P2 Digital Electronics

Part 1: Lectures A-D

Mark Cannon, Trinity Term 2026 mark.cannon@eng.ox.ac.uk

Notes originally prepared by Prof. Chris Stevens, revised by Prof. Martin Booth, Prof. Perla Maiolino and Prof. Mark Cannon

Contents

Syllabus

Basic gates, truth tables, combinational functions (AND, OR, NOT, EX-OR). The MOSFET as a switch; CMOS inverter, NOR and NAND gates. Karnaugh maps; algebraic laws (such as distribution and association). Multiplexers, ROMs and PLAs. Binary arithmetic: adders/subtractors. Sequential logic I: D-type flip-flops, registers, asynchronous counters. Sequential logic II: synchronous counters, Karnaugh transition maps. Data converters; basic principles of DACs and ADCs. R-2R ladder based DAC. Principles of ADC (Flash and SAD).

Suggested reading

These lecture notes have been prepared to provide a concise but complete coverage of the syllabus. Please remember that these notes can only be a summary of what is available in textbooks. A wide range of textbooks can be found that cover this subject and the list provided below is only a sample of what is available, so do look around. Some of the texts listed are now quite old, but cover the necessary material and are often still available in College libraries or second hand at good prices. However, similar material is also covered in the newer books.

- Digital Fundamentals, Thomas L Floyd, Pearson, 11th Ed., 2015 (ISBN 1292075988)
- Logic and Computer Design Fundamentals, M.M. Mano and C.R. Kime, Pearson, 5th Ed., 2015 (ISBN 1292096071)

- Introduction to Digital Electronics, Crowe & Hayes-Gill, Butterworth Heinemann, 1998 (ASIN: B00XWRVPTI)
- Applied Digital Electronics, D.C. Green, Longman, 4th Ed., 1999, (ISBN 0582356326)
- Digital Logic and Microprocessors, Hill and Peterson, Wiley, 1984 (ISBN 0471085391)
- The Essence of Digital Design, B. Wilkinson, Prentice Hall, 1997 (ISBN 0135701104)

1 Lecture A – Logical functions and logic gates

1.1 Introduction and comments

One of the aims of this course is to describe the rules for basic logic design, such that you will be able to appreciate the essential features of complex digital systems. The design of microprocessors, memory chips, specific integrated circuits, digital signal processing circuits, etc. has reached such a level of complexity that it relies on extensive use of computers.

In fact, with computer optimisation algorithms being applied to large circuit systems, we are already at the point when computers design the next generation of computers. Already circuit diagrams are old-fashioned and nearly all circuit designs are produced using computer languages such as VHDL. These languages are often referred to as "silicon compilers". Software programmers define the process and algorithm to be implemented in a code which is then, after compilation and error checking, sent to a "silicon vendor" to turn it into a silicon integrated circuit.

Nonetheless, it is crucial to understand the basics, because ultimately every digital computer is a (very large) collection of simple logic gates which manipulates bits of information under the control of other logic gates.

The digital computer is a general-purpose machine. (In the 1960s, analogue computers enjoyed some success, but in the end digital computing triumphed

over analogue computing.) Digital computers use the binary number system to represent information. A binary digit (0 or 1) is called a bit. Groups of bits specify to the computer the instructions to be executed and the data to be processed.

Digital circuits are the hardware components which manipulate the bits. The circuits are implemented using transistors and interconnections in complex semiconductor devices called integrated circuits. Each basic circuit is referred to as a logic gate.

The course is divided into two sections. Part I deals exclusively with combinational circuits, i.e. collections of logic gates producing an output in combination, whereas Part II extends this and introduces sequential circuits operating in discrete time steps. This leads to the concept of a state machine (the basic building block of computers).

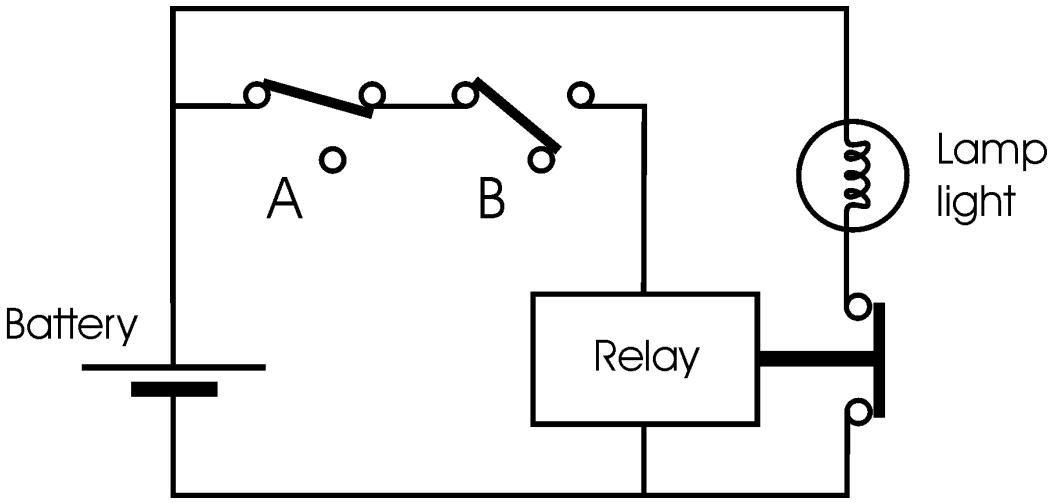
1.2 Binary logic

Binary logic deals with binary variables, and with the operations of mathematical logic applied to these variables. Boolean algebra (named after the English mathematician George Boole) describes how to perform logical operations in a binary logic system. Boolean algebra has only three basic operators: NOT, AND or OR.

Before we introduce the implementation of these Boolean operators using digital electronic circuits, let us consider the implementation of a simple logic function using an electrical circuit to switch a lamp on or off depending on the position of two switches.

1.3 Implementation of a logic function using an electrical circuit

In the electrical circuit below it can be seen that no current is reaching the relay as switch B is open. The relay is shown in its off position and the lamp should be lit. If switch B is closed then the relay arm is pushed out, and the lamp circuit broken. The light switches off.



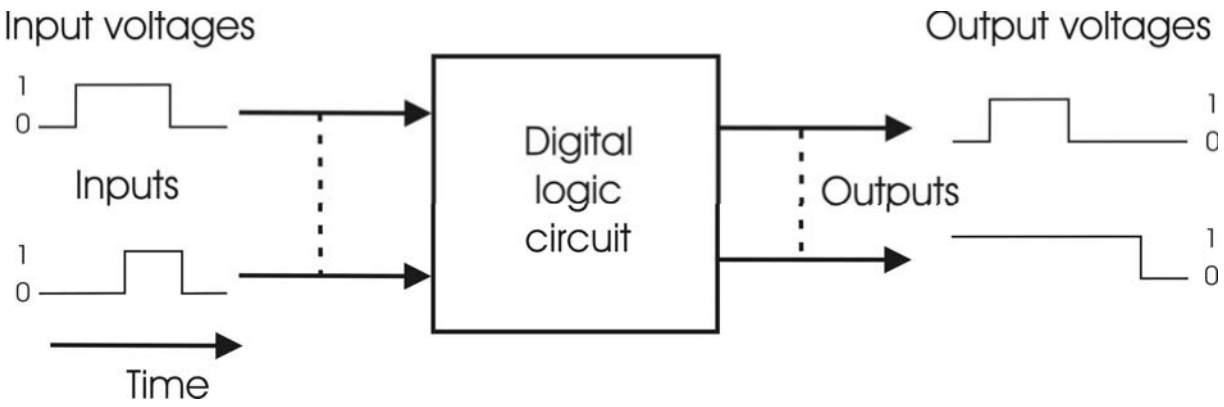
We can build up a simple logic (Boolean) expression to represent cases when the lamp is either ON or OFF. Below is a simple **Truth Table** description that illustrates all cases. Truth tables are a very helpful way of considering all possibilities arising in a logic system (i.e. a description of the output(s) of the system for all possible inputs).

Switch A	Switch B	Lamp state
Open	Open	ON
Open	Closed	ON
Closed	Open	ON
Closed	Closed	OFF

In this example, we have described the binary logic states using everyday language as "symbols" (e.g. Lamp on/Lamp off). Depending on the context, other symbols might be used to represent the state, e.g True/False or 0/1. In electrical logic circuits, the states are represented by two different voltage levels.

1.4 Digital electronic circuits

There are various different ways to implement logic in electronics. We will consider only transistor circuits, in which logic states are represented by voltage values at the inputs and outputs of the circuits.

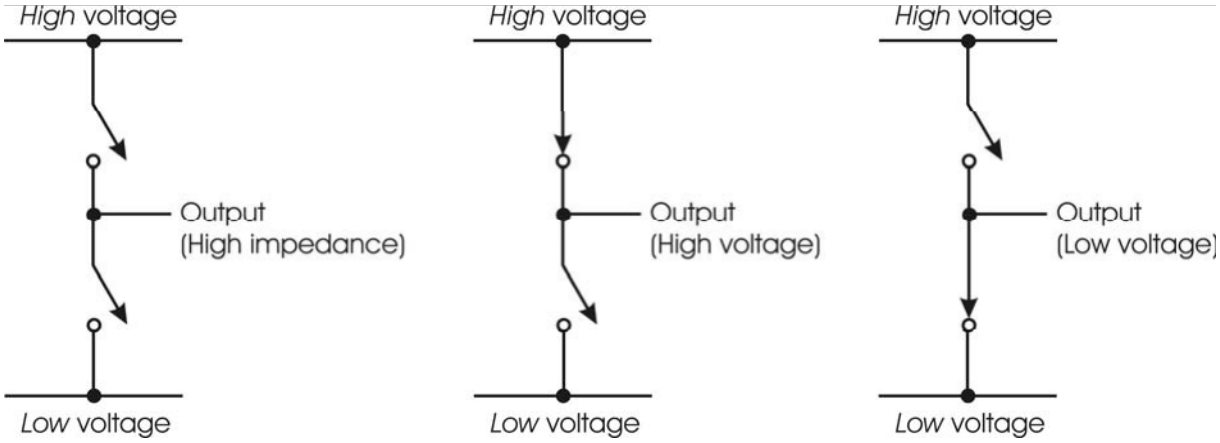


In the early days (1960's) of transistor-transistor logic (TTL), the voltage levels given in the table below were assigned to the logic 0 and logic 1 states. Note, however, that integrated circuit design has driven these voltages smaller in order to reduce the effects of power consumption as the transistor density increases. It is quite common these days to have logic 1 states represented by voltages that are lower than 2 volts.

Voltage level	Logic Value
0 V	FALSE, logic 0
5 V	TRUE, logic 1 state

1.5 MOSFET transistors as switches

The transistors used nowadays for logic circuits are based on CMOS designs (Complementary Metal-Oxide-Semiconductor Field Effect Transistors). To represent a logic level, two transistors are used in a configuration that is very similar to the switching circuit shown earlier – see the circuit below as an example.

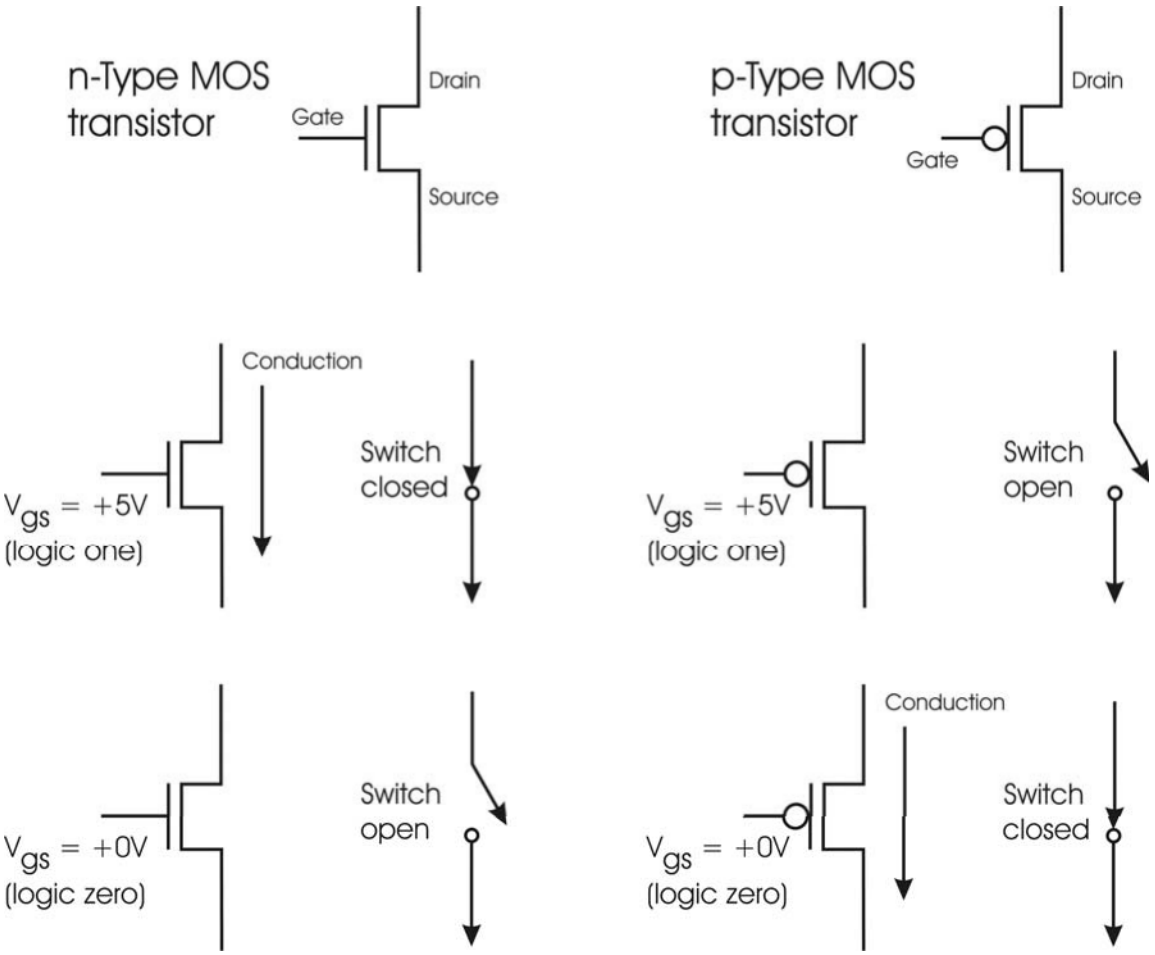


There is a switch at the top of the circuit and one at the bottom. The output is taken from the middle point. Normally one switch is open whilst the other is closed. There is a special case, which we will return to later, when both switches are open and the output is effectively in a high- impedance state.

To get a high (logic 1) output the top switch needs to be closed, whereas a low (logic 0) output is obtained when the bottom switch is closed. The case when both switches are closed is not considered, as this would short- circuit the supply rails. The actual transistor elements are given in the diagram on the next page. The term complementary in CMOS refers to the fact that the top transistor has a conducting channel that relies on hole transport (i.e. p-type conduction), whereas the bottom transistor is normally an n-type MOSFET (relies on electrons).

The two working together like this offer a number of advantages, the main feature being that power is only consumed by the transistors during a switch transition. If the transistors maintain a stable output (logic 0 or logic 1) then no power is being consumed. This is an important point, as thermal management is a considerable problem in the design of modern microprocessors.

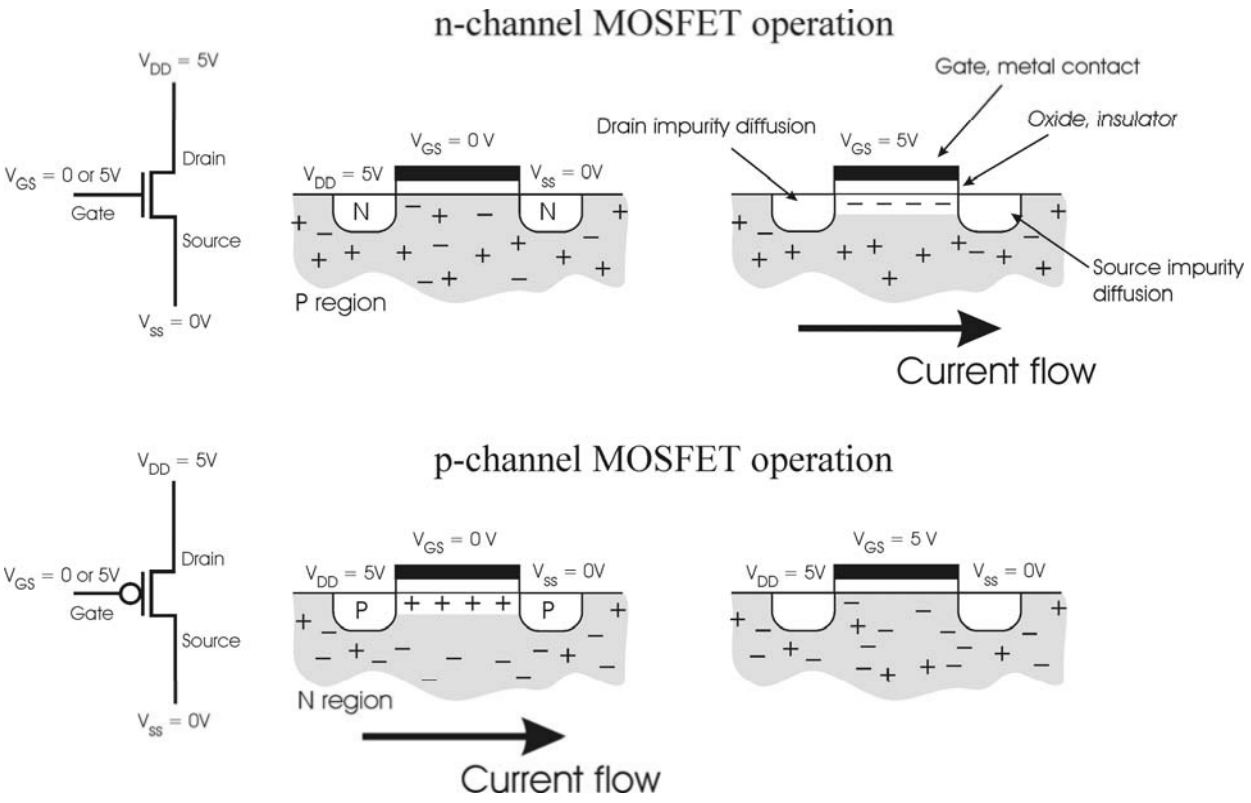
For an n-type MOSFET the channel is conducting when its Gate is at a logic 1 level. The opposite is true for the p-type device.



1.6 Overview of the basic principles of operation of the MOSFET

How do n-channel and p-channel MOSFETs work? How does the voltage applied to the gate electrode control the flow of current between the drain and source contacts (i.e. the drain-to-source resistance)?

The figure below gives a simplified picture of what happens in n-channel and p-channel MOSFETs as the gate voltage is switched between logic 0 and logic 1 (i.e. $0\,\mathrm{V}$ to the supply voltage V_{DD}, here assumed to be $5\,\mathrm{V}$). These lecture notes use a simple electrostatic model of MOSFET operation. Real devices are more complicated, and modern transistors use three-dimensional structures rather than the simple cross-sections shown here, but the same basic idea remains: the gate voltage changes the charge in the channel region and therefore changes how easily current can flow between source and drain.



In the simplified structure shown here, the gate is a conducting electrode separated from the silicon by a very thin insulating layer called the gate dielectric. In older designs the insulator was often silicon dioxide but modern CMOS commonly use a gate dielectric with a higher dielectric constant (since a pure SiO_2 layer would otherwise have to be made so thin that leakage current becomes too large). The source and drain are heavily doped regions on either side of the gate, which are formed by diffusing impurities into the silicon. Electrical contacts are made to the gate, source and drain so that the transistor can be connected into a circuit. Note that an n-channel MOSFET is made from p-type silicon, with an n-diffused drain and source region (a p-channel MOSFET is formed in an n-type body, with p-diffused drain and source region).

Let us consider the drain-to-source resistance for the n-MOSFET transistor first. In the cross-sectional view notice that the drain and source diffusions form a diode structure (p-n junction) with the bulk of the silicon region. With $0\,\mathrm{V}$ applied to the gate (top left diagram) current tries to flow from the drain to the source, but cannot because the diode structure is reversed biased and offers a high resistance. The MOSFET is described as being in cut-off.

When $5\,\mathrm{V}$ is applied to the gate, making it positively charged, the available electrons in the vicinity of the gate are attracted by an electrostatic force to the space immediately beneath the gate, creating a thin "channel" of electrons. These electrons form a continuous region of "n-type" semiconductor that allows current to flow such that the resistance between the drain and source is negligible. The space immediately beneath the gate is inverted from p-type to n-type, and the current from the drain to the source is regarded as being saturated. The value of gate voltage required to just achieve strong inversion is called the threshold voltage, V_T, and is typically about $0.2\,\mathrm{V}$. It is an important device parameter.

For the p-channel MOSFET, the doping of the bulk of the semiconductor is such that there exists a thin layer of holes beneath the gate when no gate voltage is applied. Under these conditions the drain-source resistance is negligible, and the MOSFET is described as saturated. However, when applying $5\,\mathrm{V}$ to the gate, making it positively charged, the holes are repelled by electrostatic forces and distribute into the bulk of the semiconductor. Now the influence of the reverse-biased diodes between the drain and the bulk of the semiconductor prevents current flow. The MOSFET is now in its high resistance state, or cut-off. The operation of the p-MOSFET is complementary to that of the n-MOSFET.

1.7 MOSFET current voltage characteristics

There is a simple square-law approximation for the current between drain and source. As we are only concerned with digital logic in this lecture course, we only need consider the MOSFET in one of the two states: cut-off (when no current flows) and saturation (when the drain- source current is at its largest value).

In **cut-off**:

The gate voltage, VG is less than the threshold voltage, VT and the resistance between drain and source is typically about 5×10^9 ohms (the intrinsic resistance in the vicinity of the gate region).

In **saturation**:

The saturated current flowing between drain and source, I_{DS} is given approximately by the following square-law equation:

$$I_{DS} = \frac{\beta}{2}(V_{GS} - V_T)^2 \tag{1.1}$$

Where β is a parameter relating to the physical dimensions of the MOSFET and the material from which it is made (usually Silicon)

$$\beta = \frac{\epsilon_r \epsilon_r \mu_{n,p} W}{t_{\text{ox}} L_c} \tag{1.2}$$

Where ϵ_r and ϵ_0 are the relative and free space dielectric permittivities for the silicon, $\mu_{n,p}$ is the mobility of the relevant charge carriers in the channel, t_{ox} is the oxide thickness, W is the width of the channel (or the gate which forms it) and L_c is its length.

For typical devices, the drain-to-source resistance, when the gate voltage for an n-channel device is at the logic 1 state, is $0.6\,\Omega$ – which is effectively a short circuit compared with the cut-off value determined above.

1.8 Basic logic gates and truth tables

If we return to the lamp circuit and its truth table, we can see that the lamp is lit when switch A or switch B is open. We note that logical expressions like "A AND B", "A OR B", and "NOT A" are denoted algebraically by $A.B$ (using notation like conventional multiplication), $A + B$ (using notation like conventional addition), and $\overline{A}$ (overlined), respectively. The mathematical representation is therefore:

$$\text{Lamp} = A + B \quad \text{or} \quad \text{Lamp} = \overline{A.B} \quad \text{or} \quad \overline{\text{Lamp}} = A.B \tag{1.3}$$

"Lamp" here is used as shorthand for "The lamp is on". These say Lamp is (A or B), Lamp is not (A and B) or the inverse of Lamp is (A and B). The same information can be conveyed using a Truth Table as follows, assuming that an open switch corresponds to logic 0, a closed switch to logic 1, and that the lamp being lit is a logic 1:

A	B	Lamp, or output C
0	0	1
0	1	1
1	0	1
1	1	0

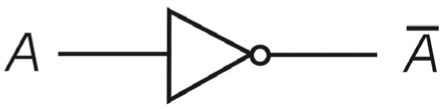
Note that the column entries for inputs A and B are an exhaustive list of all possibilities. This is an important convention, and helps to avoid missing an input condition by accident.

The diagrams below give the symbols and corresponding truth tables for the common logical functions. In addition to AND, OR and NOT, it is also sometimes convenient to use gates such as NAND (not AND) and NOR (not OR). The exclusive or (XOR or ExOR) gate gives a 1 at the output when either input is high, but not both.

Truth table

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

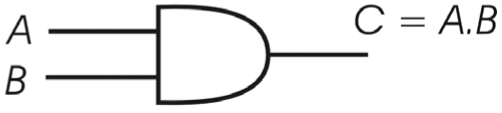
NOT gate symbol



Truth table

A	B	$C = A.B$
0	0	0
0	1	0
1	0	0
1	1	1

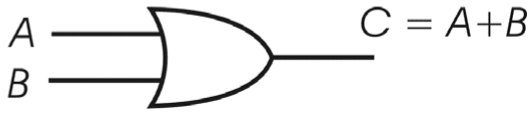
AND gate symbol



Truth table

A	B	$C = A+B$
0	0	0
0	1	1
1	0	1
1	1	1

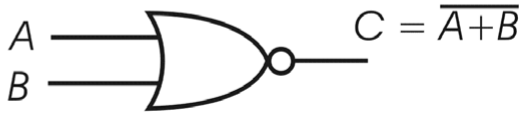
OR gate symbol



Truth table

A	B	$C = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

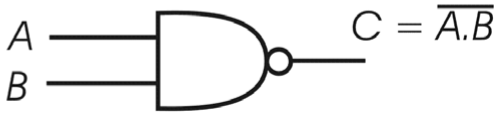
NOR gate symbol



Truth table

A	B	$C = \overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

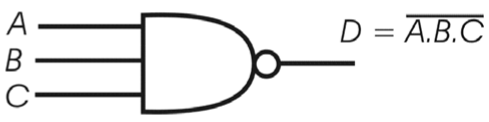
NAND gate symbol



Truth table

A	B	C	$D = \overline{A.B.C}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

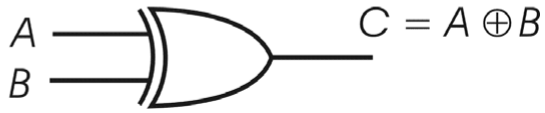
Three input
NAND gate symbol



Truth table

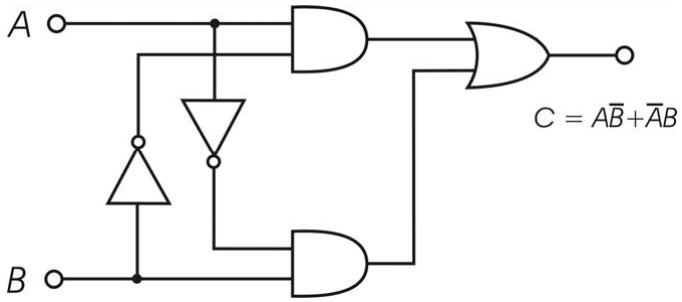
A	B	$C = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive OR gate symbol



1.9 Basic laws of Boolean algebra

Consider the logic function $C = A.\overline{B} + \overline{A}.B$, which has the following circuit diagram and truth table:



A	B	$A.\overline{B}$	$\overline{A}.B$	$C = A.\overline{B}+\overline{A}.B$
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

Note that the truth table is the same as that for the Exclusive OR gate, even though the circuit is different. Circuits that have the same truth table are called equivalent. In practice, circuits that are equivalent in the logical sense may have important differences (e.g. timing), but these will not worry us in this course.

1.10 Basic Boolean algebra relationships

Manipulating Boolean expressions is just the same as mathematical algebra. There are rules, and the aim is often to simplify a complex expression. The

following basic laws are all easily demonstrated using truth tables, but need to be learnt.

Simple properties

$A.0 = 0$

$A.1 = A$

$A.A = A$

$A.\overline{A} = 0$

$A + 0 = A$

$A + 1 = 1$

$A + A = A$

$A + \overline{A} = 1$

$\overline{\overline{A}} = A$

Note that some texts use alternative symbols:

OR	$A + B$	$A \vee B$
AND	$A.B$	$A \wedge B$

$X + Y = Y + X$ and $X.Y = Y.X$	Commutative property
$(X + Y) + Z \equiv X + (Y + Z)$ and $(X.Y).Z \equiv X.(Y.Z)$	Associative property
$X + 0 \equiv X$, $X + 1 \equiv 1$, $X.0 \equiv 0$, $X.1 \equiv X$	Dominance and Identity
$X.(Y + Z) \equiv X.Y + X.Z$	AND distributes over OR
$X + Y.Z \equiv (X + Y).(X + Z)$	OR distributes over AND
$X.X = X$	Basic redundancy

$\overline{X + Y} \equiv \overline{X}.\overline{Y}$ $\overline{X.Y} \equiv \overline{X} + \overline{Y}$	De Morgan's Laws

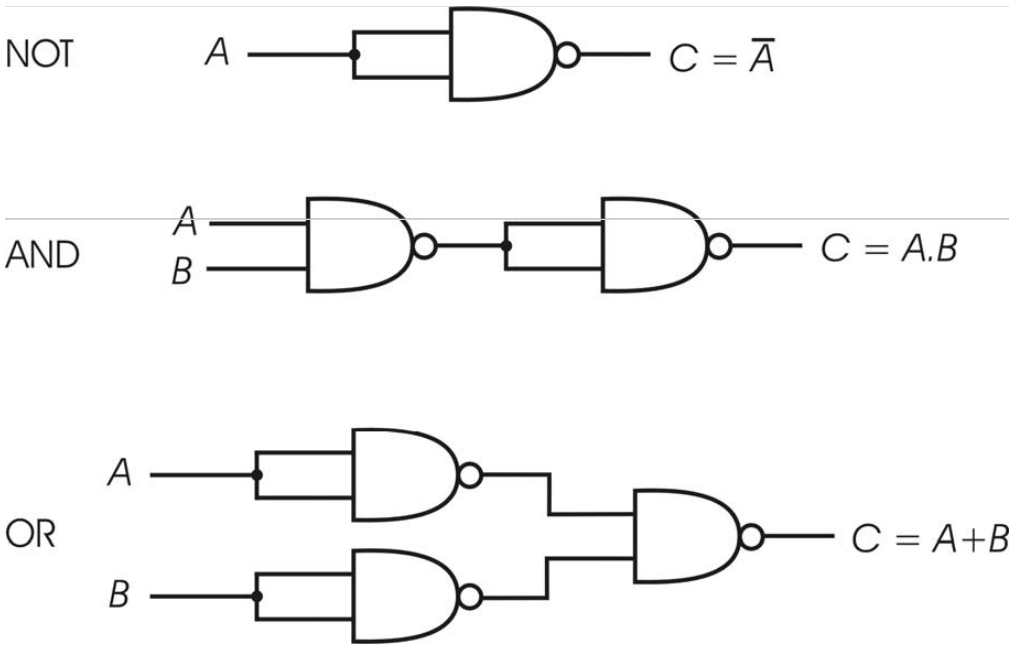
1.11 Universal gates

De Morgan's laws express a relationship between the three basic types of gate (i.e. AND, NOT, OR). It implies that only two types of gate are necessary, as

the third can be expressed in terms of the other two. The theorem says:

$$A + B \equiv \overline{\overline{A}.\overline{B}} \tag{1.4}$$

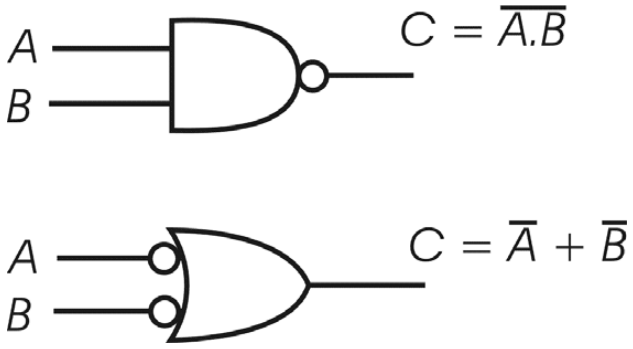
In other words, the OR function is equivalent to the NAND of NOT(A) and NOT(B). It turns out that all logic functions can be represented using NAND gates, and as such the NAND is sometimes referred to as the universal gate. The diagram below is an illustration of how the basic gates are derived from NAND gates only. This means that complex Boolean logic expressions can be implemented using NAND gates alone, which offers considerable advantages in terms of electronic design.



In a similar fashion, instead of using NAND gates, we could use NOR gates as the universal gate and build up all logic functions in terms of NOR gates only. The common switch between NOR and NAND functions is called DUALITY. A gate that performs a NAND operation in positive logic representation performs the NOR operation in negative representation and vice versa. Again, this can be proved by inspection of the relevant truth tables.

The distinction between positive and negative logic can be confusing. All it means is that for any design to be converted from positive to negative logic all that is required is to take the logic 1 and 0 levels and convert them to logic 0 and 1 values, respectively.

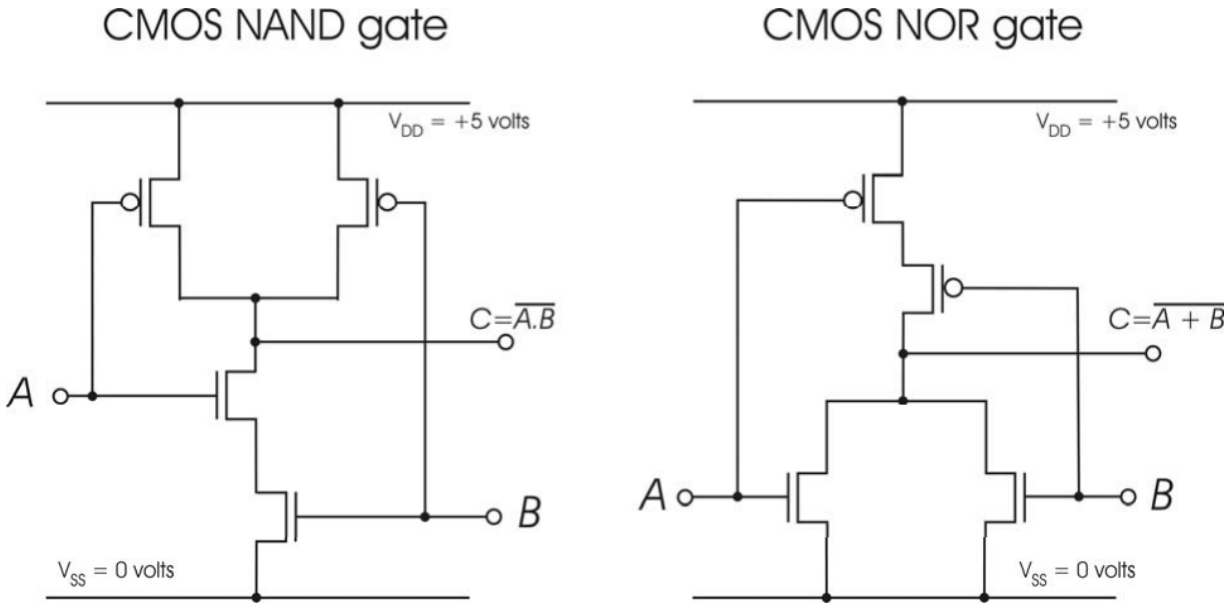
The two gates below are equivalent, with the circles at the input to the OR gates referring to the negative (or NOT) of the input logic.



1.12 CMOS implementation of digital logic

So far we have only looked at schematic symbols for the gates we have been considering, but of course these symbols represent electronic devices that behave in a characteristic way due to resistance and capacitance (and to a much lesser extent inductance). In this section we will examine how gates are made up of the CMOS transistor elements we have already discussed.

The next diagram shows circuit diagrams for CMOS NAND and NOR gates. Note that the basic function of CMOS logic is a NOT gate, in the sense that the very simplest circuit we could obtain would be an inverter comprising of a single input and two MOSFETs.



All other functions are obtained by placing more transistors either in parallel

or in series. The NAND and NOR gates have two input connections (A and B in this case) and a single output. The inputs are divided between the n and p type transistors. If the bottom gates are in series, then typically the top gates are in parallel, and vice versa. Consider the NAND gate:

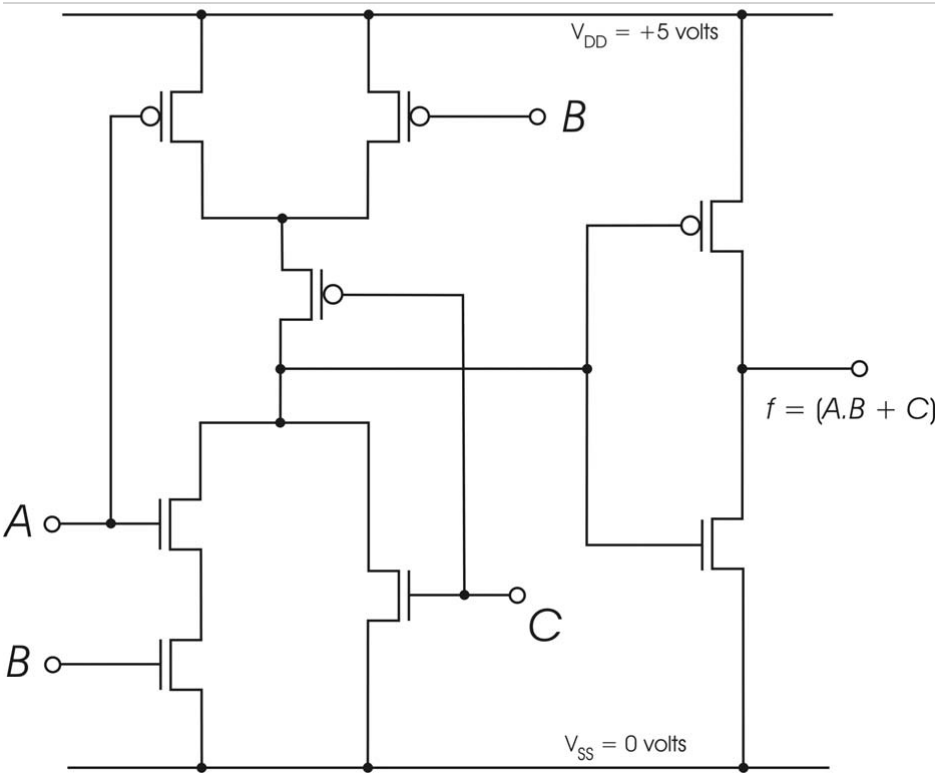
i. Suppose that inputs A and B are both logic 1. Then the lower transistors are conducting whereas both upper p-type transistors are open-circuit. The output C is therefore connected to the 0 V supply (logic 0).

ii. Suppose now that either or both of the inputs are logic 0. Then the output is connected to the V_{DD} supply rail and is represented as the logic 1 state.

The truth table for (i) and (ii) corresponds to the NAND functionality. It is left as an exercise for the reader to prove the NOR functionality for the second gate using similar arguments.

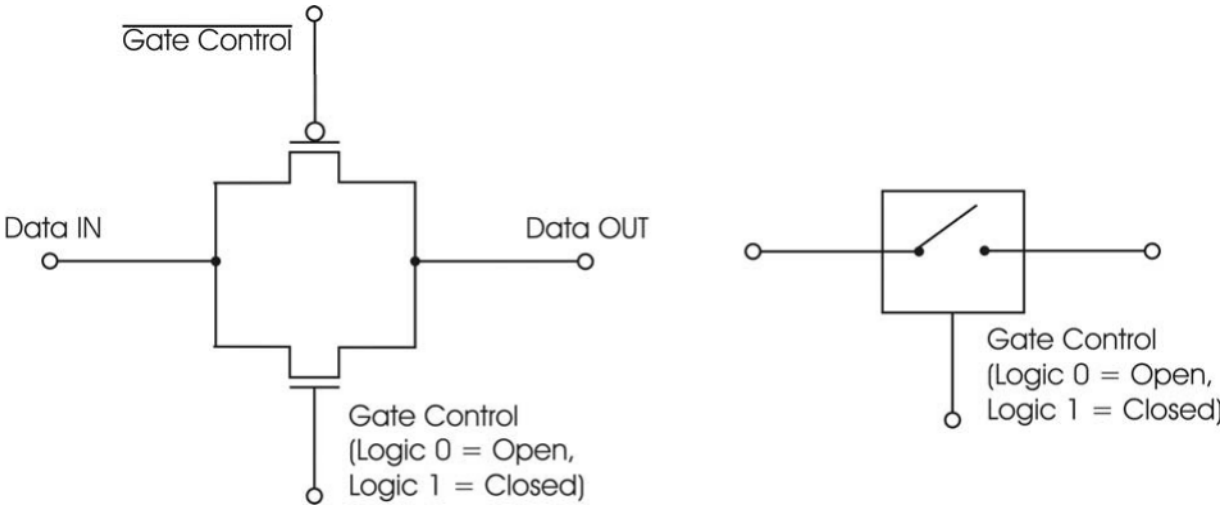
More complicated circuits can be built up by adding more transistors and by cascading CMOS sections. The diagram below is a typical example, with a single inverter CMOS element (NOT gate) as the output stage.

CMOS logic for (*A.B* + *C*)



1.13 Transmission gate

There are of course many other implementations of CMOS gates. Sometimes, the flow of data (i.e. a stream of logic values) needs to be controlled with something similar to a standard open or shut switch. The next figure shows the correct circuit for a simple CMOS switch. Surprisingly we need two transistors in a configuration that is called a CMOS Transmission Gate. It is exactly equivalent to a simple switch with a single gate control, such as a logic 0 opens the gate, and logic 1 closes it. Note that the Gate Control signal is inverted for the p-channel MOSFETs.



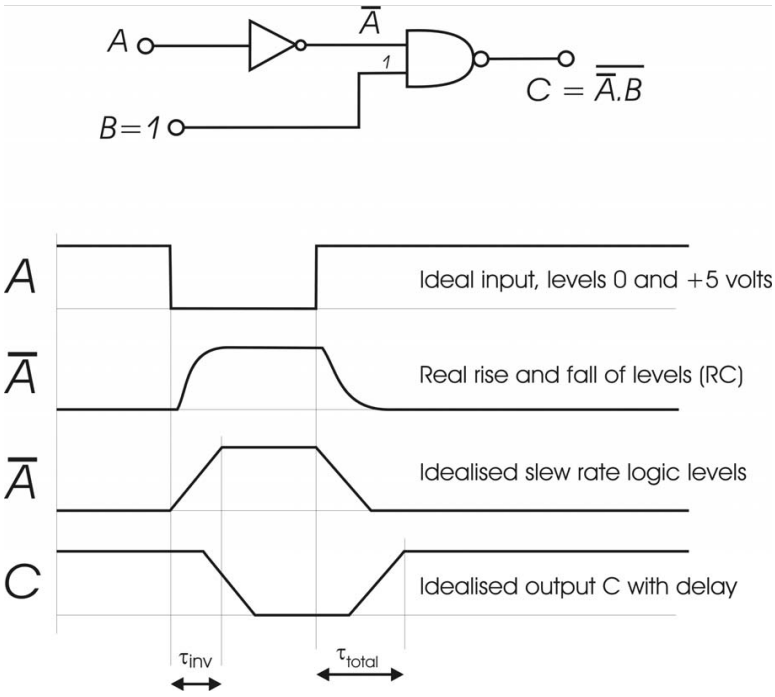
The operation is as follows; when the control signal is a logic 0 the bottom MOSFET is OFF (i.e. open circuit) and the top MOSFET receives a logic 1 signal as the gate voltage. It is therefore OFF as well. Overall, we see that there is no connection between the Data IN and Data OUT sections of the circuit and the switch is considered to be open (as in the illustration on the right). However, if the control signal is a logic 1, then the bottom MOSFET is ON and conducting. Likewise the top MOSFET is conducting due to the logic 0 state on its gate contact. The switch is now closed and data passes from the input to the output.

Both gates are necessary to ensure that the logic levels produced at the output preserve the correct voltage levels for the logic 0 and logic 1 states. It turns out that if only one gate was used then perfect switching of the output voltage levels would not be possible.

1.14 Real integrated circuit gates

In real (as opposed to ideal) circuits, there are delays in signal switching due to capacitances. There are capacitances associated with cables, tracks on printed circuit boards, metallisation on the surface of an integrated circuit and due to the charge storage effects associated with switching the gate contact on a MOSFET.

The next figure tries to illustrate this point. The A input is switched from logic 1 to logic 0 and back again, whilst the B input remains at the logic 1 state. The influence on the output C is shown in the timing diagram below the circuit.



The inverter has some delay due to various sources of capacitance (but in the best case equal to the CMOS gate capacitance of the order of picofarads) and resistance arising from the conducting tracks that connect the circuit together. This is shown in the timing diagram by the non-instantaneous rise of the form $1 - e^{-at}$ in the inverter output voltage labelled as $\overline{A}$.

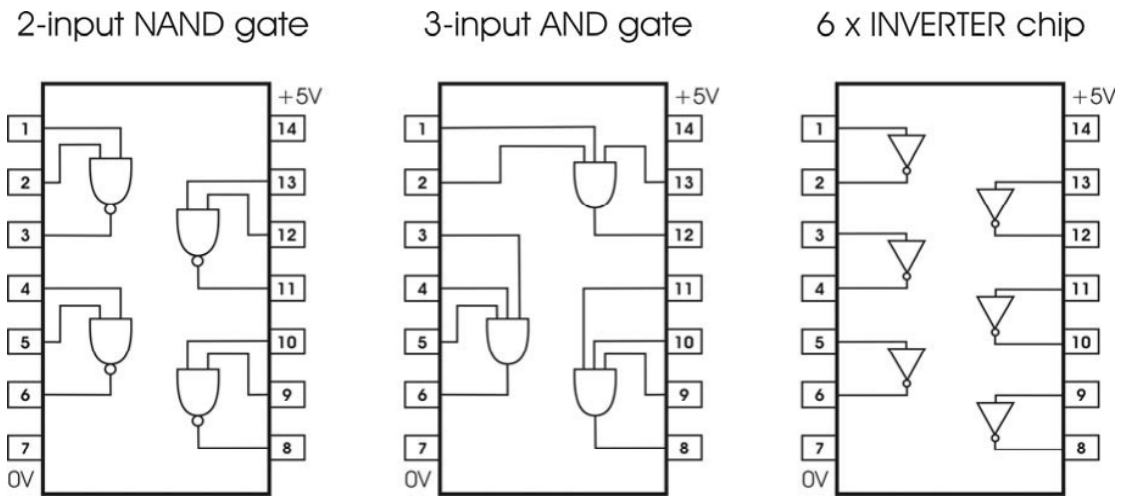
This non-exponential rise (and fall) of $\overline{A}$ represents the capacitive charging (and discharging) events that are seen in simple R-C circuits. Normally for timing diagrams this is idealised as a simple slope in the trace. The gradient

of the slope is called the slew rate (in volts per second) and a numerical figure for this parameter is often quoted on data sheets.

Also shown on the timing diagram is the characteristic delay time for the inverter τ_{inv}, which is typically a few nanoseconds. The output C is shown switching in response to the changing logic state at its inputs. The timing diagram also shows the total propagation delay, as τ_{total}.

Real gates are also influenced by the flow of current. If too many gates are connected to the output of a single NAND gate (for example), then the voltage levels might not be able to represent the logic 1 and 0 states accurately. This problem is characterised by a parameter called the **fan out**, an integer number giving the number of gates that can be connected to a single output before the logic levels become uncertain. The gate fan out must therefore not be exceeded.

When designing a circuit using basic gates laid out on a printed circuit board it is necessary to examine carefully the data sheets corresponding to the type of CMOS chips that are available. For interest, the figure below shows 3 types of simple logic integrated circuit chips, in a 14-pin DIL package (DIL stands for direct in-line).



Note that two pins are used to supply power to the chips. The other pins make connections to different logic gates, as shown. This sort of information (together with fan out, slew rate, current consumption, etc) is necessary in the

design of a printed circuit board.

2 Lecture B – Low level logic design

2.1 Simplification of Boolean expressions

Any realisation of a complex circuit design will be influenced by cost, either design costs (in terms of design hours) or fabrication costs (in terms of the number of chips or the amount of silicon space taken up within a single chip). These are important issues because all designs have to be commercially competitive. Also, customers have to have confidence in the designs. Circuits therefore need to be proven to work, and consequently have to be testable. There is a whole research area devoted to testing complex integrated circuit designs and there are specific rules and theorems that can be applied. Although design for testability is well beyond the limits of this course, the same motivation underpins the methods for reducing a complex design to its simplest form, namely the desire to prove that a circuit will work as expected.

When faced with a Boolean expression there are several techniques that can be employed for simplification, such as using the simple identities from the first lecture to rewrite the equations. Below are a few more identities, which it is advisable to learn.

Redundancy Theorem $X + X.Y = X.(1 + Y) \equiv X$

Race Hazard Theorem $X.Y + Y.Z + \overline{X}.Z \equiv X.Y + \overline{X}.Z$

Special Case of Race Hazard Theorem $X.Y + \overline{X}.Y = (X + \overline{X}).Y \equiv Y$

(You will come across the topic of hazards later in the course.)

Consider the following example of Boolean logic simplification by algebra.

$$
\begin{aligned}
& h(A,B,C,D) \\
& \quad = A.\overline{C} + A.B.\overline{C} + \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.C.\overline{D} \\
& \quad = A.\overline{C} + \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.C.\overline{D} && \text{since } X + X.Y \equiv X \\
& \quad = A.(1+\overline{B}).\overline{C} + \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.C.\overline{D} && \text{since } X \equiv X.(1+Y) \\
& \quad = A.\overline{C} + A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.C.\overline{D} \\
& \quad = A.\overline{C} + \overline{B}.\overline{C} + A.\overline{B}.C.\overline{D} && \text{since } X.Y + \overline{X}.Y \equiv Y \\
& \quad = A.\overline{C} + \overline{B}.\overline{C}.(1+A.D) + A.\overline{B}.C.\overline{D} && \text{since } X.(1+Y) \equiv X \\
& \quad = A.\overline{C} + \overline{B}.\overline{C} + A.\overline{B}.\overline{C}.\overline{D} + A.\overline{B}.C.\overline{D} \\
& \quad = A.\overline{C} + \overline{B}.\overline{C} + A.\overline{B}.\overline{D} && \text{since } X.Y + \overline{X}.Y \equiv Y
\end{aligned}
$$

Therefore

$$h(A,B,C,D) = A.\overline{C} + \overline{B}.\overline{C} + A.\overline{B}.\overline{D},$$

which is the simplest representation (i.e. it requires the fewest AND and OR operations) of the Boolean function h.

Taking this approach requires some skill (and practice). Later in this lecture, we will look at an alternative technique that relies on using a graphical method to achieve the same result.

2.2 Equivalence re-visited

Two functions f and g are said to be equivalent if their exhaustive truth tables lead to identical entries. For example, take the two functions,

$$f = \overline{a} + b \qquad \text{and} \qquad g = \overline{a}.b + a.b + \overline{a}.\overline{b} \tag{2.1}$$

We can prove that these are identical (sometimes written as $f \Leftrightarrow g$) using truth tables as follows:

a	b	f	$\overline{a}.b$	$a.b$	$\overline{a}.\overline{b}$	g
0	0	1	0	0	1	1
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	1	1	0	1	0	1

When comparing the column entries for both f and g we obtain the surprising result that the expressions are equivalent, such that $f \Leftrightarrow g$.

2.3 Standard forms of logic expressions

There is a standard form of expressing Boolean functions that has the Boolean variables connected with AND operators to form terms that are summed with OR operators. For example, consider the following standard form expression:

$$f(A, B, C) = \overline{A}.\overline{C} + A.\overline{B}.C + \overline{A}.B \tag{2.2}$$

Here the three variables are arranged into what are called "product" terms formed by the AND operator, and these product terms are then "summed" using the OR operator. (Although the terms "product" and "sum" are used, it is good to remember that these terms are only used because of the similarity in symbols to conventional algebra.)

This standard form is called the SUM-OF-PRODUCTS form and is very popular in Boolean expressions. It can be extended to form what is called the canonical sum-of-products term by expanding the expression to include all variables (or their complement) in each of the product terms.

For example, the next expression is in canonical form.

$$g(A, B, C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.B.C + A.\overline{B}.C \tag{2.3}$$

The product terms in the canonical sum-of-products expression are called *minterms*, and the variables are called *literals*. The minterms can be obtained either by using Boolean operations to expand an arbitrary function into its canonical form, or by using a truth table. Below is the truth table for the function $g(A, B, C)$:

A	B	C	g		
0	0	0	1	$\longleftarrow$	$\overline{A}.\overline{B}.\overline{C}$
0	0	1	0		
0	1	0	1	$\longleftarrow$	$\overline{A}.B.\overline{C}$
0	1	1	1	$\longleftarrow$	$\overline{A}.B.C$
1	0	0	0		
1	0	1	1	$\longleftarrow$	$A.\overline{B}.C$
1	1	0	0		
1	1	1	0		

Each row, when $g(A, B, C)$ is equal to logic 1, generates a new minterm, as this is when the OR operator would make $g = 1$ (or true). In this way the canonical sum-of-products form can easily be derived.

There is another standard form called the product-of-sums form, which has the variables summed using the OR operator and each term connected by the AND operator. This time the function f is in the product-of-sums form:

$$f(A, B, C) = (A + B + \overline{C}).(\overline{A} + C).(\overline{A} + \overline{B}) \tag{2.4}$$

Similarly, the function g is now given below in canonical product-of-sums form, with each sum term referred to as a maxterm:

$$g(A, B, C) = (A+B+\overline{C}).(\overline{A}+B+C).(\overline{A}+\overline{B}+C).(\overline{A}+\overline{B}+\overline{C}) \tag{2.5}$$

Each of the maxterms in the canonical product-of-sums form can easily be obtained from the truth table. For each entry with $g = 0$, take the complement of the variables and OR them together to form the corresponding maxterm. This is really just a quick way of applying de Morgan's theorem.

2.4 Karnaugh map reduction method

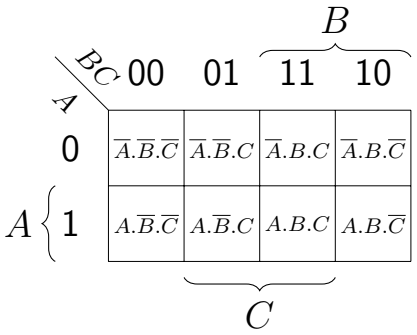
So far, we have attempted to reduce a complex Boolean expression into its simplest form using algebraic techniques. The simplest realisation of an expression is important to obtain, as this often represents the most cost-effective electronic circuit design (i.e. the least number of gates). However, algebraic manipulation requires a lot of skill and patience.

A very popular graphical method can be employed for a low number of inputs, called *Karnaugh map minimisation*. Note that for larger more complicated designs, circuit designers use more sophisticated software design tools (such as VHDL compilers), but this is beyond the scope of the course.

The Karnaugh map consists of a two-dimensional grid, with each element in the grid corresponding to one minterm in the canonical form of the sum-of-products expression. The arrangement of the minterms, either horizontally or vertically, allows for adjacent elements in the grid to differ only by one variable (which is complemented).

This is the key to how the Karnaugh map works and is achieved by labelling the grid according to a binary code called the *Gray code* (see later) which only differs by one bit in the code sequence.

The diagram below is an illustration of how the Karnaugh map is constructed. Each box has a unique code, for example the top-corner-right box has the code 010 for $\overline{A}.B.\overline{C}$. As a further point, notice that the top row corresponds to the case that $A = 0$ throughout.



Two types of labelling are used, with both given in the above diagram for illustration purposes. Only one labelling scheme is used in practice. The binary numbering around the edges is the Gray code labelling scheme. The brackets represent a reduced labelling format that shows over which squares the variable indicated is equal to a logic one – it is an exercise for the reader to confirm this. The two alternative types of labelling are given in the next diagram for the function: $f = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.C + A.B.C$

A \ BC	00	01	11	10
0	1	0	1	0
1	0	0	1	0

OR

			B	
	1	0	1	0
A	0	0	1	0
		C		

In the diagram we enter in to each box the logic value of the overall function, $f(A, B, C)$ corresponding to the values of each of the variables. So for example, $f = 1$, when $ABC = 000$, which is the top-left-corner box entry. Now check the other boxes!

2.4.1 Eliminating functions in Karnaugh maps

Let us consider some more examples of Karnaugh Maps. Either labelling scheme is fine – you can adopt the one you prefer, but for now both are shown. Shown below is a new map, corresponding to the function $f = \overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.B.\overline{C}$.

A \ BC	00	01	11	10
0	1	1	0	0
1	0	0	0	1

OR

			B	
	1	1	0	0
A	0	0	0	1
		C		

We notice that the top left-most two boxes both have logic 1 entries. This means that the function gives a logic 1 result when $ABC = 000$ (corresponding to $\overline{A}.\overline{B}.\overline{C}$) and when $ABC = 001$ (corresponding to $\overline{A}.\overline{B}.C$). Remember that adjacent boxes have been arranged so that only one variable changes to its complement. We draw a loop around both these 1's to indicate that they are neighbours; we call this *grouping*. This group shows that the variable C changes from a $\overline{C}$ to a C and yet the function f remains at a logic 1 state. In other words C does not make any difference to the function result when $AB = 00$. In which case C is eliminated from that particular minterm.

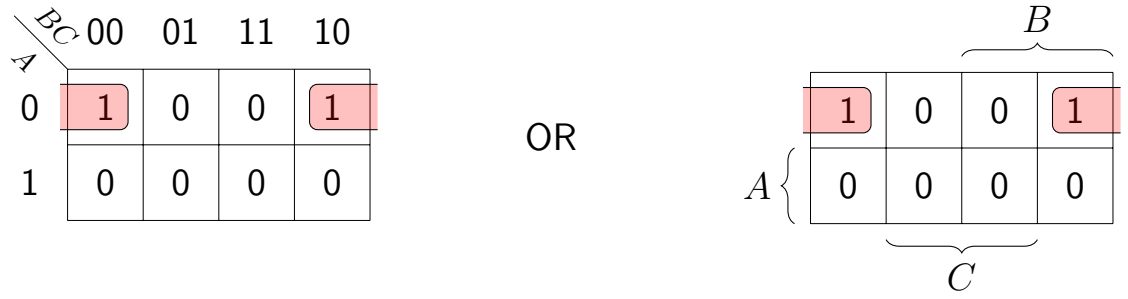
We can also group the single logic 1 shown in the bottom-right-corner box,

but in this case there are no adjacent boxes containing a logic 1. Once we have finished grouping we can write down the minterms that have been identified. The first grouping showed that the two minterms $\overline{A}.\overline{B}.\overline{C}$ and $\overline{A}.\overline{B}.C$ can be reduced by elimination to $\overline{A}.\overline{B}$, whereas the last minterm cannot be reduced.

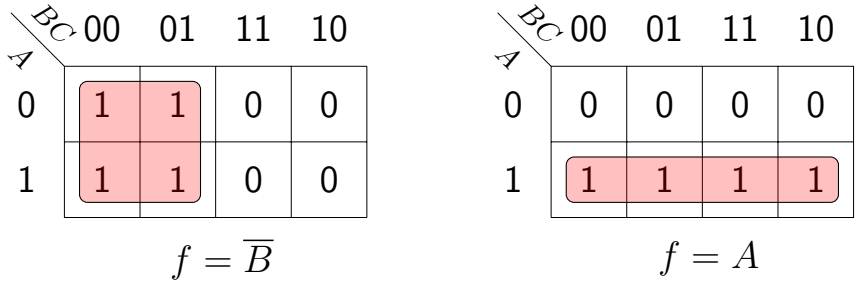
Overall, using this graphical method, we find that a simplified function for f is $f = \overline{A}.\overline{B} + A.B.\overline{C}$. The equivalence can be proved using a truth table.

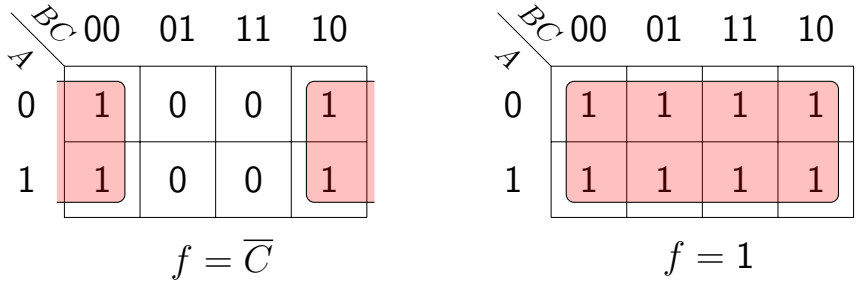
2.4.2 Grouping rules

Entries in the map can be grouped in numbers of $1, 2, 4, 8, \ldots$ (i.e. in powers of 2). As boxes around the perimeter still only have one variable changing at a time, it is possible to group around the edges of the map, as illustrated in the following example, which shows the function $f = \overline{A}.\overline{C}$.



Examples of different groupings are shown in the K-maps below. We need to enclose as **large a group as possible** to achieve the best simplification. The corresponding simplified function is found by considering which variables change logic state without changing the value of the overall function.



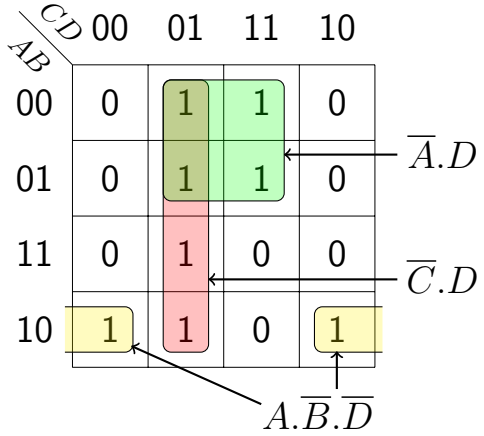


$f = \overline{C}$ $\quad$ $f = 1$

2.4.3 Karnaugh maps with more variables

The Karnaugh Map approach can be applied as a simplification tool for any number of variables, but it soon becomes cumbersome for more than 5 variables and different software tools are normally employed in such circumstances. Below is an example of a function with 4 variables, with the original function in canonical sum-of-products form given on the right of the map. Grouping has identified the simplified expression:

$$f = \overline{A}.D + \overline{C}.D + A.\overline{B}.\overline{D}$$

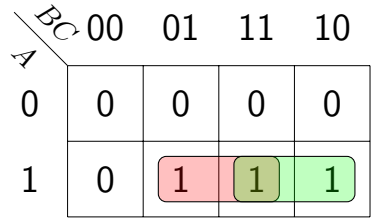


$$\begin{aligned} f(A,B,C,D) = {} & \overline{A}.\overline{B}.\overline{C}.D + \overline{A}.\overline{B}.C.D \\ & + \overline{A}.B.\overline{C}.D + \overline{A}.B.C.D \\ & + A.B.\overline{C}.D + A.\overline{B}.\overline{C}.\overline{D} \\ & + A.\overline{B}.\overline{C}.D + A.\overline{B}.C.\overline{D} \end{aligned}$$

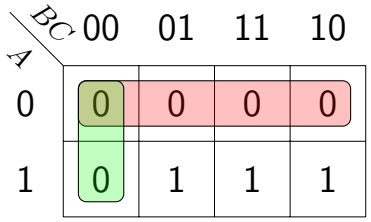
2.4.4 Simplification using 0's on Karnaugh map

Sometimes, once the map has been filled in, it becomes apparent that a simpler representation of the function can be achieved by grouping the 0's, instead of the 1's. This is illustrated below, for the same function showing alternative groupings. Grouping the 1's gives a function that is TRUE under certain conditions, whereas grouping the 0's provides a simplification that indicates when

the function is FALSE. Deciding which to adopt depends on the choice of gates available for the design, but in practice grouping 1's is more common.



OR



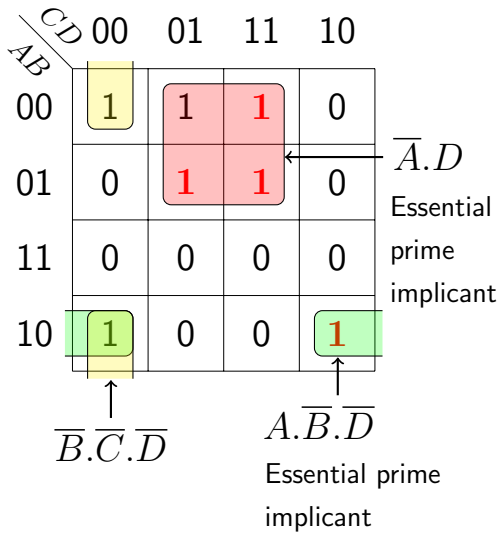
$f = A.B + A.C$ from 1's

$\overline{f} = \overline{A} + \overline{B}.\overline{C}$ from 0's

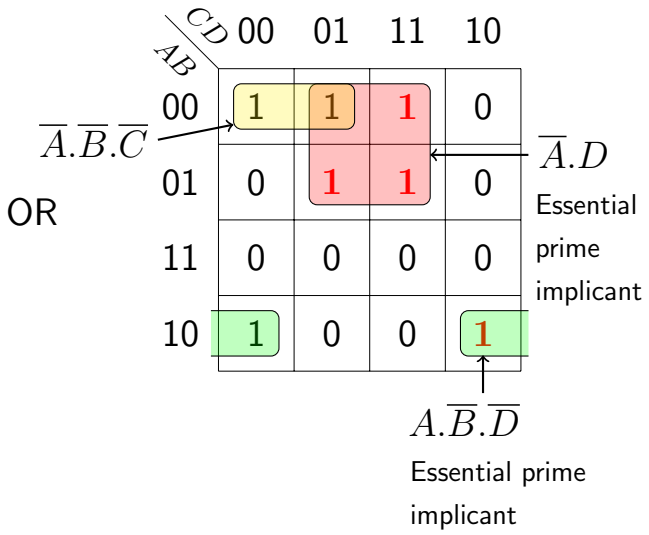
2.4.5 Identifying prime implicants

There are further conventions that have been developed in order to avoid difficulties associated with the electronic realisation of the circuit function (e.g. avoiding glitches in the output caused by timing delays, which are known as *race hazards*). The groups on the Karnaugh map are called *implicants*, with the largest groups being called the *prime implicants* for the function. Prime implicants that contain 1's that cannot be grouped in any other way are called *Essential Prime Implicants*, and the terms they represent are crucial to the correct realisation of the function.

Two alternative grouping strategies are shown in the diagram below. The essential prime implicants are in bold red font. A general strategy is to form the groups containing the essential prime implicants first and then to group the remaining 1's. There are options – for example, the top-left-corner grouping is different between the two cases shown below. This leads to a different function simplification, but both are correct. You will notice that the essential prime implicants are the same in both solutions.



OR



$$f = \overline{A}.D + A.\overline{B}.\overline{D} + \overline{B}.\overline{C}.\overline{D} \qquad f = \overline{A}.D + A.\overline{B}.\overline{D} + \overline{A}.\overline{B}.\overline{C}$$

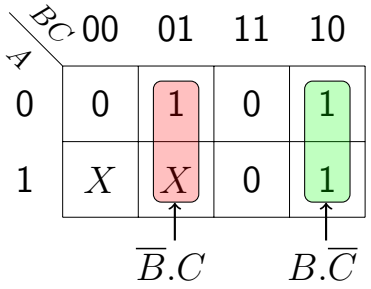
2.4.6 Incomplete variables, and don't cares

There are cases when not all variables have been linked to an outcome. So, we are uncertain as to what the function will do for certain combinations of input variables. This happens in designs that have not been fully specified, and we will come across examples of this later. The approach taken is to construct a truth table with an exhaustive list of all input variable conditions, and mark the unspecified function outputs as X's (which is called a don't care state).

The significance of this truth table entry is that we don't care what the output is under these conditions: it does not matter whether it is a logic 1 or logic 0 (usually because the design does not consider these cases). Such an example is given in the next figure, showing the truth table and the corresponding Karnaugh Map containing the don't cares. When grouping, we can consider an X as either a 1 or a 0, and form the groups using the strategies explained previously.

A	B	C	$f(A,B,C)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	X
1	0	1	X
1	1	0	1
1	1	1	0

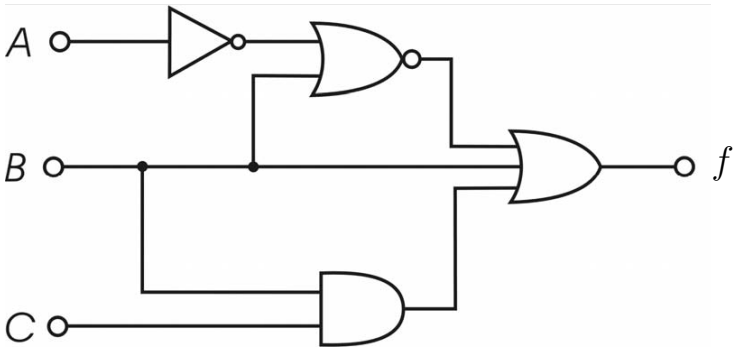
$\Longrightarrow$



$$f(A,B,C) = \overline{B}.C + B.\overline{C}$$

2.5 A circuit design example

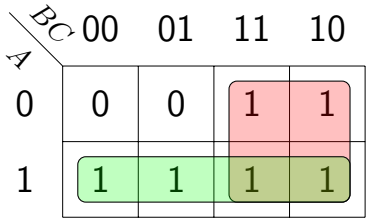
For the logic circuit shown below, find a simpler implementation that generates the same output.



Start with a truth table, then write down the Karnaugh Map, then simplify the function, as shown below.

A	B	C	$f(A,B,C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$\Longrightarrow$



$$f(A,B,C) = A + B$$

Alternatively we can use Boolean algebra to write (by inspecting the circuit and working from the top down):

$$f = \overline{\overline{A} + B} + B + B.C$$

Apply de Morgan's Theorem to the first term:

$$f = A.\overline{B} + B + B.C$$

Factorise this expression, making use of Redundancy theorem:

$$\begin{aligned} f &= A.\overline{B} + B.(1 + C) \\ &= A.\overline{B} + B \\ &= A.\overline{B} + B.(A + 1) \\ &= A.(\overline{B} + B) + B = A + B \end{aligned}$$

which gives the same result.

Selecting which method to adopt for function simplification depends a lot on your skills at algebra. The Karnaugh Map method has many attractions due to the fact that it is graphical and relies on a strict methodical approach, but it can be tedious for a large number of variables. Of course, you should know both methods.

3 Lecture C – Binary number representation

3.1 Counting in binary

Nature gave us 10 fingers and therefore we evolved a counting system based on units of 10. This counting system is so deeply rooted in our education that we often think there is something special about the number 10 – but there isn't. In fact, mathematicians throughout history have explored other alternatives.

Using the "base 10" (or decimal, or "radix 10") system, we make use of the following notation for representing a number n using m+1 digits:

$$n \equiv d_m, d_{m-1} \dots d_0$$

where d_i is one of the numbers $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ and

$$n = d_0 \times 10^0 + d_1 \times 10^1 + d_2 \times 10^2 + \cdots + d_m \times 10^m \tag{3.1}$$

An example is the number 598136, an integer. We interpret this number as follows:

$$598136 = 6 \times 10^0 + 3 \times 10^1 + 1 \times 10^2 + 8 \times 10^4 + 9 \times 10^5 + 5 \times 10^6$$

The same rules apply when considering a number of any radix or base, using the general form:

$$n = d_0 \times r^0 + d_1 \times r^1 + d_2 \times r^3 + \cdots + d_m \times r^n = \sum_{i=0}^{m} d_i \times r^i$$

where r is the "*radix" or base*.

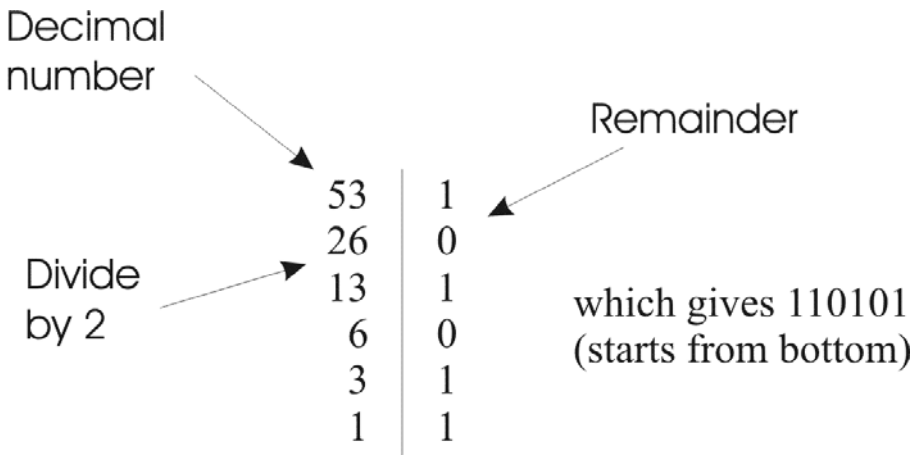
So, there is nothing special about binary representation, which is radix (or base) 2, and uses just digits 0 and 1. These numbers are ideal for logic systems, for which each variable can take the condition of being either TRUE (i.e. 1) or FALSE (i.e. 0). As an example, let's take the binary number 101011, of the form: $n = b_m b_{m-1} \dots b_0$ with $m = 6$ in this case. Converting from binary to decimal is straightforward:

$$\begin{aligned} 101011_2 &= 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 \\ &= 1 + 2 + 8 + 32 = 43_{10} \end{aligned}$$

Note that the subscript included in the answer is often added to indicate the radix, when it is not otherwise clear.

Most/Least significant bit: note that we often refer to the left-most bit (representing the highest power of 2) as the most significant bit (MSB) and the right-most bit (representing the lowest power of 2) as the least significant bit (LSB).

Conversion from decimal to binary is achieved by repeatedly dividing the number by 2 and writing down the remainder at each step. Keep going until it is no longer possible to divide, as illustrated in the figure below:



This works for any radix and so, for example, converting the decimal number 53 to base 6 gives 125_6.

3.2 Octal (8) and hexadecimal (16) numbers

The binary representation of a decimal number can lead to long strings of binary digits for large numbers. Two other bases are in common use for this reason, base/radix 8 and base/radix 16. The latter is very useful as groups of 4 digits are common in digital design (a group of 16 bits is often called a word, 8 a byte and 4 a nybble. . . .).

Base / radix 8, also called "octal":	$n_8 = \sum_{i=0}^{m} d_i \times 8^i$
Base / radix 16, also called "hexadecimal"	$n_{16} = \sum_{i=0}^{m} d_i \times 16^i$

Notice that in hexadecimal we need to represent digits greater than 9, hence the use of letters A (corresponding to 10) to F (corresponding to 15). Summarising the higher hexadecimal digits:

$$\mathsf{A}_{16} \equiv 10_{10} = 1010_2$$
$$\mathsf{B}_{16} \equiv 11_{10} = 1011_2$$
$$\mathsf{C}_{16} \equiv 12_{10} = 1100_2$$
$$\mathsf{D}_{16} \equiv 13_{10} = 1101_2$$
$$\mathsf{E}_{16} \equiv 14_{10} = 1110_2$$
$$\mathsf{F}_{16} \equiv 15_{10} = 1111_2$$

Working with octal and hexadecimal representation becomes easier when you consider the numbers in groups of 3 or 4 binary digits. For example:

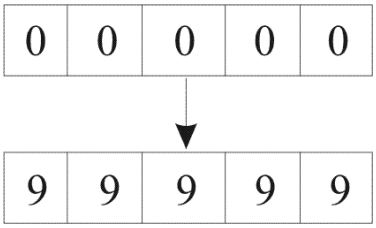
Binary $\rightarrow$ Octal :	$1111011101_2 \equiv 001111011101 = 1735_8$
Octal $\rightarrow$ Binary :	$217_8 \equiv 010001111 = 10001111_2$
Binary $\rightarrow$ Hex :	$1111011101_2 \equiv 001111011101 = 3\mathsf{DD}_{16}$
Hex $\rightarrow$ Binary :	$1\mathsf{AE}_{16} \equiv 000110101110 = 110101110_2$

3.3 Dealing with negative numbers

There are two common ways of representing negative numbers in the binary representation (2's complement and offset binary). The default assumption is that a binary number is positive UNLESS it is known that a negative number is being represented. If the computer architecture has been designed to implement negative numbers, then the most significant bit is usually assigned to be the "sign bit" and a number is *negative if its most significant bit is a* 1.

3.3.1 2's complement

To understand the 2's complement representation, consider first an example using the 10's complement representation. Imagine what would happen to a car milometer set to zero if the car was driven backwards exactly 1 mile:



Since the car has in fact gone backwards the distance travelled could be considered to be negative. But the mileometer has no way to represent the negative. Hence $99999 = 10^5 - 1$ is the ten's complement representation of -1 for 5-digit decimal numbers ($10^n - 1$ where $n = 5$).

In 2's complement form, a negative binary number $-b$ is represented by

$$2^n - b \tag{3.2}$$

where n is the number of bits used, which is set by the length of the binary representation employed within the computer. Usually n is chosen to be one of 8, 16, 32 or 64 (on the increase as chip technology improves).

For an 8-bit number, -1 is represented as $2^8 - 1 = 11111111_2 = \mathrm{FF}_{16}$. For a 16 bit number -1 is represented as $2^{16} - 1 = \mathrm{FFFF}_{16}$. As noted earlier, in the binary representation of "signed" numbers using 2's complement, the number is negative if the leading digit is a 1.

Thus, in the 2's complement form, the 4-digit binary number, 1011, is a negative number and its absolute value is: $2^4 - (1011) = 2^4 - 11_{10} = 5$. Note that $+5$ is represented by 0101, whereas -5 is represented by 1011. From this, it can be seen that there is a very simple way to obtain the 2's complement of a binary number, which is to invert all the bits and then add one.

To check: 0101 (i.e. $+5$) $\rightarrow 1010 + 1 \rightarrow 1011 = -5$, and it works going the other way as well.

This can be proved as follows. Note first that $2^n - 1$ generates a binary number with all ones. By inspection (but you should demonstrate this in the tutorial example), we recognise therefore that $(2^n - 1) - b$ will invert all the bits in b, i.e. creating the one's complement of b. We therefore have:

$$(2^n - 1) - b = \overline{b} \quad \Longrightarrow \quad 2^n - b = \overline{b} + 1 = \text{2's complement of } b.$$

Another way to look at 2's complement is to consider how it's encoded. For an ordinary binary number with 8 bits we write:

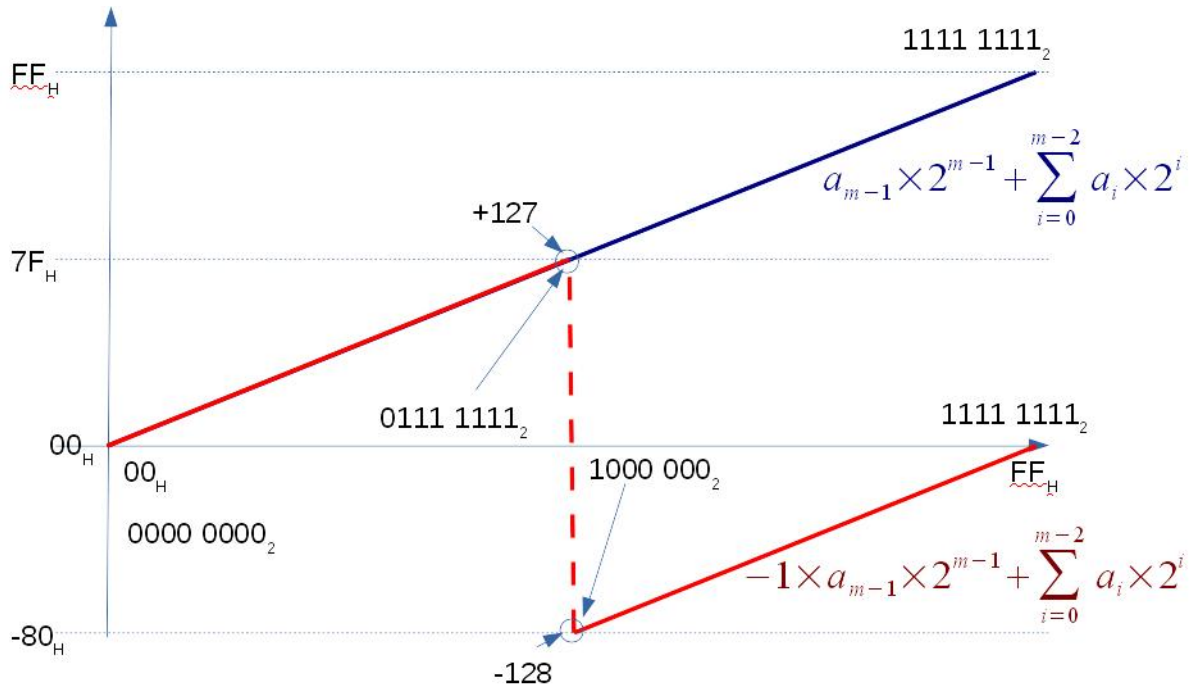
$$n = \sum_{i=0}^{7} d_i \times 2^i = d_7 \times 2^7 + \sum_{i=0}^{6} d_i \times 2^i$$

Where the MSB is separated to compare to the next equation for 2's complement. The 2's complement would be

$$n^{(2's)} = -1 \times d_7 \times 2^7 + \sum_{i=0}^{6} d_i \times 2^i,$$

i.e. if the MSB is 1, we subtract 2^7 from the number.

It's useful to show the 2's complement representation compared to 'normal' positive binary. The following graph compares them. The horizontal axis can be considered the conventional value of the binary digits, whereas the vertical axis shows the represented number. You can see that for the lower half of the numbers both 2's complement and standard binary agree, but, when the binary MSB becomes 1 the 2's complement is now negative.



3.3.2 Offset binary (or Excess n)

If a constant offset, c, is added to all numbers, then we have a representation given by:

$$b + c = b_{\text{offset}} \tag{3.3}$$

This is similar to shifting the origin. Consider an 8-bit offset binary representation where c = 128 (or 80_{16}), then:

$$\begin{aligned} 0_{10} &= 80_{16,\text{ offset}} \\ -128_{10} &= 00_{16,\text{ offset}} \\ 127_{10} &= \text{FF}_{16,\text{ offset}} \end{aligned}$$

Thus, the binary number 0 is considered to represent $-c$ and the binary number c to represent zero. This representation is used in hardware such as analogue-to-digital converters to represent negative as well as positive voltages.

3.4 Introducing binary codes

Binary codes are used to encode binary data for a variety of reasons; for example, ASCII codes are a very common means of communication between a computer and its peripherals (keyboard or printer). By introducing extra bits (redundancy) in the code, transmission errors (due to a "noisy" channel of communication, for example) can be corrected. Other forms of codes, used for secure data transmission, introduce dependency rather than redundancy.

The sections that follow consider the most useful binary codes.

3.4.1 Binary Coded Decimal (BCD)

Some operations are more efficient if each decimal digit is represented individually by its binary code – despite this being an inefficient use of memory. Binary Coded Decimal is used for storing decimal digits as in a telephone directory, or for converting data into an appropriate form for a decimal display (such as a seven-segment LED display).

Example:

$$9257 = \overbrace{1001}^{9}\ \overbrace{0010}^{2}\ \overbrace{0101}^{5}\ \overbrace{0111}^{7}$$

3.4.2 ASCII

ASCII (American Standard Code for Information Interchange) codes are used to represent alphanumeric characters, primarily for communication between a computer and its keyboard or printer. The character set for the standard ASCII code is shown below:

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

Code values are assigned to characters consecutively in the order in which the characters are listed above (row-wise), starting from 32 (assigned to the '!' character) and ending up with 126 (assigned to the tilde character '$\sim$'). Positions 0 through 31 and 127 are reserved for control codes. ASCII was first used to drive displays like terminals and these codes did things like moving the cursor backwards, linefeed, or sounding the 'bell'.

Examples of ASCII code

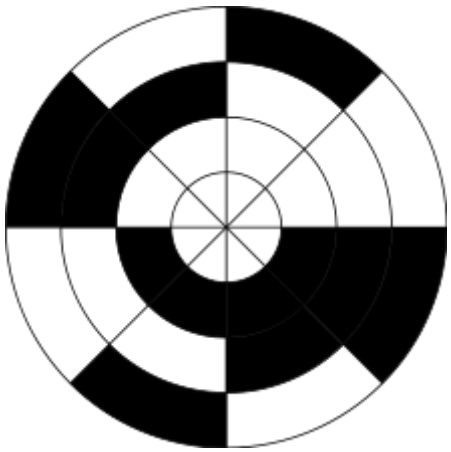
Upper case	'A' $= 41_{16}$, 'B' $= 42_{16}$, ... 'Z' $= 5\text{A}_{16}$
Lower case	'a' $= 61_{16}$, 'b' $= 62_{16}$, ... 'z' $= 7\text{A}_{16}$
Digits	'1' $= 31_{16}$, '2' $= 32_{16}$, ...

Modern systems used more advanced character encoding, but they follow similar principles to ASCII.

3.4.3 Unit distance codes

Consider a shaft encoder connected to a motor to give position feedback. If such an encoder uses 3 bits to represent angular position (a resolution of $45°$),

consider what happens as the position crosses from 179° to 181°. The code should change from 011 to 100. However, the manufacturers will not be able to ensure that all the lines on the encoder are accurately aligned and there will be a short time in which all the bit values will be uncertain and/or changing. Thus any 3-bit number could be generated at the boundary between 011 and 100.



A 3-bit shaft encoder disc – optical readout

This problem may be avoided by using a unit-distance code (the distance between two n-bit binary numbers is measured as the number of binary digits that are different) in which only one bit at a time is allowed to change (hence unit distance).

3.4.4 Gray code

A popular unit-distance code is the *Gray code*, which is often referred to as a *reflected binary code*. The code is built up by starting with the sequence $00, 01, 11, 10$ (this is the same sequence used in Karnaugh maps). The next most significant bit is then set and the code is reflected (i.e. reversed) – $10, 11, 01, 00$ – resulting in a 3-bit code. The fourth bit is then set and the previous sequence reflected about this point. This is repeated until an n-bit code is obtained.

Example: 4 bit Gray code

Decimal	Binary	Gray	
0	0000	0000	– start
1	0001	0001	
2	0010	0011	
3	0011	0010	– reflect about 3rd bit
4	0100	0110	
5	0101	0111	
6	0110	0101	
7	0111	0100	– reflect about 4th bit
8	1000	1100	
9	1001	1101	
10	1010	1111	
11	1011	1110	
12	1100	1010	
13	1101	1011	
14	1110	1001	
15	1111	1000	– and so on ...

A useful feature of the Gray code is that it is possible to convert directly from binary into Gray code.

Let g_i denote the ith bit of the Gray code representation of a binary number $b_n b_{n-1} \ldots b_0$. Then we can write:

$$\begin{aligned} g_i &= b_i \oplus b_{i+1} \quad \text{for } i = 0, \ldots, n-1, \\ g_n &= b_n \end{aligned} \tag{3.4}$$

where $\oplus$ is the Exclusive OR operation: $A \oplus B = A.\overline{B} + \overline{A}.B$.

3.5 Error detection

In real applications, noise and/or interference can corrupt binary data (i.e. a '1' can become a '0' and vice-versa) when data is transmitted or written to a storage medium. Error detection is particularly important for storage media when, for example, stored data is recovered from a hard drive or another

storage medium, such as DVD/CD ROM. There are two options for dealing with corrupted data:

- In the simplest case, we need to know that an error has occurred. This can be done with a parity bit (see below).
- With sophisticated encoding of the data (e.g. Hamming codes), some errors can be corrected. A classical example is the transmission of data from a spacecraft. Codes with a huge amount of redundancy can be transmitted which will allow the very weakly received data to be reconstructed.

3.5.1 Use of parity

One way of using redundancy is to define a parity bit (usually the most significant bit in an 8-bit byte). **Even** parity means that the total number of 1's in the 8 bits (including the parity bit) is even, whereas **odd** parity means that the total number of 1's is odd.

Computers often communicate with devices such as printers using 7-bit data (such as an ASCII code) and one parity bit. A system must expect even or odd parity as the normal situation and then detect an error if the parity bit is wrong. There is of course no way of telling which particular bit is wrong. Furthermore, two bit errors will go undetected when using a single parity bit for error checking.

The parity bit can be generated as follows. Consider two bits, $b_1 b_0$. We can determine if they represent an even or odd sum of bits by using the Exclusive OR function:

$$\begin{aligned} b_1 \oplus b_0 &= 1 \text{ if there is only one bit set (i.e. equal to one)} \\ &= 0 \text{ if there is an even number of bits set (0 or 2)} \end{aligned} \tag{3.5}$$

This can be generalised to any number of bits by cascading the Exclusive OR gates.

Example: Assuming even parity for a 4-bit binary number, how should the decimal value 4 be represented as a binary code?

Since $4_{10} = 0100$ and we are using even parity, the parity bit needs to be set. In hardware, the parity bit b_3 would be obtained from:

$$b_3 = b_2 \oplus b_1 \oplus b_0$$

and so the final code would be 1100.

4 Lecture D – Binary arithmetic

4.1 Addition

With 2's complement representation, it is possible to perform both addition and subtraction using the same hardware. In this lecture the hardware necessary for this will be introduced and then the concepts will be extended to include multiplication and the representation of fractions in computer memory.

4.1.1 The half-adder

The simplest possible addition is to add together two binary bits, i.e. $S = A + B$, (where $+$ means addition here and not a logical OR) where A and B can be 0 or 1. The set of possible results is:

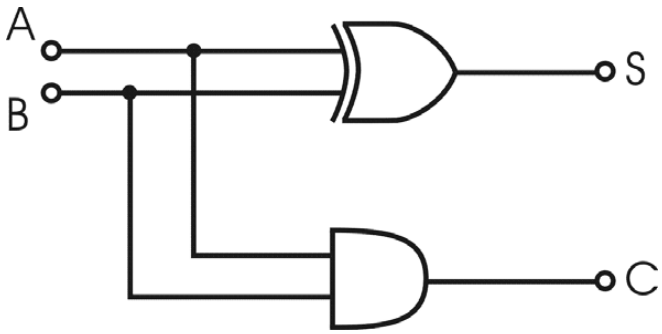
A	B	$S = A + B$
0	0	0
0	1	1
1	0	1
1	1	0 Carry 1

The above truth table describes how the output of the addition changes for all possible inputs. Note there is an *extra bit* produced by the addition shown in the bottom right cell. This bit is called the *Carry*, and is exactly the same as the "Carry over" produced, for example, when adding 8 to 6 in decimal (i.e. the next power, or higher digit, is incremented by one).

It can be observed that the truth table is identical to the truth table for the logic function Exclusive OR logic function. We can write:

$S = A \oplus B$ and $C = A.B$ (i.e., the logical "AND" of A and B)

The circuit description for the implementation of these functions, which is known as the half-adder circuit, is shown below:



4.1.2 The full adder – using carry-in

The half-adder allows two bits A and B to be added together. Suppose that these two bits happen to be in the middle of a group of binary bits. What is then missing is the ability to deal with a carry-over from less significant bits (i.e. bits to the right). This is easily remedied by enabling a *carry in bit*, as well as a carry out. In the general case the ith bit would be:

$$S = A \oplus B \oplus \text{Carry-in}$$

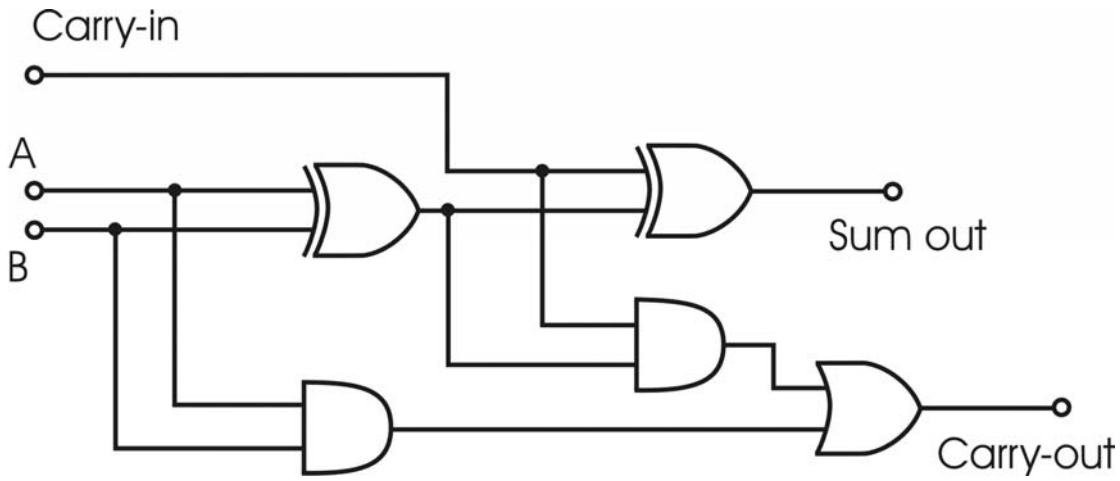
The truth table for a full adder, which has A, B, and Carry-in (C_{in}) as inputs and S, and Carry-out (C_{out}) as outputs is as shown here:

A_i	B_i	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

It is simple to show, from the truth table, that

$$\begin{aligned} S &= A \oplus B \oplus C_{\text{in}} \\ C_{\text{out}} &= A.B + (A \oplus B).C_{\text{in}} \end{aligned} \tag{4.1}$$

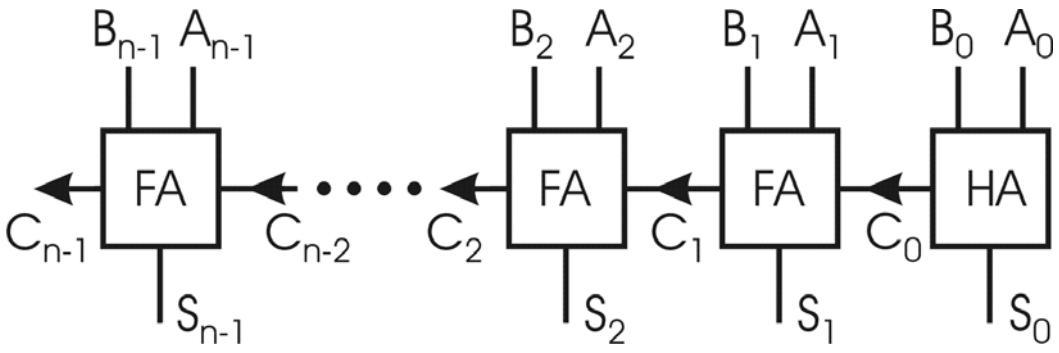
Hence the combined logic circuit becomes:



4.1.3 Ripple addition

So far, we have added single bits together, taking into account the need to include carry bits. Real numbers, however, are represented by a combination of bits and therefore it is necessary to develop a strategy for dealing with groups of bits that form words. The least significant bit (LSB) can be handled conveniently using the half adder circuit described above, but all the other bits must be able to handle carry overs from the adjacent lower bits. So, we need a series of full adders to achieve this. The carry from each bit addition cascades along the chain as illustrated below.

One problem with this circuit (as indicated in its name) arises because it takes time for the effects of the carry bits to "ripple" through from the LSB end to the MSB. This means that the total sum only becomes correct after a fixed amount of time, which increases in proportion to n, the number of bits.



Ripple-Carry adder (using Full and Half Adders)

4.1.4 Look-ahead carry

Sometimes it is important to add two n-bit numbers (or words) with minimal delay. We can perform addition much faster than is offered by the ripple-carry adder if we compute the carry ahead of the sum. This may be important when detecting "overflow" in addition, for example.

Overflow occurs when the addition of two n-bit words produces a result which requires $n+1$ bits; for example, the addition of two 8-bit numbers can produce a sum that can only be properly represented using 9 bits. (In this case, the 9th bit can be recognised as the carry bit beyond the MSB of the 8-bit numbers). Detecting the result of the MSB carry-over could be more important than the addition itself, and a circuit can be designed that calculates the carry in advance. This is called "Look-Ahead" and the combinational logic required to do this is given below. Note that the addition is not performed any faster, just the computation of the final carry bit.

Consider the problem of generating C_i, the Carry-out from bits A_i, B_i and C_{i-1}. Three conditions govern the generation of a Carry-out:

1. $A_i = B_i = 1$	$\Longrightarrow$	$C_i = 1$	... Carry-out is generated
2. $A_i \neq B_i$	$\Longrightarrow$	$C_i = 1$ if $C_{i-1} = 1$	... Carry is propagated
3. $A_i = B_i = 0$	$\Longrightarrow$	$C_i = 0$	... Carry-out isn't generated

Let G_i denote carry generate, $G_i = A_i.B_i$ (remember that the "dot" means logical AND), and let P_i denote carry propagate, $P_i = A_i + B_i$, then[1]

$$C_i = G_i + P_i.C_{i-1}.$$

This is a recurrence relation giving the current Carry-out in terms of the ith bits of A and B and the Carry-out from the previous stage. We can therefore write down a logic expression that will predict the overall carry $C_{\text{out}} = C_{n-1}$ when two n-bit numbers are added together:

$$C_{n-1} = G_{n-1} + P_{n-1}.\Big(G_{n-2} + P_{n-2}.\Big(\cdots.\big(G_1 + P_1.(G_0 + P_0.C_{-1})\big)\cdots\Big)\Big) \tag{4.2}$$

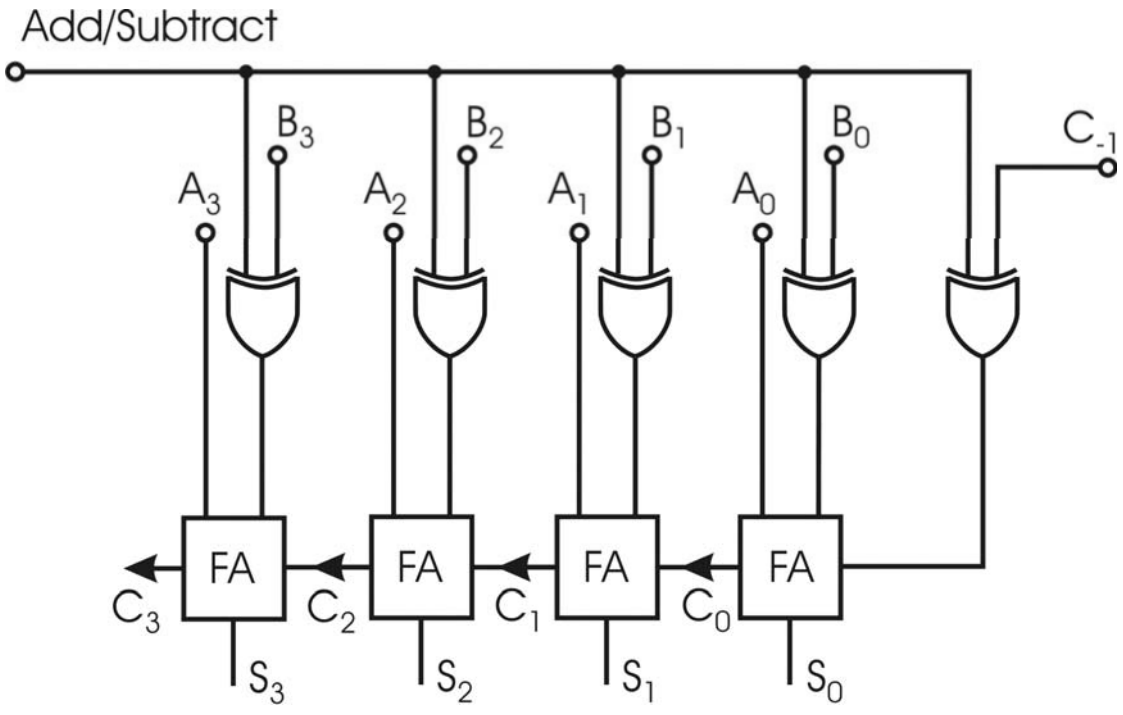
[1] It doesn't matter if we use OR or EX-OR to define P_i (i.e. $P_i = A \oplus B$) since $G_i = A_i.B_i$

This can be implemented in a logic circuit. (Note that we have included a Carry-in C_{-1} to allow addition of longer numbers in n-bit chunks.)

4.2 Subtraction

Implementing subtraction can be achieved using the same hardware as that used for addition. This is possible due to the 2's complement representation of binary numbers. To obtain $A - B$, we can compute $A+(B$ in 2's complement form).

We have already seen that the 2's complement form of a number can be generated by inverting all the bits and adding a binary "1" to the result. Achieving the addition of "1" is easily obtained by using the carry-in line, as shown below:



4-bit Adder/Subtractor

When the Add/Subtract line is equal to "0" (i.e. add) the EX-OR gates do not modify the value of B. When this line is "1", however, the action of the EX-OR gates is to invert all the individual bits of B (thus generating the 1's complement of B). The Carry-in to the least significant bit must be zero when the subtraction is executed, so the addition of "1" (completing the 2's

complement of B) is performed by the action of the EX-OR gate furthest to the right (i.e. Add/Subtract$\oplus C_{-1} = 1 \oplus C_{-1} = 1$ since we must have $C_{-1} = 0$ when performing subtraction).

4.2.1 Carry and Overflow flags

The Carry bit indicates whether or not the sum of two binary words creates a "Carry over" from the MSB. But this alone is insufficient when signed arithmetic is implemented. When two binary numbers are being added using 2's complement representation, a case may arise when the sum of two positive numbers might incorrectly be interpreted as a negative number (which in 2's complement representation would be identified as a binary number with the MSB equal to 1). Similarly, the sum of two negative numbers might be incorrectly displayed as a positive number (i.e. a 0 for the MSB). These errors are a consequence of the fact that the result of the addition cannot be adequately represented by the limited number of bits available, and in this case an overflow must be indicated. For an n-bit binary number, the largest positive number that can be represented is equal to $2^{n-1} - 1$ and the most negative number is -2^{n-1}

As an illustration, consider the problem of adding 65 to 65, using 8-bit binary numbers with the 2's complement representation:

$$\begin{array}{rr} & 01000001 \\ + & 01000001 \\ \hline & 10000010 \end{array} \qquad \begin{array}{rr} & 65 \\ + & 65 \\ \hline & -126 \end{array}$$

The correct sum (i.e. 130) has a 1 in the MSB, and is therefore incorrectly interpreted as -126. We need to indicate the fact that an overflow has occurred. Within the Arithmetic Logic Unit (ALU) in a microprocessor, there is a specific register that holds a series of bits that indicate the "condition" of the sum. These condition bits are called flags and they are useful in a computer program because decisions can be made in the code based on the values of these flags.

As well as the C flag (Carry), there is the V flag (Overflow). Determining the V flag is straightforward given the arguments above. For the case of $C = A + B$, Overflow has the logic description (for the case of addition of two n-bit numbers):

$$V = A_{n-1}.B_{n-1}.\overline{C}_{n-1} + \overline{A}_{n-1}.\overline{B}_{n-1}.C_{n-1} \tag{4.3}$$

4.3 Multiplication

Integer multiplication may be achieved by successive shifts and adds, these being the basic operations of long multiplication. Consider the multiplication of two positive binary numbers $C = A \times B$. Write B as

$$B = \sum_{i=0}^{n-1} b_i \times 2^i$$

where b_i are the bits of B and are either 1 or 0. Then

$$C = A \times B = A \times \sum_{i=0}^{n-1} b_i \times 2^i = \sum_{i=0}^{n-1} [(A \times 2^i) \times b_i]$$

However, the bits b_i are either one or zero and thus the sum reduces to:

$$C = \sum_{i=0, b_i=1}^{n-1} (A \times 2^i) \tag{4.4}$$

Multiplication by 2^i corresponds to shifting left by i places and replacing the right-most vacated bits by zeros. Hence the above multiplication may be carried out using the following operations:

1: Load the n-bit number A into a $2n$-bit memory location
2: Load the n-bit number B into an n-bit memory location
3: Clear another $2n$-bit memory location and allocate it to C
4: Set a counter k to 0
5: **while** $k < n$ **do**
6: **if** b_i is equal to 1 **then** add the contents of A to C
7: **end if**

8: Shift A left one place and add 1 to k
9: **end while**

This is an example of *pseudo-code*, i.e. a free description of a sequence of operations that together perform the multiplication $C = A \times B$. Implementing this in hardware is of course possible, but beyond the limits of this course. Note also that this simple algorithm does not work for signed arithmetic, i.e. when using 2's complement representation.

4.4 Fixed-point arithmetic

Numerical computation would not be very useful if it were not possible to represent fractions with binary numbers. After all, this is how we represent and manipulate numbers using decimal notation. For example, to do numerical calculations with a number such as $\pi = 3.141592653589793\cdots$ we need to introduce a finite-precision approximation. Thus, representing π to an accuracy of 8 decimal places ($\pi \approx 3.14159265$), we can write

$$\begin{aligned} 3.14159265 = 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3} + 5 \times 10^{-4} \\ + 9 \times 10^{-5} + 2 \times 10^{-6} + 6 \times 10^{-7} + 5 \times 10^{-8}. \end{aligned}$$

The same approach works for binary numbers. For example, with an accuracy of 18 bits, we get

$$\begin{aligned} &11.0010\,0100\,0011\,1111 \\ &\quad = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} \\ &\qquad + 0 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8} \\ &\qquad + 0 \times 2^{-9} + 0 \times 2^{-10} + 1 \times 2^{-11} + 1 \times 2^{-12} \\ &\qquad + 1 \times 2^{-13} + 1 \times 2^{-14} + 1 \times 2^{-15} + 1 \times 2^{-16} \\ &\quad = 2 + 1 + \tfrac{1}{8} + \tfrac{1}{64} + \tfrac{1}{2048} + \tfrac{1}{4096} + \tfrac{1}{8192} + \tfrac{1}{16384} + \tfrac{1}{32768} + \tfrac{1}{65536} \\ &\quad = 3.14158 \end{aligned}$$

It is important to be careful about the position of the "binary" point however, as computers normally have a fixed number of bits in order to represent each

binary number. There are two clear strategies, either have a fixed position for the point or implement a procedure that allows it to move.

4.4.1 Fixed-point addition

When adding A and B to obtain C, we can essentially use the same hardware we use for adding integer binary numbers. One could think that both A and B can be shifted left by exactly the same number of times as there are bits representing the fractional part of the number. (Remember that shifting a binary number by one place to the left is equivalent to multiplication by 2.) This temporarily removes the influence of the binary point and allows addition to proceed as before. Equivalently, one could just remember where the binary point was placed, add the bits as for integer numbers, then replace the binary point.

Hence (noting that the dot here is the binary point and $+$ represents addition):

$$C_{\text{integer-part}}.C_{\text{fractional-part}} = A_{\text{integer-part}}.A_{\text{fractional-part}} + B_{\text{integer-part}}.B_{\text{fractional-part}}$$

becomes

$$C_{\text{integer-part}}.C_{\text{fractional-part}} = (2^{-f} \times A') + (2^{-f} \times B') = 2^{-f} \times (A' + B')$$

where A' and B' are the binary numbers with the binary point removed. it is assumed that f is the number of bits used to describe the fractional part of the number. Therefore the addition part of the operation has not changed; *the result just needs to be shifted back the same number of bits*.

4.4.2 Fixed-point multiplication

Applying the same concept gives:

$$C_{\text{integer-part}}.C_{\text{fractional-part}} = A_{\text{integer-part}}.A_{\text{fractional-part}} \times B_{\text{integer-part}}.B_{\text{fractional-part}}$$

leading to

$$C_{\text{integer-part}}.C_{\text{fractional-part}} = (2^{-f} \times A) \times (2^{-f} \times B) = 2^{-2f} \times (A \times B)$$

There can be a problem here, arising from the need to shift the bits by $2f$. Bits may be shifted so many times that they go beyond the number of bits available to represent the number. This means that an error occurs and that the result will be incorrect. This is one of the drawbacks of such a simple method as fixed-point arithmetic. Another obvious problem is that it is an inefficient use of the available bits (creating a lack of precision). Allowing the binary point to move is much more practical and this is called *floating-point arithmetic*.

4.5 Floating point numbers

We need to keep track of the position of the binary point, and so it seems sensible to separate this from the numerical part of the number. Thus:

$$1011.10101 \qquad \text{becomes} \qquad 1.01110101 \times 2^3$$

This is identical in form to the exponential representation of decimal numbers, and can therefore be precisely defined in the standard format:

$$\pm 1.??????? \times 2^{\pm f}$$

where the $?$'s refer to the bits in the fractional part of the number, and f is the exponent. Note that it is a sensible strategy always to adjust the exponent of the number such that the *mantissa* is in the form of $1.???????$, i.e. a 1 followed by the binary point, then the fraction. This standard form must be strictly adhered to, otherwise the floating-point scheme will fail to be consistent when manipulating numbers.

How do we express this form in a fixed number of binary digits? This problem was solved a long time ago by adopting what is now called the IEEE floating-point representation. This has the following format:

s	exponent	fraction

So, in the case of a 32-bit number, s is the sign bit (left most bit), the $\mathrm{exponent}$ is the next 8 bits, and the $\mathrm{fraction}$ is represented by the remaining 23 bits. The precision with which numbers can be expressed using this standard form

is determined by the number of bits used in the fraction, whereas the range of the number is dependent on the bits used to record the exponent. In more detail:

Sign bit: This allows us to use "signed arithmetic". If $\mathrm{s} = 1$, then we have a negative number.

Exponent: The 8-bit $\mathrm{exponent}$ has a numerical range of 0 to 255. However, it is represented using what it is called "excess 127" format. This was introduced earlier in these lectures as offset binary. What this means is that the number 127 is regarded as being equivalent to zero, so that numbers greater than 127 are interpreted as positive and numbers less than 127 as negative. The mantissa is therefore multiplied by 2 raised to the power of $(\mathrm{exponent} - 127)$, with the additional restriction that $\mathrm{exponent}$ values of 00000000 and 11111111 are reserved to indicate numerical under/over-flow (i.e. the number is outside the range of representable numbers).

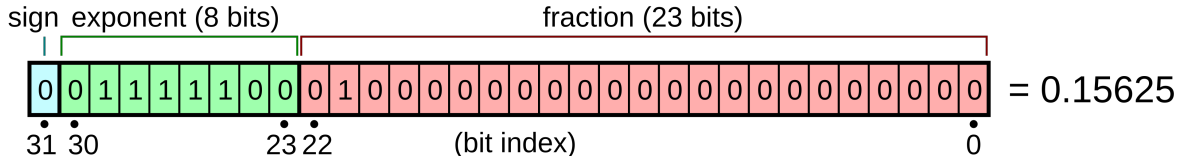
Fraction: The rules require the number to have its exponent adjusted so that the mantissa is always $1.??????????$. As the 1 at the start of the mantissa must always be present, it can be removed from the full number representation. It is called the *implied bit* (sometimes referred to as the *missing bit*). The $\mathrm{fraction}$ therefore only needs to record the binary values to the right of the binary point and essentially this provides one extra bit of precision in representing the number. This process is called normalisation and must be applied to all numbers at all times.

In summary, the 32-bit IEEE floating point representation has the following format

$$(-1)^s \times 2^{(\mathrm{exponent}-127)} \times 1.\mathrm{fraction} \tag{4.5}$$

with s, $\mathrm{exponent}$, and $\mathrm{fraction}$ represented using 1, 8 and 23 bits.

Example:



$$\left.\begin{aligned} \text{sign} &= 0 \\ \text{exponent} &= 124 \\ \text{fraction} &= 0.25 \end{aligned}\right\} \quad \Longrightarrow \quad (-1)^0 \times 2^{-3} \times (1 + 0.25) = 0.15625.$$