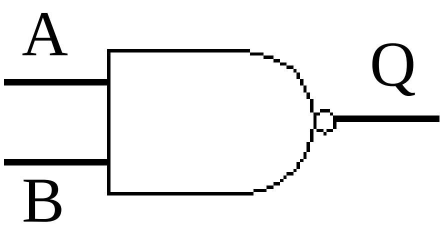
NOT A'



A	Q
0	1
1	0

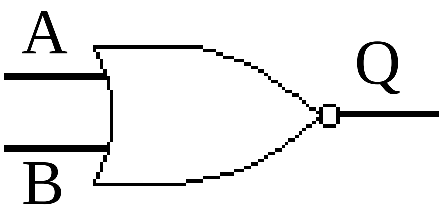
Lecture 1, 17th September 2012

NAND (A.B)’



A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NOR (A + B)’



A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Boolean arithmetic: (H&H 8.12 p.491)
The arithmetic operation associated with each gate (for example A.B is AND) can be used to simplify logical problems. Typically, if you understand the problem that you are trying to solve then you can write it in the form of a truth table.

Truth tables codify statements such as "if the fridge door is shut the light should be off". In this example you can choose to define the states as follows: A(door open) = 0 and Q(light on) =0. Note that this assignment of states may feel unnatural to you in the first instance. But it has the benefit of an easy mathematical expression (Q = A) and you may want to use a hardware implementation of the logic where the state "1" is the low power state, like in *emitter coupled logic* (ECL), to save power over the extended times where the fridge door is closed. The truth table in this example looks like:

A	**Q**
0	0
1	1

This truth table is that of a *source follower*, which is regularly used in *line drivers* to transmit signals over longer distances or to *match impedances* between two different parts of a circuit. But that already leads into the inherent analogue nature of even digital electronics and we will return to that at a later stage.

Once you have codified your problem using logic statements the code may look rather complex. Often it can be simplified using Boolean arithmetic, specifically by using **Boolean identities**, which can be used like standard mathematical identities in arithmetic. For your reference you find a useful subset tabled below.

OR function 0+A = A 1+A = 1 A+A = A A+A' = 1	AND function 0.A = 0 1.A = A A.A = A A.A' = 0	NOT function (A')' = A
Association A+(B+C) = (A+B)+C A.(B.C) = (A.B).C	Commutation A+B = B+A A.B = B.A	Distribution A.(B+C) = A.B+A.C (A+B).(A+C) = A+B.C
Absorption A+A.B = A A.(A+B) = A	De Morgan's Theorems (A+B)' = A'.B' (A.B)' = A'+B'	

We are going to go now through some examples of logic design where we apply the introduced elements.

Logic Design:

The elements needed to solve a logic problem were sketched above. Generally, you should succeed if you follow this sequence of steps:

1. Create a truth table which matched your needs
2. Write down the logical expression that describes the truth table*, possibly simplify using Boolean algebra
3. Draw the circuit in terms of the fundamental logic gates

*Alternatively one can employ the method of *Karnaugh maps* (H&H 8.13 p.492) to find the logic expression for problems with up to four input variables.

Example 1 – Create an NXOR gate

A NXOR gate is a negated, exclusive OR gate. An exclusive OR (XOR) gives an output Q=1 if exactly one of the inputs is 1. Negating that means an output of Q=1 if both of the inputs are the same.

1. Create a truth table:

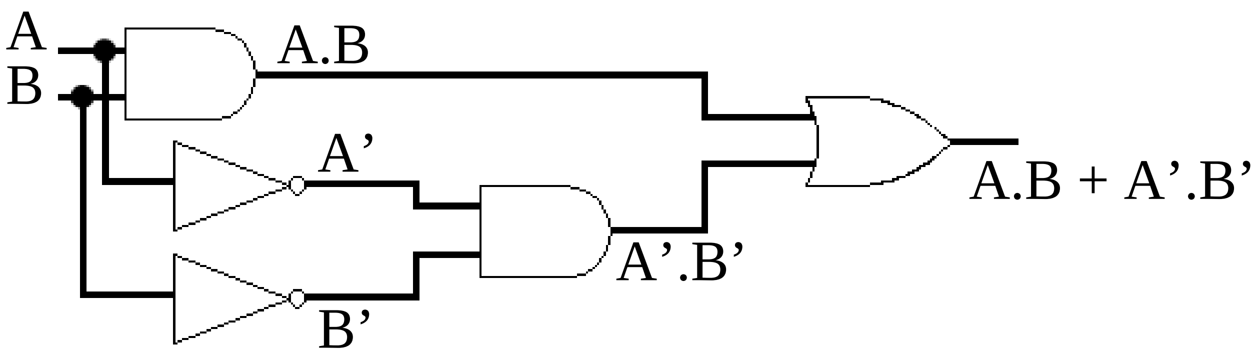
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

2. Write down the logical expression:
   - There are two rows (1 & 4) in the truth table with Q=1
   - We can test for row 4 by saying: Q = A.B (meaning Q=1 if A=B=1)
   - Row 1 is the opposite of this, hence: Q=A'.B' (meaning Q=1 if A=B=0)
   - These two conditions can be combined via an OR gate to collect all Q=1 cases:

     Q = A.B + A'.B'

This is the arithmetic expression which describes the truth table.

3. Draw circuit in terms of fundamental gates:



- We have two conditions that are combined by an OR gate, which is the final element of the circuit.
- The first condition (Q=A.B) is implemented in the top AND gate and feeds into the top input of the OR gate.
- The second condition (Q=A'.B') employs two NOT gates to first complement the inputs before a second AND gate tests for the condition. The result is feed into the second input of the OR gate.

Example 2 – Create a greater-than circuit for 2-bit numbers (A > B)
Design a circuit which compares the sizes of two 2-bit numbers A and B and outputs Z=1 if A is larger than B. The numbers A and B consist of the bits A1, A0 and B1, B0, respectively.

1. Create a truth table:
We are comparing two numbers between 0 and 3. The truth table covering all possible comparisons has 16 entries. We need two columns to represent each of the two inputs.

A1	A0	B1	B0	Z
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	0	1
0	0	0	1	0
0	1	0	1	0
1	0	0	1	1
1	1	0	1	1
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	1	0	1
0	0	1	1	0
0	1	1	1	0
1	0	1	1	0
1	1	1	1	0

It is more compact, and also more useful, to display the truth table a different way. Since we want to compare all possible values of A with all possible values of B we can put them at the edges of a grid and then write the output values in the grid spaces:

A1A0\B1B0	00	01	10	11
00	0	0	0	0
01	1	0	0	0
10	1	1	0	0
11	1	1	1	0

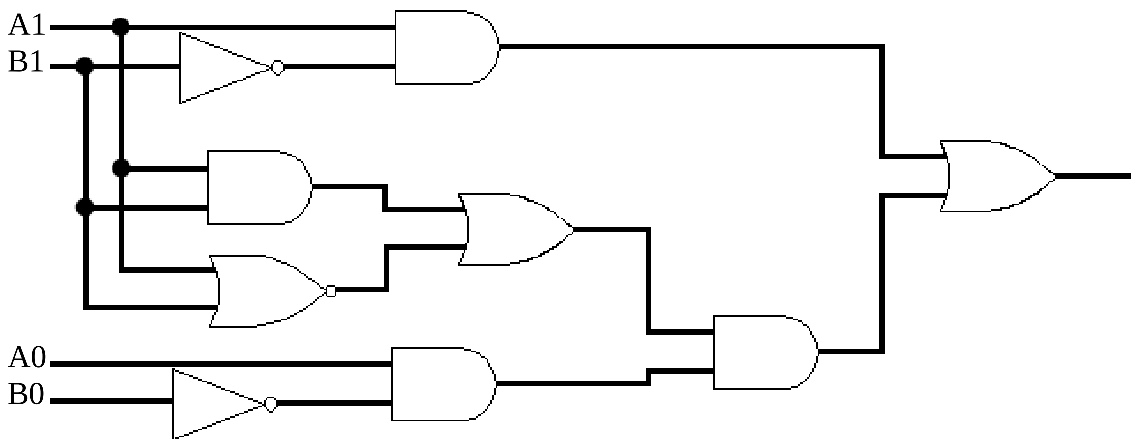
This arrangement makes it easier to see groups of "1"s and hence to determine a logical expression for this operation.

2. Write down the logical expression:
The input conditions giving an output of 1 can be grouped.

- The four "1"s in the shape of a square in the lower left hand corner can all be described in one go: if the most significant bit of A is 1 (A1=1) and the most significant bit of B is 0 (B1=0) then the output is 1. This gives:
  Z = A1.B1'
- The two remaining "1"s can also be grouped: these are the cases where the most significant bit of A and B are the same (A1=B1) and the least significant bit of A is 1 (A0=1) while the least significant bit of B is 0 (B0=0). This gives:
  Z = (A1.B1 + A1'.B1').(A0.B0')

- As with example 1 these two conditions can be combined via an OR gate:
  Z =(A1.B1’) + (A1.B1+A1’.B1’).(A0.B0’)
- This is now quite a long expression which would require 5 AND gates, 4 NOT gates and 2 OR gate to implement. Can it be implemented with fewer than 11 gates?
- If you look at the table of Boolean identities above you will find De Morgan’s theorem, i.e. that:
  (A1’.B1’) = (A1 + B1)’
- We can use it to reduce the number of gates by one:
  Z =(A1.B1’) + (A1.B1+(A1+B1)’).(A0.B0’)

3. Draw circuit in terms of fundamental gates:
We need four inputs for two-bits from two numbers and one output. The circuit looks like this:



- We have two conditions to a final OR in this circuitry.
- The first condition is shown in the top section where A1 and B1, via a NOT gate, are evaluated in an AND – this represents the first term in the expression.
- The bottom section compares A0 and B0 in the equivalent way – this represents the last term of the expression, i.e. the second term to the AND which determines the outcome of the second condition to the final OR.
- The middle section – notice the NOR gate, saving one more node – is where the evaluation of A1 and B1 is performed which we simplified using the De Morgan’s theorem.

Having worked through these concepts and examples you are now supposed to understand how to put together logic gates to solve basic problems with binary numbers.

Working the technicalities in the lab…

Very soon you are going to be building some logic circuits in the lab. A key question will be: what is "1" and what is "0"? Over the decades, as the technology of integrated circuits and the demands to it evolved, several different standards have been defined. In the lab we use the ***Transistor-Transistor Logic*** (**TTL**) convention – the oldest and most robust standard, still widely in use. Other commonly used conventions are: the *Nuclear Instruments and Methods* (NIM), *Emitter Coupled Logic* (ECL), *Complementary Metal Oxide Semiconductor* (CMOS) and *Low Voltage Differential Signal* (LVDS) standards.

TTL voltage convention: (H&H p.475)
Based on a power supply of +5V, the logic levels correspond to voltage levels, roughly like:

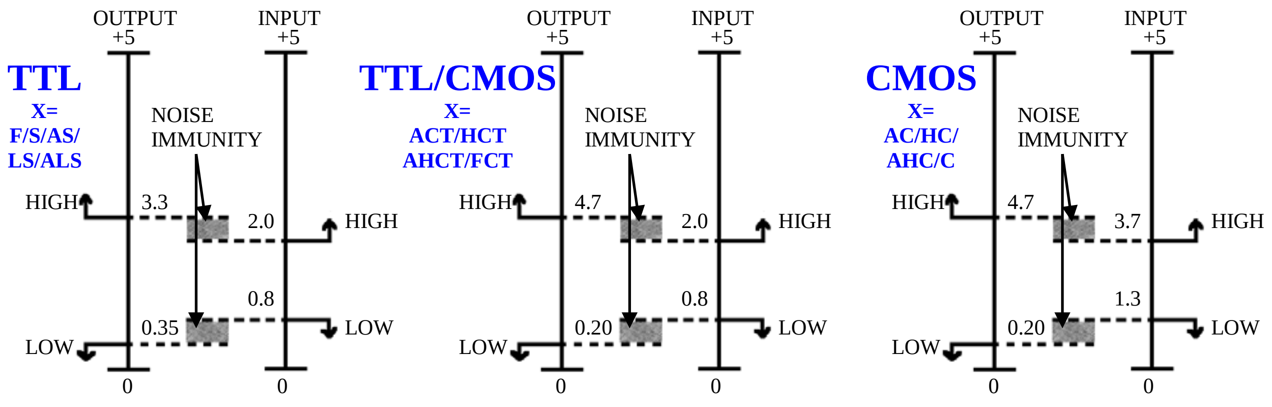
logic "0" ≈ 0V
logic "1" ≈ 4V

In detail one finds the following margins and transition thresholds for the **basic TTL family**:

TTL family **74yy**	Minimum (V)	Nominal (V)	Maximum (V)
Supply Voltage (V_{cc})	4.75	5.00	5.25
Input: High (Logic 1)	2.00		5.25
Input: Low (Logic 0)	0.00		0.80
Output: High (Logic 1)	2.40	3.40	
Output: Low (Logic 0)		0.20	0.40

As you will have noticed, the input to indicate a "1" state may be much lower (>2V) than the output will typically be (3.4V). Likewise the input to indicate a "0" state may be much higher (<0.8V) than the output will typically be (0.2V). The difference between the state conditions at input and at output is called the **noise margin**. This can be seen as the logic states being cleaned up at each gate.

You see this behaviour illustrated below for other selected **TTL** and **CMOS families X** you may come across (**chip IDs 74Xyy**). Note: a TTL chip may not properly switch a CMOS chip.



The noise margin effectively means that the signal is born again at each gate – rather than being progressively corrupted as it traverses a complicated circuit. The fact that the signal is renewed in this manner is an important aspect of digital circuitry and computer memory.

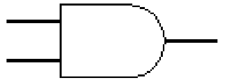
You are now supposed to be able to distinguish TTL "1"s and "0"s by voltage measurements.

Lecture 1: key equations

Fundamental logic functions:

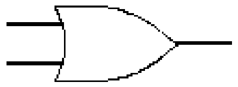
AND (A.B)

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



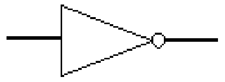
OR (A+B)

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



NOT (A')

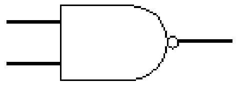
A	Q
0	1
1	0



Universal logic functions:

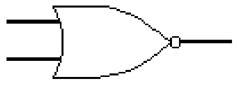
NAND (A.B)'

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



NOR (A+B)'

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0



1. Association $A+(B+C) = (A+B)+C$
   $A.(B.C) = (A.B).C$
2. Commutation $A+B = B+A$
   $A.B = B.A$
3. Distribution $A.(B+C) = A.B+A.C$
   $(A+B).(A+C) = A+B.C$
4. Absorption $A+A.B = A$
   $A.(A+B) = A$
5. De Morgan's Theorems $(A+B)' = A'.B'$
   $(A.B)' = A'+B'$

Voltage conventions:

Transistor-Transistor Logic (TTL)

TTL: Low < +0.8 V High > +2.0 V

Complementary Metal Oxide Semiconductor (CMOS)

CMOS: Low < +1.3 V High > +3.7 V