

Chapter 1 Introduction

1.1 Introducing digital electronics

Computers are at the heart of our daily existence, as are mobile phones. Both implement *software* using electronic circuits called *hardware*. The hardware is made up from *digital electronics*: these are circuits that can implement the software instructions. This book describes how you design the hardware, the digital electronic circuits. These circuits are used more widely than just in computers and phones. Around the home you will find them in many domestic appliances, even in front door chimes and kids' toys – researchers are even putting them into cloth to make it 'intelligent'.

This book starts with the basic devices and techniques and by its end you will have a good stock of building blocks and design techniques. The current position of digital electronics is a long way from its origins since it has progressed rapidly with the awesome developments in electronic (and particularly microelectronic) technology. Its origins lie in studies by mathematicians in the 19th century. One might be familiar; though Lewis Carroll is perhaps most famous for *Alice in Wonderland* he was also (as Charles Dodgson) an eminent mathematician [Dodgson87] (references are to be found in Appendix 2). However it was George Boole who published *An Investigation into the Laws of Thought* [Boole54] which was such a major contribution that the algebra is called Boolean logic. This algebra had to await application in an engineering sense until Claude Shannon implemented it in switching circuits using relays which were the earliest form of switching devices. Relay circuits progressed to valve circuits but the computers that were built using them were primitive, bulky and power greedy. The next major breakthrough was the semiconductor transistor which miniaturised switching circuit implementation. When small numbers of switching circuits were combined (or integrated) in a single chip the technology was known as *Small Scale Integration* (SSI). As miniaturisation increased so did the number of devices on a chip and microelectronic technology matured through *Medium Scale Integration* (MSI) to *Large Scale Integration* (LSI) in the 1970s to *Very Large Scale Integration* (VLSI) in the early 1980s. The term VLSI still applies though *Ultra Large Scale Integration* (ULSI) found some favour. Miniaturisation has continued and 'microelectronics' seems rather dated now and the continuing progress is reflected in the newer term 'nanoelectronics'. We now get several computers (cores) on a chip, and that was a dream once.

Design techniques have developed concurrently with these hardware developments and design techniques have even been motivated by technological development. Some older design techniques have little relevance now having been developed concurrently with, say, the devices in MSI or LSI technology. The main developments in recent years have included programmable systems that require software specification of a device's function. Principles

do not change, and that is what this book concentrates on (and why!).

1.2 Organisation of this book

The book starts with study of combinational logic in Chapters 2 and 3. Chapter 2 covers the basics of combinational logic, how basic logic circuits can be designed to implement digital functions, whilst Chapter 3 covers how we reduce or minimise the circuits to achieve practical designs. This is the basis of the subject and several presentations of this subject can be found in the selection of texts to be found in Appendix 2. The major difference in this book is that we shall concentrate on principles in a practical framework and in an approachable way.

Logic circuits are implemented using electronic circuits which are described in Chapter 4. Computer Science students can study these circuits to appreciate the advantages of particular technologies since these confer advantages to the computer systems that they program and interface. Electronic and Electrical Engineering students should note that the treatment of microelectronic technology here concerns switching circuits only, though these are just a branch of a much wider subject. Digital circuits are implemented using logic technologies which continue to evolve and this presentation is a snapshot of those technologies predominating at the time of writing, and this is currently dominated by CMOS (Complementary Metal Oxide Silicon no less) since it can combine high speed operation with low power consumption.

A major part of digital electronics is sequential system design which concerns digital circuits that go through a sequence of specified conditions, or states, as in a computer program. The main components are called bistables and are described in Chapter 5; the main sequential design techniques are the focus of Chapter 6. The main design techniques here use positive edge-triggered bistables, and design systems using the Algorithmic State Machine method.

A number of textbooks on digital electronics actually start with number systems and their arithmetic, concentrating on the binary system in particular, to introduce digital electronics as a vehicle for implementing binary arithmetic. This subject is covered here in Chapter 7, in part to allow for inclusion of design examples concentrating on circuit design. This allows Computer Scientists, in particular, to appreciate exactly how the circuits at the heart of a computer are designed. The design examples concentrate on comparing the advantages of different approaches, contrasting results achieved with those of commercial designs.

There are design examples throughout and these predominate in the design chapters: 3, 5, and 6. At the end of every Chapter there are example questions, and abbreviated solutions to the questions are to be found in Appendix 1. There is other support material and most of the circuits used herein can be found in working form for a freely available logic simulator, and downloaded from the web. In this way, we aim to ensure you have a full preparation of the principles of logic design. The references made in the text are to be found in Appendix 2. The earlier version of this text [Nixon95] covered more ground; here we concentrate on principles of logic circuit design and use a logic simulator (to make it more digestible). Here we concentrate on principles to enable a tight focus on the technology that is central to our daily lives.

1.3 Preliminaries

1.3.1 Binary Numbers

A number of assumptions have been made concerning prior knowledge and the most important of these are reviewed briefly here. Though binary arithmetic is studied in Chapter 7, earlier chapters do assume that you know what binary numbers are. The representation of a binary *integer* (without a fractional part) uses symbols 0 and 1 as *binary digits* or *bits* in the number. This can be compared with decimal arithmetic which has ten symbols, going from 0 to 9 which are used for the *decimal digits*. In decimal arithmetic the four digit integer 6203 is actually

$$6 \times 1000 + 2 \times 100 + 0 \times 10 + 3 = 6 \times 10^3 + 2 \times 10^2 + 0 \times 10^1 + 3 \times 10^0$$

Each decimal digit is then multiplied by a factor of 10. This factor corresponds to the position or significance of the digit which increases from the right (in this case 3 is the least significant digit) to the left (and 6 is the most significant digit). The basis of the decimal system is 10 and that is why we have ten symbols and a factor of ten multiplying each.

The *bit* is the basic unit in digital electronics and in digital communications. It is derived from *binary + digit = bit*. The basis of the binary system is 2: a bit can have one of two values and we have two symbols 0 and 1. The binary number 1100 (twelve in decimal) is made up from bits as

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 0 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

Since the base is smaller, and we have fewer symbols, the numbers need more digits. These are the bits and the bit with the highest factor of 2 (here 2^3) is the *Most Significant Bit* (MSB) and the one which multiplies 2^0 (1) is the *Least Significant Bit* (LSB). A 4-bit binary count sequence compared with its decimal equivalent is given in Table 1.1.

Decimal	Binary	Decimal	Binary	Decimal	Binary	Decimal	Binary
0	0 0 0 0	4	0 1 0 0	8	1 0 0 0	12	1 1 0 0
1	0 0 0 1	5	0 1 0 1	9	1 0 0 1	13	1 1 0 1
2	0 0 1 0	6	0 1 1 0	10	1 0 1 0	14	1 1 1 0
3	0 0 1 1	7	0 1 1 1	11	1 0 1 1	15	1 1 1 1

Table 1.1 Binary Numbers

The main advantage of the binary system is that it is very convenient for digital circuits in general and computers in particular. It is covered in much more detail later in Chapter 7, where other number systems are introduced, and we consider conversion between systems and circuits that implement computer arithmetic.

1.3.2 (Very) Basic Circuit Theory

Electronic circuit theory is the mathematical theory describing the operation of electronic circuits. Here is a very basic introduction: the theory concerns in particular the relation between voltage (or potential) and current (the flow of charge or electrons). By analogy to water, voltage can be considered as water pressure and current as the rate of water flow. Water flows from the supply to the destination, induced by the pressure difference between the two. Similarly, current in electronic circuits is related to voltage. As in Figure 1.1, electronic current, denoted i , is shown by an arrow to flow from the positive power supply (+Volts) to the negative power supply (–Volts, usually ground or 0 Volts) and is depicted as opposite to the flow of electrons which are negative charges (and thus attracted to the positive voltage supply).

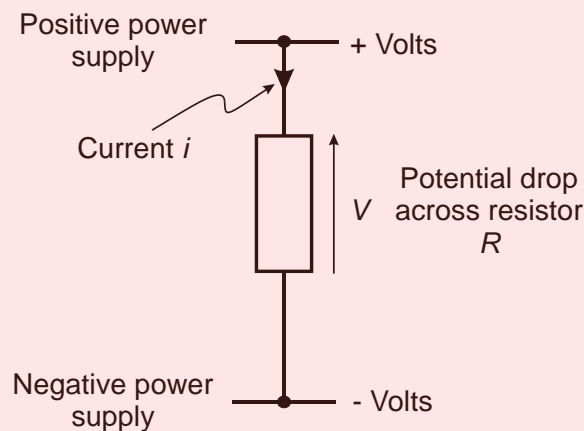


Figure 1.1 Circuit Characteristics

The relationship between voltage, resistance and current is given by *Ohm's law* as

$$\text{Voltage (V)} = \text{Current (i)} \times \text{Resistance (R)}$$

This implies that for a constant resistance, if we increase the voltage then the current increases as well, by analogy greater water pressure increases the rate of flow. Also, for a constant voltage supply then an increase in resistance implies a reduction in current. That is actually all we need for most of this book (in that we will not delve any deeper). Electronics and electrical engineering students will meet this as *analog circuit design*, which underlies the digital circuit design covered here.

1.3.3 Circuit Drawing

There are several conventions for *drawing logic circuits* that are in common practice and will be used throughout this book. The first concerns how joins are depicted between circuit conductors. The join between one conductor and another, where the first one terminates is shown in Figure 1.2(a). When the join is between two conductors where both continue, this is shown in Figure 1.2(b). The circuit where two conductors cross without intersection is shown

in Figure 1.2(c) – there is no join between the two conductors in this circuit. (At the time of writing, the standard for ‘structuring of the information about systems’ is the International Standard IEC 81346-1, and this is a highly abbreviated form.) Note that if you follow the convention of Figures 1.2(a) and (b) precisely, then there is actually no need for the dot.

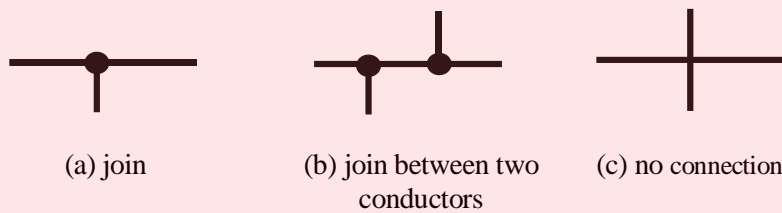


Figure 1.2 Drawing Circuit Connections

In practice, when conductors are joined together the circuit can sometimes be INCORRECTLY depicted as shown in Figure 1.3(a). This is avoided since it could be wrongly interpreted when, say, the circuit was built from a photocopied diagram of a circuit like Fig 1.2(c) and there was a small speck of dirt on the photocopier which made the circuit appear like Figure 1.3(a). An old fashioned way of drawing the circuit in Figure 1.2(c) is shown in Figure 1.3(b) where little “bridges” signify the absence of connection. This is very old fashioned: imagine doing the little bumps when one line is crossed by 64 others. That’s a big ouch and we do not do it that way these days. Don’t draw circuits like Figure 1.3!

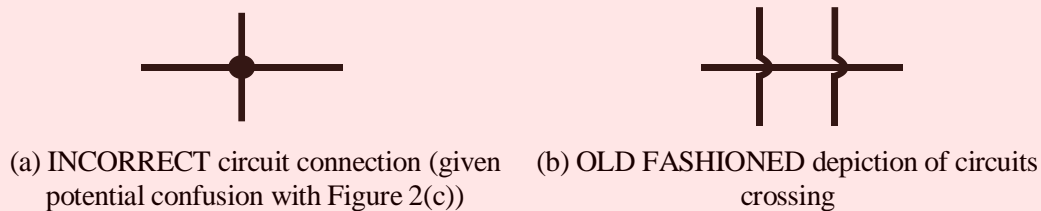


Figure 1.3 Bad Drawing Styles

Another convention is in terms of direction of flow: inputs to circuits appear on the left and outputs appear on the right, as shown in Figure 1.4(a), and (when in circuits) do not have arrows indicating direction. In digital electronics, we can have connections made from multiple signals (on a ‘bus’), shown in Figure 1.4(b) where each bus contains three signals (wires).

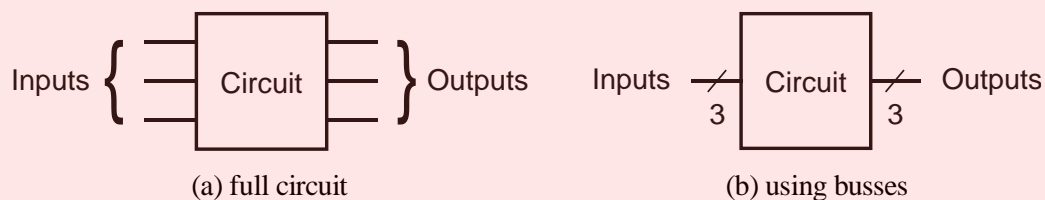


Figure 1.4 Input/ Output Convention

1.3.4 Logic Simulators

This is where we are grateful to Carl Burch who developed Logisim (<http://ozark.hendrix.edu/~burch/logisim/>) which is a nice and simple tool for simulating logic circuits. It is open-source, cross platform (it runs on Linux, MacOS X, and Windows) and Java based. In their blurb:

“Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes”

Logisim is an excellent tool for introductory logic design. It is certainly not perfect as it misses out some points we shall need. Overall though, it serves its task very well indeed and you can quickly draw up circuits and see how they operate. There are worksheets for each of the main logic design chapters (Chapters 2, 3, 5 and 6) and these can be found at www.users.ecs.soton.ac.uk/msn/digitsbook/ (which is this book’s website). There is a new version of Logisim in development, but at the time this book was completed it was not available in operational form. There are others, such as the closely named LogicSim (http://www.tetzi.de/java_logic_simulator.html), which largely serve the same purpose - but my students have found Logisim the easiest to install and to use. Some of the disadvantages of Logisim are that it has no timing information and cannot draw waveforms, but these are easily outweighed by ease of use and clarity of display. Most of the documentation is on Logisim’s website and there are links to two published papers, [Burch02] and [Burch04]. I have used simulators in my lectures since their inception and my students find them extremely useful. Naturally, the simulators give a way for you to check that your design really works, and that is always wonderful to see.

These preliminaries serve as a basic introduction to the remainder of this book. We shall start at the beginning of digital circuit design, combinational circuit design, which is the basis of the whole subject.

1.4 Conclusions

We’re ready to start now and we’ll start at the beginning. The circuit theory overviewed here will not be used until Chapter 4, but it’s worth practicing it now in the example questions which follow. For those who would like more background on electronic circuit theory then have a look at Sedra, A. S. and Smith, K. C., *Microelectronic Circuits* [Sedra10], or Spencer R. and Ghausi M., *Introduction to Electronic Circuit Design* [Spencer03] and there are many other similar texts.

We will be using the drawing conventions, binary numbers and Logisim in the next Chapter. You can either memorise Table 1.1, or go to the Chapter 7 and have a read there. It is interesting stuff, but we decided to place it later in the book so we could head straight into the design of basic digital circuits. That’s the next Chapter.

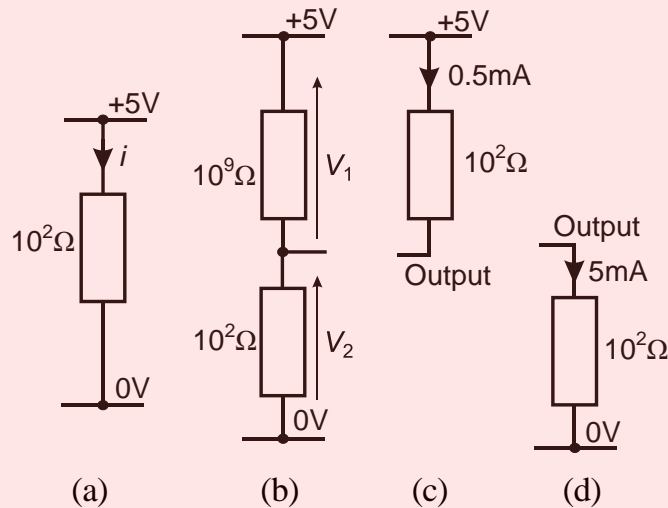
1.5 Chapter 1 Questions (Solutions in Appendix 1)

Figure. 1.5 (Very) Basic Circuits

Q1.1 For the circuit in Figure 1.5(a), determine the current i .

Q1.2 For the circuit in Figure 1.5(b), determine the voltages across the two resistors.

Q1.3 For the circuit in Figures 1.5(c) and (d), determine the output voltage in each case.

Don't forget this analysis, as we are actually describing the outputs of logic gates and that will be covered in Chapter 4.

Q1.4 Install Logisim (<http://ozark.hendrix.edu/~burch/logisim>).

Chapter 2 Combinational logic design

2.1 Combinational logic/Boolean algebra

This book is about introductory digital electronic design. The focus is design and so we shall start with the building blocks we use to make up designs. As such, we shall leave the formal descriptions until later, and here we shall use *Boolean algebra* to develop digital electronics. In its broader form it is known as discrete mathematics which, amongst other subjects, has wide application and implication in software engineering. This introduction is functional and serves as a basis for design. For deeper study (albeit sometimes pedagogic to an engineer) you can one of the texts described at the end of the chapter (and referenced in Appendix 2).

One definition of *logic* is as a chain or science of reasoning. For Boolean algebra we are concerned only with truth or falsity. We are concerned with propositions that can have either of two values: **FALSE** or **TRUE**. A *proposition* is a variable or fact. For example, the proposition ‘the contact lens is not cracked’ is TRUE if the lens no crack in it, but FALSE if the lens is actually cracked. There is no halfway house here: propositions are only TRUE or FALSE.

A *statement* is formed by joining propositions; the overall validity of the statement depends on the validity of each proposition from which it is formed. Consider for example the statement ‘A good contact lens is not cracked and circular’. (This will later be formulated in terms of industrial quality control.) If the proposition ‘the contact lens is not cracked’ is TRUE and the proposition ‘the contact lens is circular’ is TRUE then the statement ‘the contact lens is good’ is TRUE, since it is neither cracked nor an ellipse. This can be summarised using a *truth table* which shows the validity of a statement according to the validity of its constituent propositions.

Note here that the contact lens can only be cracked or not - just one of two values. A contact lens is not slightly cracked or part broken (when it's cracked you wouldn't buy it: it's bust). This is in accordance with Boolean algebra where variables can have two states only – though in practice small cracks invisible to the human eye are doubtless fine (depending on who's wearing the lens).

A truth table for a statement made from two propositions is a list of the validity according to the four possible sets of the two propositions. For propositions *A* and *B* the four sets or combinations are:

- (i) both FALSE; (ii) *A* FALSE, *B* TRUE; (iii) *A* TRUE, *B* FALSE; and (iv) both TRUE.

Using FALSE = F and TRUE = T, the truth table for a statement which is a function of these two variables, $f(A,B)$, is

A	B	$f(A,B)$
F	F	
F	T	
T	F	
T	T	

The question then remains to determine whether the output is TRUE or FALSE for each input combination. Take the statement 'A good contact lens is not cracked and circular'. This can be broken into a statement 'The contact lens is good' and two propositions 'the contact lens is not cracked' and 'the contact lens is circular'. The truth table then indicates when or not the contact lens is good according to whether it is cracked (or not) or circular (or not).

A = proposition 1, 'the contact lens is not cracked' (TRUE = unbroken, FALSE = broken)
 B = proposition 2, 'the contact lens is circular' (TRUE = circular, FALSE = elliptical)
 $f(A,B)$ = statement, 'the contact lens is good'

Propositions			Statement
A	B	$f(A,B)$	
F	F	F	The contact lens is cracked and not circular, it is not a good lens.
F	T	F	The contact lens is round but no good since it is cracked.
T	F	F	The contact lens is not cracked but it is not round: it's no good.
T	T	T	The contact lens is good since it is not cracked and it is circular.

2.2 Logic functions

What we have actually defined is a way of linking variables to form a result. As in traditional mathematics, the variables are linked by a function. The function that we have just seen illustrated is actually the *AND* function, which is perhaps unsurprising since the propositions were connected using the word 'and'. The truth table for the AND function is then

A	B	$A \text{ AND } B$
F	F	F
F	T	F
T	F	F
T	T	T

This shows that the AND function is identical to the way that we use it in language. The function $A \text{ AND } B$ is only TRUE when A is TRUE and B is TRUE, otherwise it is FALSE.

There is also a logical *OR* function which again follows the way we use it in language. Consider the statement: 'I will watch television if I have nothing else to do or there is a

programme I really want to see'. Given the statement 'I will watch television', this will be TRUE if either or both the propositions 'there is a programme I really want to see' and 'I have nothing else to do' are TRUE. The OR function can be summarised by its truth table

<i>A</i>	<i>B</i>	<i>A OR B</i>
F	F	F
F	T	T
T	F	T
T	T	T

As with the AND function, the word 'or' connects the two propositions in the original statement, so it is used in a logical way in a similar way as we use it in language.

The AND and OR functions are basic logic functions. The last member of the basic set is called *NOT*. Its function is implicit in its name; it provides a logical **complement** or *inversion*. NOT TRUE is FALSE; NOT FALSE is TRUE, which gives a truth table

<i>A</i>	NOT <i>A</i>
F	T
T	F

These functions are actually mathematical functions and there is a **logic algebra** formulated around them. This is the basis of discrete mathematics. The aim here has been to introduce combinational logic as part of digital electronics; we shall now move to logic circuits, and then define the logic algebra later in the context of combinational logic design.

2.3 Combinational logic implementation

We have so far considered Boolean algebra as a formalism to demonstrate logical relations. Consider for example the problem of the contact lens which when on a production line has a natural system implementation: a quality control system to remove the lens from the production line.

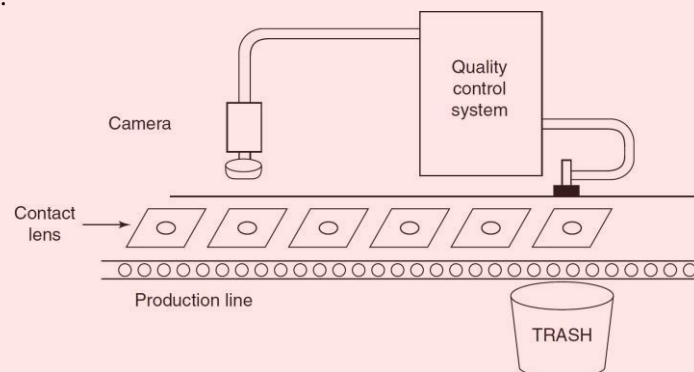


Figure 2.1 On Quality Control

In your studies, away from digital electronics, you will doubtless learn how to design systems to measure whether the articles are cracked (can we use ultrasound or computer vision?) or whether it is circular (how do we measure shape?), and systems to remove objects from a production line. Before that though, we need to design a system that can interpret the signals from the sensors, to combine them together to provide a signal that stimulates removal of the contact lens. Generally that is part of a computer program. Here we are the heart of the matter: how do we combine signals to achieve a desired function. This is combinational logic, the implementation of Boolean algebra in switching circuits. Logical variables, *inputs*, are connected by logical functions, *gates*, to give a logical result, an *output*. A gate is an electronic circuit that implements a logic function via a switching circuit.

2.3.1 Combinational logic levels

We need signals to represent true and false. These are necessarily electronic signals since we shall implement them in electronic circuits. We could use current, or voltage, and both are indeed used in logic circuits. At first though we shall use *level logic*, voltage levels to denote TRUE and FALSE.

$$\text{TRUE} = '1' = +5 \text{ V} \quad \text{and} \quad \text{FALSE} = '0' = 0 \text{ V}$$

Modern logic systems actually use a voltage lower than +5V, but this remains a good starting point. We use the quotes to denote logical variables. This is because we are using a binary system with just two values and this is just one way of indicating it.

The use of an explicit level for TRUE and FALSE would be very difficult to achieve. We actually use a range of levels (for reliability).

The use of a high level to indicate a '1' and a low level to represent a '0' is called *positive logic*. You will meet this often in *manufacturers' data* which define how circuits operate.

2.3.2 Combinational logic gates

(i) AND

The AND gate is a circuit implementing the logical AND function. The AND function is TRUE when both propositions are TRUE, the output of an AND gate is '1' when both inputs are '1'. If either input is a '0' then the output is '0'.

Truth Table		
A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

AND Circuit Symbol

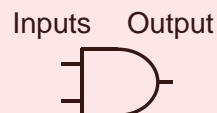


Figure 2.2 AND Gate

The symbol \cdot symbolises the AND function - it looks like multiplication and in a loose way we can now think of the AND function as multiplication (anything multiplied by zero results in zero). As in algebra, the \cdot can be (and often is) omitted. In circuit diagrams we need a symbol for the AND function. The symbol for AND is a bit like the letter D in AND, but stretched a bit horizontally. In terms of a switching circuit we can visualise the AND function as the collection of electrical switches to switch on a light; we connect a power supply V_{cc} so that current flows from through the switches illuminating the light to ground (0V). The buttons are normally open (which represents '0') and close when pressed (representing '1'), and the light being on represents a '1'.

Positive power supply V_{cc}

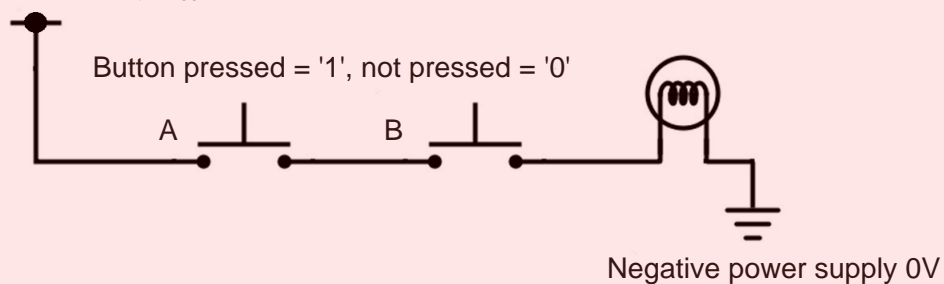


Figure 2.3 AND Gate Equivalent Circuit

In the AND function we need both inputs to be a '1' before the output is a '1'; here we need both buttons to be pressed before the light is switched on. This can be seen in the operation of the gate: the lamp will be illuminated only when both inputs are at '1'. This is the same for an AND gate: both inputs at '1' and the output is '1' (and a bit lighter!); any input at '0' and the output is '0' (the output is black) as in Figure 2.4.

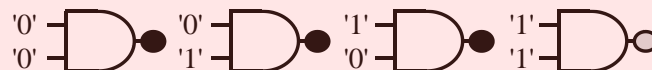


Figure 2.4 AND Gate Function

ii) OR

Truth Table

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

OR Circuit Symbol

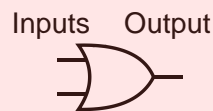


Figure 2.5 OR Gate

OR is symbolised by addition which forms a useful memoriser. It is a loose mnemonic since $1+1=1$! (We shall study arithmetic systems later and find the proper circuit for addition.) The switching circuit for OR is given by two switches in parallel. Should we press either or both buttons then the lamp will light.

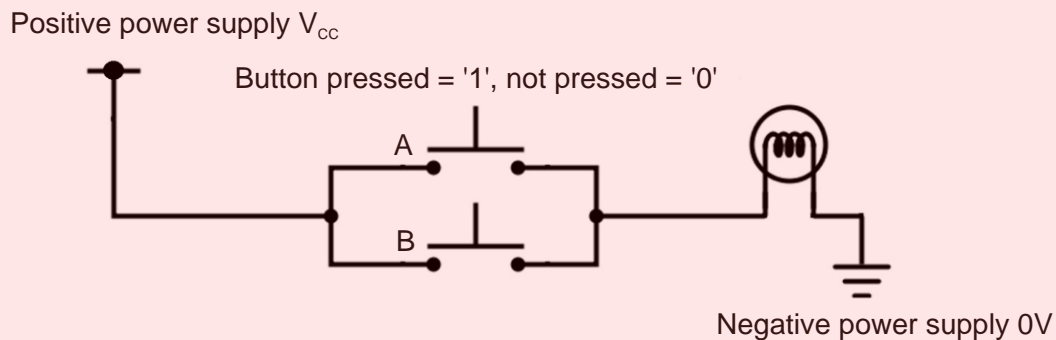


Figure 2.6 OR Gate Equivalent Circuit

Note that diodes can be used to replace the switches in the OR circuit. If the diodes' anodes are connected to logic inputs, and their cathodes are connected to the lamp then we effectively form a diode OR gate.

iii) NOT/ Inverter

The NOT function is symbolised by placing a bar across a logic variable: \bar{A} represents NOT A. It has to be drawn carefully so as to cover all factors that are to be inverted.

Truth Table

A	\bar{A}
0	1
1	0

NOT/ Inverter Symbol

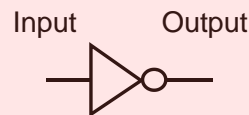


Figure 2.7 NOT Gate

2.3.3 Circuits and truth tables for more than two inputs

Truth tables and gates have so far been designed in terms of two inputs only. The truth table is merely a list showing the value of an output (or statement) for a combination of inputs (propositions). It can easily be extended to handle more than two input variables. Consider a function 'A cup is good if it has a handle, it is unbroken and it is glazed'. This function is TRUE when all three propositions are TRUE. In terms of a truth table, we need to tabulate whether or not the output is '1' (TRUE) according to the status of the three propositions. Each of the propositions can have either of two values and so there are $2 \times 2 \times 2$ (i.e. 8) combinations. The truth table is then formed from the variables or propositions. The propositions are

A = the cup has a handle ('1' = TRUE = it has a handle)

B = the cup is unbroken ('1' = TRUE = unbroken)

C = the cup is glazed ('1' = TRUE = glazed, '0' = unglazed)

$f(A, B, C)$ = the cup is good ('1' = TRUE = good cup)

The truth table, Figure 2.8, then covers the eight combinations of A , B , and C giving the output value $f(A, B, C)$ for each combination. The function is TRUE only when all inputs are TRUE and therefore describes a 3-input AND function. It is implemented using a 3-input AND gate which provides, as output, a '1' (5 V) when all inputs are '1' (5 V) and its symbol is the AND symbol with three input connections. (For a switching circuit, we could simply add in a third switch for C in series with the ones for A and B to the circuit in Figure 2.3.)

There are two ways to construct a truth table. One is to list all input combinations by counting up in binary. The other way is to take an input and as a column write it down as 01010101, take the next column and write its values as 00110011, and the last as 00001111. Note that if you go too far (with too many rows), you'll just repeat the truth table (starting with a second entry for 000).

Truth Table			
A	B	C	$f(A, B, C) = A \cdot B \cdot C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3-input AND Symbol

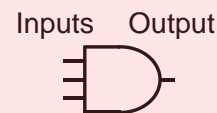


Figure 2.8 Three Input AND Gate

There is also a 3-input OR function which is TRUE if one or more inputs is TRUE. It is implemented using a 3-input OR gate which provides an output '1' (5 V) when any of the inputs is '1' (5 V) with truth table

Truth Table			
A	B	C	$f(A, B, C) = A + B + C$
0	0	0	0
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	1

3-input OR Symbol

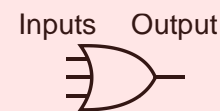


Figure 2.9 Three Input OR Gate

This can of course be extended into truth tables which describe circuits with four inputs or more. Note that for n inputs, there are 2^n combinations. A truth table for 10 inputs tabulates the value of the output for 2^{10} (i.e. 1024) combinations and is rather large. Not all circuits reduce to a simple gate implementation such as a 3-input AND gate; most usually reduce to a collection of different gates.

2.3.4 Don't care inputs

When implementing designs there are sometimes input combinations where the status of one input cannot affect the output. This may be because either it has no effect in that combination, or because it physically cannot occur. These are called *don't care inputs* and they are usually denoted using a X in the truth table; this signifies that the signal can be '0' or '1' and its value has no consequence on the output in that state. For an input B whose value cannot change the output function f when the input A is '1', the following truth tables are equivalent:

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

2.3.5 Further logic gates

Two other important gates are formed by combining NOT with AND and OR. The bubble on the inverter symbol actually symbolises inversion, so the combination of NOT and AND, called *NAND* is given by the inversion of AND (NOT AND) and is symbolised by the NOT bar across the AND function, and by the inversion bubble attached to the AND symbol.


Truth Table			NAND Symbol	
A	B	$\overline{A \cdot B}$	Inputs	Output
0	0	1		
0	1	1		
1	0	1		
1	1	0		

Figure 2.10 NAND Gate

A switching circuit can again be constructed but in this case by using switches which are normally closed (representing '0' here) and open when pressed (representing a '1').

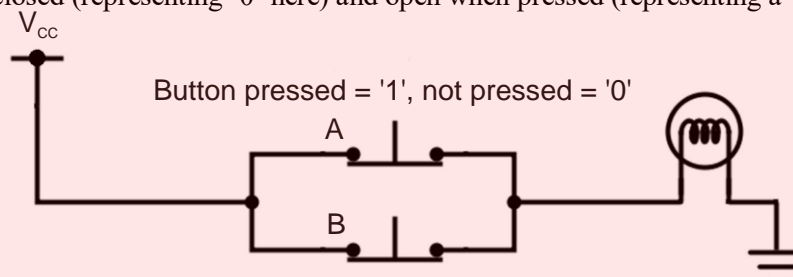


Figure 2.11 NAND Gate Function

Only when both buttons are pressed does the lamp switch off. If one, or neither, button is pressed, the lamp lights.

The AND circuit is formed by a series of switches whereas for an OR function they are in parallel. For NAND the switches are in parallel and for NOR in series, but with the normally closed switches rather than normally open switches. This illustrates that the NAND gate can be implemented as an OR gate with inverted inputs; it is often drawn this way, as part of a circuit drawing convention.

The combination of NOT and OR gives NOT (OR) called **NOR**. The truth table and symbol are then

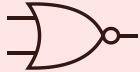
Truth Table				NOR Symbol	
<i>A</i>	<i>B</i>	$A + B$	$\overline{A + B}$	Inputs	Output
0	0	0	1		
0	1	1	0		
1	0	1	0		
1	1	1	0		

Figure 2.12 NOR Gate

with a switching circuit again using normally closed switches but this time connected in series.

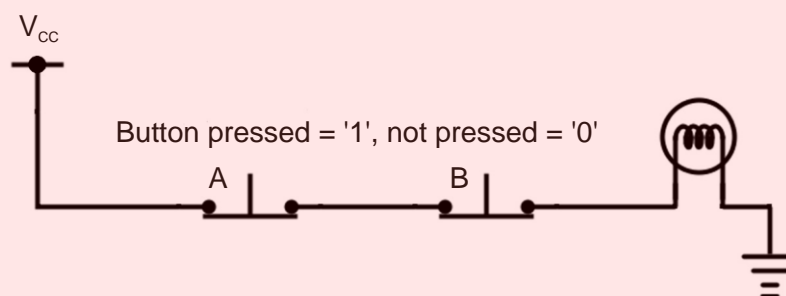


Figure 2.13 NOR Gate Function

Here the lamp is switched off when either button is pressed.

2.3.6 Other symbols

Other logic symbols are used, for example those defined according to British Standards (BS3939:1985).

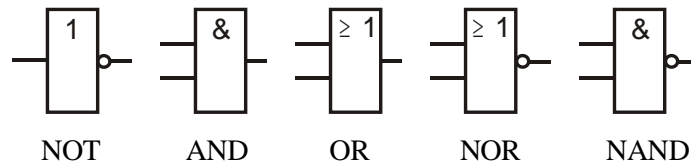


Figure 2.14 BSI Symbols

These symbols have found useage, but here we have used the symbols from standard IEEE/ANSI 91:1984 which derived from a US Military Standard set. These symbols will be used for logic gates throughout this book. The bubble signifying inversion can actually be attached either to inputs or to outputs giving two possible representations for an inverter.

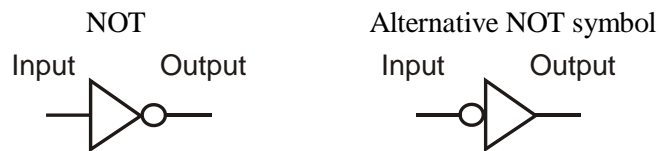


Figure 2.15 Mixed Logic Drawing

This is part of a drawing convention known as *mixed logic*, a convention for drawing circuit diagrams (which we shall avoid as it's very confusing for an introductory view).

2.3.7 Complete set of 2-input logic functions

For a 2-input logic gate there are four possible input combinations. For each of the combinations there are two possible output values. A 2-input logic gate then has $2^4 = 16$ possible output sets.

Inputs		Possible Output Sets															
A	B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

By inspection we can recognise some of these. Set 1 represents AND, set 7 is the OR function, set 8 is NOR and set 14 is NAND. Some are evidently recognisable and equally evidently not a logic gate: set 0 is '0' for all combinations, whilst set 15 is '1' for all combinations. As logic functions they are called FALSE and TRUE respectively; there is no gate which implements them since it wouldn't be any use (if we want a '0' or a '1' we use the power supply since that's what it's there for). Set 3 is a direct copy of A, set 5 is B, set 10 is \bar{B} and set 12 is \bar{A} . The existence of these as a function is a consequence of determining all possible output

combinations. Others are equally simple: Set 2 is '1' in one case, for $A = '1'$ and $B = '0'$. This is then $A \cdot \bar{B}$ and so set 2 represents an AND gate with one input inverted. So does set 4, $\bar{A} \cdot B$. Set 11 is $A + \bar{B}$ and set 13 is $\bar{A} + B$; again there is no gate to implement them. Only two combinations remain, set 6 and set 9, and for combinational logic these are functions in their own right. Set 6 represents the OR but excluding the case when the inputs are both '1'; in consequence it is called *exclusive OR* or *XOR*. Set 9 is the logical inversion of set 6 and is called *exclusive NOR* or *XNOR*. The XNOR function is '1' when the inputs are the same is for this reason is often called *equivalence*. All functions in this table are actually named in discrete mathematics, but except for AND, NAND, OR, NOR or NOT, only XNOR and XOR have any consequence in combinational logic design. The final member of the set of logic functions is a *buffer*, whose output follows the input.

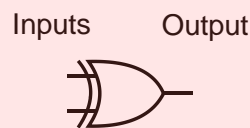
The functions that are gates are actually those which are symmetric in both inputs. That way it does not matter which way round they are connected. We shall come to this later, when we consider logic laws

XOR

Truth Table

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XOR Symbol

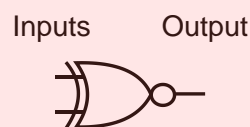


XNOR

Truth Table

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

XNOR Symbol



Buffer

Truth Table

A	A
0	0
1	1

Buffer Symbol

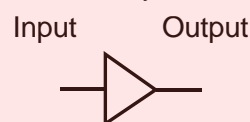


Figure 2.16 Further Logic Symbols

A buffer is used to distribute and *interface* (connect) signals in logic circuits. Its symbol is that of an amplifier and that is its function: it is used to maintain power (or to top it up) in practical circuits. Logic gates can only source limited amounts of current and thus only drive a limited number of gates. If we need to distribute a signal via a particular gate, to more than its specified maximum, then we need to buffer the signal and feed it via the buffer circuits. This coincidentally explains the use of the bubble to signify inversion - by its omission, the buffer is not NOT! These complete the full set of available functions.

2.4 Integrated circuits

AND, OR and NOT gates are available as *integrated circuits* or *chips* which give packages of gates. The circuit is actually inside the chip and the legs provide connections to your circuit.

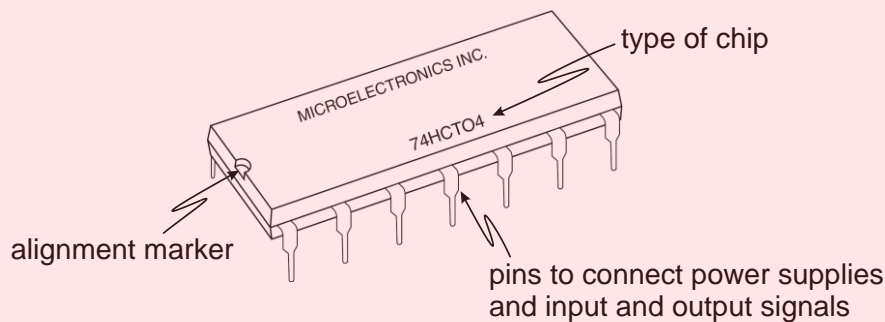


Figure 2.17 Dual in Line (DIL) Chip

There is an alignment marker to ensure that you use a chip the right way round. There are two pins for the *power supplies* and the remainder are for connections to the circuits inside. There are many varieties of chips; common circuits are usually available in a *logic family* which is a series of common logic functions implemented using a particular *logic technology*. The most famous and enduring logic family was the 74 series introduced by Texas Instruments. Though it is largely obsolescent now, it remains an excellent introduction to logic technology. Amongst its members was the 7404 which provided six NOT gates (hex NOT). The six NOT gates together with two power supplies (note the 5V power supply – modern systems use a lower voltage) results in a 14 pin chip arranged as shown in Figure 2.18. The other power supply is ground GND which is the negative power supply 0V.

The 74 series was particularly enduring in that it was implemented in differing logic technologies (TTL and CMOS). Any logic technology is a compromise between many factors, of which speed and power are amongst the most important, as considered in Chapter 4.

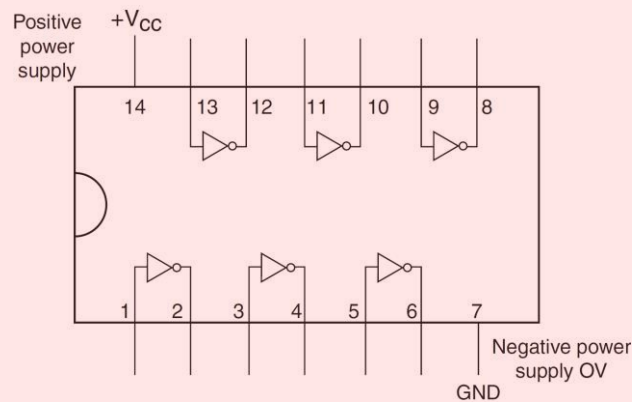


Figure 2.18 7404 NOT DIL Chip

The 74 series included many more chips which extended to combinations of gates. Less use is made of these chips now as *programmable logic* now dominates implementation. Programmable logic chips offer a set of gates and the interconnection between these gates can be specified by software, thus controlling the chip's function. The chip then has a clearly documented design and reprogrammable chips can easily be reconfigured. These chips are a bit complex at this stage, and will be covered later.

2.5 Basic logic devices

2.5.1 Coders and decoders

The function of a *decoder* is naturally to **decipher** encoded information. Information is encoded primarily to maximise usage of channel capacity, such as for satellite communication where the available power is limited. By way of example, consider a traffic light control system where a controller determines the sequence and timing of the lights, but then needs to turn them on and off.

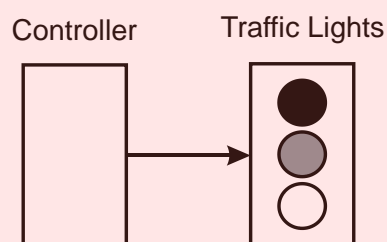


Figure 2.19 Traffic Light Control

The controller could provide three signals, one to turn on each light. To save money we could use just two wires and transmit logic signals by noting that two logic signals have four possible combinations. We can then design a *coder* which encodes the chosen information. By choosing the signals as

Transmitted signals		<i>A</i>	<i>B</i>
illuminate red bulb	R	0	0
illuminate yellow bulb	Y	0	1
illuminate green bulb	G	1	0

In software this would be of the form

```

IF red THEN BEGIN
    A=LOW
    B=LOW
    END
IF yellow THEN BEGIN
    A=LOW
    B=HIGH
    END
IF green THEN BEGIN
    A=HIGH
    B=LOW
    END

```

This software could also have been written using a CASE statement which would have simplified it slightly.

We can then use two lines to communicate between the controller and the lights. The signals need to be decoded from the chosen coding scheme to light the correct bulb. We then need a *decoder*. In this case, the specification for the decoder is

Received Signals		Illuminated Light	
<i>A</i>	<i>B</i>		
0	0	R	illuminate red bulb
0	1	Y	yellow
1	0	G	green
1	1	U	unused combination

This is by way of introduction, since the decoder is actually a basic building block in digital electronics and there are standard circuits.

2.5.2 2-4 line decoder

A decoder which decodes the four possible combinations of two input signals is known as a *2-4 line decoder*. Its function is given by the truth table in Figure 2.20. This shows which input combination selects which channel.

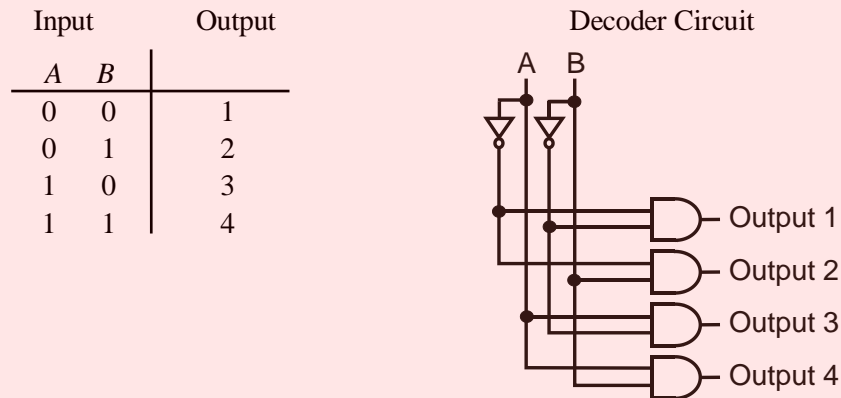


Figure 2.20 Decoder

The logic circuit which implements the 2-4 line decoder is shown in Figure 2.20. If both inputs A and B are '0' then Output 1 is selected and is '1' and all the other outputs are '0'. If A is '0' and B is '1' then the second output is selected and the others are unselected. The four cases of the circuit's operation are the four conditions for which the output channels are selected are shown in Figure 2.21 where a '1' is depicted in light grey and '0' in black.

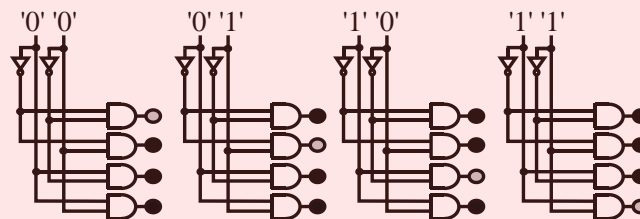


Figure 2.21 The Four States of a Decoder

The *decoder symbol* is a rectangle, shown in Figure 2.22. In this symbol, the numbering of the output channels is from 1 to 4, so channel 1 is output 1. The input selection is by two bits and these are shown underneath the decoder symbol. There are other symbols for a decoder, and this is the most common.

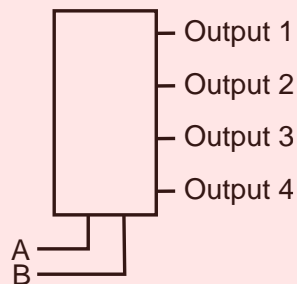


Figure 2.22 Decoder Symbol

A software description of the 2-4 line decoder is

```

CASE (A,B) ;comments follow the semicolon and are not executed
BEGIN
  00: BEGIN channel_0=1 ;If A=0 AND B=0 THEN channel_0=1
        channel_1=0, channel_2=0, channel_3=0 ;The others are 0
      END
  01: BEGIN channel_1=1 ;If A=0 AND B=1 THEN channel_1=1
        channel_0=0, channel_2=0, channel_3=0 ;The others are 0
      END
  10: BEGIN channel_2=1 ;If A=1 AND B=0 THEN channel_2=1
        channel_0=0, channel_1=0, channel_3=0 ;The others are 0
      END
  11: BEGIN channel_3=1 ;If A=1 AND B=1 THEN channel_3=1
        channel_0=0, channel_1=0, channel_2=0 ;The others are 0
      END
END
END

```

2.5.3 Multiplexers and demultiplexers

The function of a *multiplexer* is to choose an output from a selection of inputs in a manner similar to a rotary switch, which for two inputs is

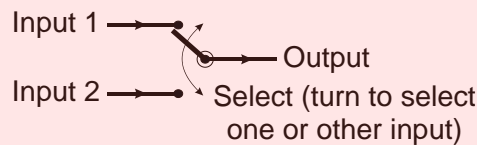


Figure 2.23 Multiplexer Function

A *multiplexer* symbol is

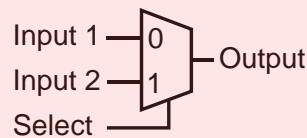


Figure 2.24 Multiplexer Symbol

The Select input determines which of these two buttons is connected to the circuit's output.

Its function is described in software as

```

IF (channel_select) THEN output = Channel_2
  ;control = 1 connects Channel_2 to the output
  ELSE output = Channel_1
  ;control = 0 connects Channel_1 to the output

```

The function of a *demultiplexer* is the converse, to choose an output signal. In software its function is

```

IF (control) THEN output_2 = input
    ;control = 1 connects output_2 to the input
    ELSE output_1 = input
    ;control = 0 connects output_1 to the input

```

There are other forms of multiplexing such as time domain multiplexing where signals occupy a communication channel for specified time slots (e.g. in radio communications). The function of the multiplexer is then to choose from the input signals at the appropriate time slot.

2.5.4 4-1 line multiplexer

A circuit which chooses a single output from one of four inputs is called a *4-1 line multiplexer*. Note that we need to choose which of four lines is to be selected and then provided as the circuit output. Two logic signals provide four combinations and the channel to be chosen is then encoded using two select lines.

Select Inputs		Output
A	B	
0	0	Input 1
0	1	Input 2
1	0	Input 3
1	1	Input 4

4-1 Line Multiplexer Symbol

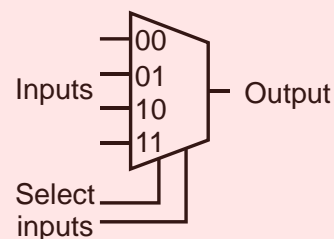


Figure 2.25 4-1 Line Multiplexer

Here the input selection is by two bits and this is shown underneath the decoder symbol. The circuit for a 4-1 line multiplexer, shown in Figure 2.26, then gates the channels with the select lines. The output is an OR gate as this combines the selected channel with signals which are all at '0' from the channels which are not selected (the output of one AND gate follows a selected channel whereas the output of all the other AND gates is '0'). In this way, only the selected channel can change the output of the circuit to '1' since all the other inputs to the OR gate are '0'.

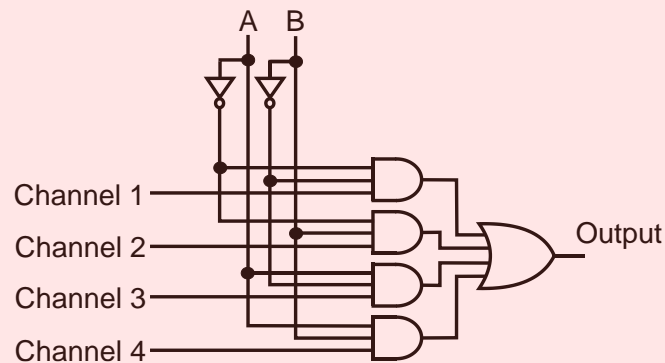


Figure 2.26 4-1 Line Multiplexer Circuit

A software description is

```

CASE (A,B)
BEGIN
  00: output=channel_1 ;If A=0 AND B=0 THEN output=channel_1
  01: output=channel_2 ;If A=0 AND B=1 THEN output=channel_2
  10: output=channel_3 ;If A=1 AND B=0 THEN output=channel_3
  11: output=channel_4 ;If A=1 AND B=1 THEN output=channel_4
END

```

Note that the specification on channel selection is identical to that for the 1-4 line decoder. This is hardly surprising since we are decoding two input lines to determine which of four inputs should be selected as output. The decoder signals are enabled when a particular channel is chosen and the multiplexer output is then given by ANDing the decoder outputs with the appropriate input signal. The outputs of the AND gates are then combined with a 4-input OR gate whose output will follow the selected channel. Figure 2.27 shows a working multiplexer, from a Logisim implementation. The circuit is the same as that for Figure 2.26 with two small changes. First, the input channels have signal generators attached and second the OR gate which collects the gated inputs is a 5-input gate (with one input set to '0') since Logisim does not allow four input gates. The lighter connections reflect a signal at '1' and the darker lines a '0'. In Figure 2.27(a) then connects Channel 1 to the Output when the select inputs A and B are both '0'. When A='0' and B='1' the output is derived from Channel 2, as in Figure 2.27(b). The same channel is connected in Figure 2.27(c) but the signal is '0' not '1' (note that Channels 3 and 4 are both '1' but are ignored).

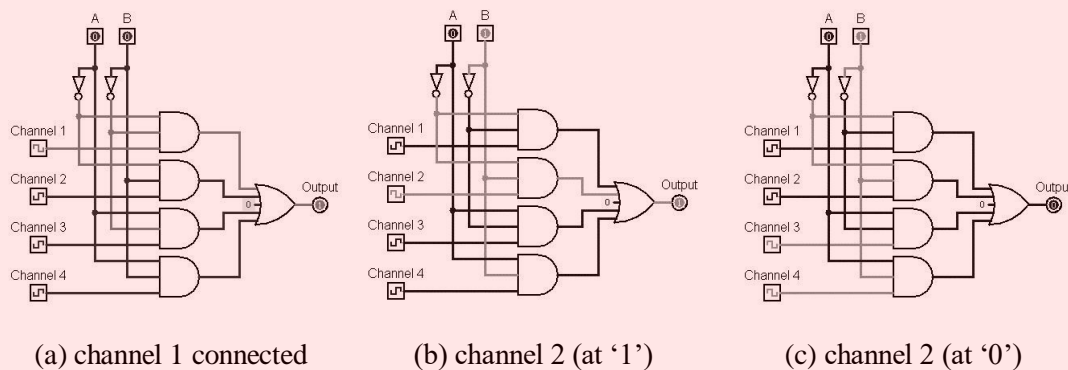


Figure 2.27 Multiplexer Operation

The function is then defined by a truth table using don't care inputs for the channels which are not selected. The output has the same (logic) value as the selected input channel.

Input Channel Status				Select Lines		Output Channel	
0	1	2	3	A	B	Selected	
0	X	X	X	0	0	0	} 1
1	X	X	X	0	0	1	
X	0	X	X	0	1	0	} 2
X	1	X	X	0	1	1	
X	X	0	X	1	0	0	} 3
X	X	1	X	1	0	1	
X	X	X	0	1	1	0	} 4
X	X	X	1	1	1	1	

2.6 Designing a combinational logic circuit

The design of a circuit can be extracted from a truth table specification. This is to be expected since a truth table is a complete specification of a circuit's operation. In a basic form the design can be implemented using AND, OR or NOT gates.

The basis for such design is that the function will be '1' for various combinations of inputs. These input combinations can be grouped using AND gates, since the output of an AND gate is '1' when all its inputs are a '1'. The output of the circuit should be '1' if any of these combinations produce a '1', and we can therefore group the AND gate outputs using an OR gate, since the output of an OR gate is '1' if any of its inputs are '1'. This procedure is reflected in the earlier circuit for the multiplexer where the gated decoder outputs were collected using an OR gate. The procedure for design is then

1. Fill in the truth table
2. Extract conditions for which the output = '1'
3. Group the conditions to form the final output

The process is best illustrated by example. Consider the following 'specification'.

John Wonderland is going on holiday with his wife Alice, his beautiful daughter Bo and her friends, Chas and Dave. He wants to fish peacefully while his wife chaperones Bo, so he decides to make an alarm which goes off when Bo is with either Chas or Dave (or both) but not Alice, either inside or outside the house. He buys radio transmitters to indicate whether or not people are inside the house and surreptitiously fixes them to everyone. He then needs an alarm circuit to act on the information from the transmitted signals.

This is clearly a fictitious problem. My students have suggested that they would steal Alice's transmitter and fix it to Bo.

We shall follow the stated design procedure and the first step is to draw up the truth table.

We shall use $A = \text{Alice}, B = \text{Bo}, C = \text{Chas}, D = \text{Dave}$ and $f = \text{alarm}$;
 and for all inputs, '1' = inside house, '0' = outside house;
 and for the alarm '1' = alarm rings, '0' = alarm is silent.

The truth table is then

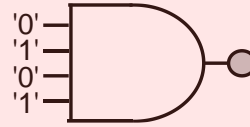
A	B	C	D	f	
0	0	0	0	0	All outside together
0	0	0	1	0	Alice & Bo outside together
0	0	1	0	0	Alice & Bo together
0	0	1	1	0	Alice & Bo together
0	1	0	0	0	Bo inside alone
0	1	0	1	1	f_1 Bo inside with Dave
0	1	1	0	1	f_2 Bo inside with Chas
0	1	1	1	1	f_3 Bo with Chas and Dave
1	0	0	0	1	f_4 Alice inside, Bo outside with Chas and Dave
1	0	0	1	1	f_5 Alice inside, Bo outside with Chas only
1	0	1	0	1	f_6 Alice inside, Bo outside with Dave
1	0	1	1	0	Bo outside alone
1	1	0	0	0	Bo and Alice inside
1	1	0	1	0	Alice, Bo and Dave inside
1	1	1	0	0	Alice, Bo and Chas inside
1	1	1	1	0	All inside

The output is true in six of the sixteen cases; in all others Bo is chaperoned by Alice, or alone, and the alarm does not go off. These six cases are labelled f_1, f_2, f_3, f_4, f_5 , and f_6 . In each of these cases a combination of inputs forces the output to be '1'.

For f_1 the output = '1' for $A = '0', B = '1', C = '0',$ and $D = '1'$
 i.e. when $\bar{A} = '1', B = '1', \bar{C} = '1',$ and $D = '1'$

f_1 is now expressed as variables which are all '1' which can thus be grouped using a 4-input AND gate. f_1 can now be implemented as

$$f_1 = \bar{A} \cdot B \cdot \bar{C} \cdot D$$

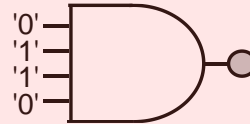


and the output will be '1' when Alice and Chas are outside and Bo and Dave are inside. Note that f_1 will be '0' for all other input combinations.

We can do the same for f_2 , so $f_2 = '1'$ for $A='0', B='1', C='1',$ and $D='0'$
i.e. when $\bar{A}='1', B='1', C='1',$ and $\bar{D}='1'$

which can be implemented as

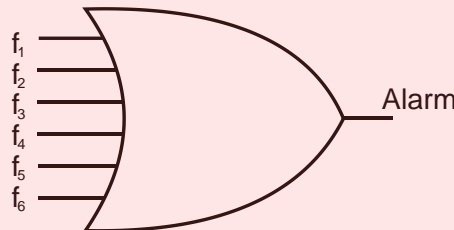
$$f_2 = \bar{A} \cdot B \cdot C \cdot \bar{D}$$



which is '1' only when Alice and Dave are outside, and Bo is inside with Chas.

In the same way, $f_3 = \bar{A} \cdot B \cdot C \cdot D$, $f_4 = A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}$, $f_5 = A \cdot \bar{B} \cdot \bar{C} \cdot D$, and
 $f_6 = A \cdot \bar{B} \cdot C \cdot \bar{D}$

We now have six circuits which provide the alarm if their particular input combination is satisfied. We want to collect together the six terms to provide a single alarm. We need a circuit whose output is '1' when any of the inputs are a '1' - this is an OR gate. We can then collect $f_1, f_2, f_3, f_4, f_5,$ and f_6 together, using



This then gives the full alarm circuit to provide the specified alarm according to the location of the conspirators which is

$$\text{alarm} = \bar{A} \cdot B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot D + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D}$$

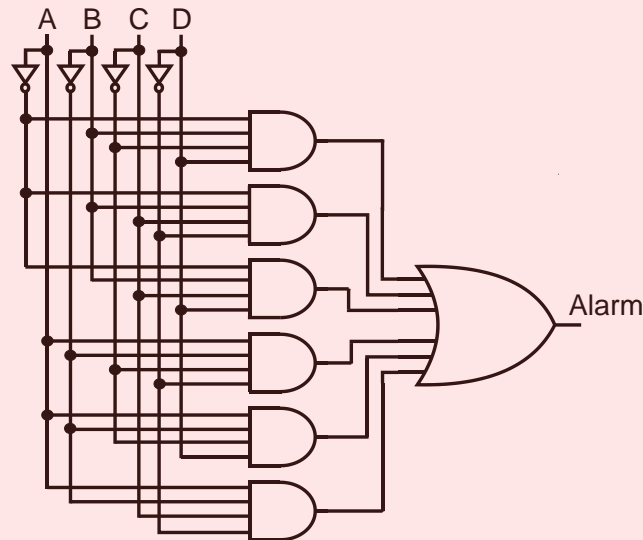


Figure 2.28 Wonderland Alarm

This circuit uses more gates than it needs to and it can be reduced or minimised, see Section 3.3.

A software description for the alarm function for the same inputs is

```
IF ((NOT(alice) AND bo AND chas AND dave)
    OR (NOT(alice) AND bo AND chas AND NOT(dave))
    OR (NOT(alice) AND bo AND NOT(chas) AND dave)
    OR (alice AND NOT(bo) AND NOT(chas) AND NOT(dave))
    OR (alice AND NOT(bo) AND NOT(chas) AND dave)
    OR (alice AND NOT(bo) AND chas AND NOT(dave)))
    THEN alarm=TRUE
    ELSE alarm=FALSE
```

This software uses more words than it needs to (it is too prolix) and it can be reduced or minimised, see Section 3.3.

We have actually decoded the six cases where we want the output alarm to ring. As such, we could use a decoder. The alarm can ring in any one of the six cases, so we group the outputs using an OR gate. There are wasted gates since those that decode the unused input combinations lead to outputs that never change and do not contribute to the alarm function. Note that the unused decoder outputs are left unconnected, since they do not affect the alarm.

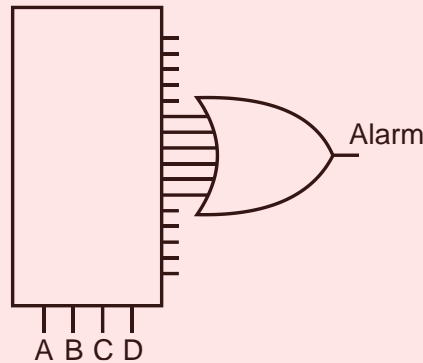


Figure 2.29 Wonderland Alarm via Decoder

We could also use a multiplexer, since that is largely a decoder controlling inputs to an OR gate. To do this we can use logic levels as the input channels and connect the inputs $ABCD$ to the select inputs. We connect a '0' to the channels for which the alarm should not ring, and '1' to channels where the alarm will ring. Then, since the function of the multiplexer is to select from the inputs according to the values of the signals on the select lines, the input '1's to the multiplexer will be selected by those (six) combinations of $ABCD$ for which the alarm should ring.

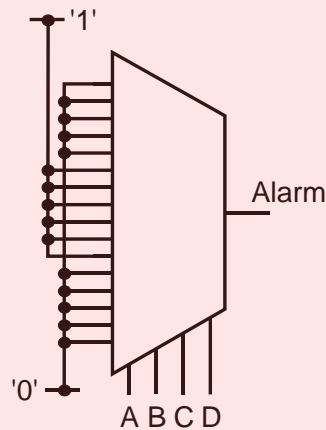


Figure 2.30 Wonderland Alarm via Multiplexer

Note that the inputs other than the ones for which the alarm should ring are connected to '0' and are not left unconnected because the output must be defined for every combination of inputs. The alarm should ring for six combinations and not ring for the remaining 10 combinations. As a general rule, inputs are never left unconnected, whereas outputs can be left unconnected as they are unused (as in the decoder implementation of the circuit).

2.7 Concluding comments and further reading

In this Chapter we have seen the development of logic from its formal status to a circuit implementation. Many of the themes presented in this Chapter will be manifest later in this

book.

Two books in particular give the historical basis of logic: Boole, G., *An Investigation into the Laws of Thought* [Boole54], originally published in 1854, and republished in 1954 by Dover Publications, New York, and Carroll, L., *Game of Logic* [Dogson87], originally published in 1896 in private form and pulped, then again in 1887, and republished again by Dover Publications, in 1958. These might be worth a look if you can find a copy. The discrete mathematics approach to logic is ably expounded in Ross, K. A. and Wright, C. R. B., *Discrete Mathematics* [Ross03].

There is a plethora of texts on digital electronics. This book concentrates on principles, which do not change. The implementation technology most certainly does. Amongst those which have found popularity are: Wakerly, J. F., *Digital Design: Principles and Practices*, 4th Ed., [Wakerly05] (though at the time of writing it is >100\$ and weighs 3.6 pounds), Floyd, T. L., *Digital Fundamentals with VHDL* [Floyd02], Katz, R. H. and Borriello, G., *Contemporary Logic Design* [Katz04] and Roth, C. H., Kinney, L. L., *Fundamentals of Logic Design* [Roth09]. These are weighty texts which cover a great amount of material, but perhaps in rather excessive detail.

2.9 Chapter 2 Questions (Solutions in Appendix 1.1)

Q2.1 By truth table analysis, show that

$$\begin{aligned}(A + B) \cdot \overline{A \cdot B} &= A \oplus B \\ \overline{A + B} \cdot A \cdot B &= 0 \\ \overline{A + B} &= \overline{A \cdot B} \\ \overline{\overline{A + B} + \overline{A \cdot B}} &= A \cdot B\end{aligned}$$

Q2.2 Draw up a truth table for each element the expression

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

and hence derive a truth table for the function f.

Q2.3 Draw the circuit to implement the function

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

using AND, OR and NOT gates

Q2.4 Implement the function

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

using an 3-8 line decoder and an OR gate.

Q2.5 Implement the function

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

using an 8-1 line multiplexer.

Q2.6 Draw the logic circuits (using AND, OR and NOT gates) for your solutions to 2.4) and 2.5) and show which parts of the circuits are redundant compared with your solution to 2.3)

Q2.7 Express the function

$$f = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot C$$

in software with logic functions AND, OR and NOT applied to logic variables which can be assigned HIGH ($f = '1'$) or LOW ($f = '0'$). Use IF..THEN..ELSE statements only.

Q2.8 Express the function in 2.7) using the same logical variable specification, but using CASE statements only.

Q2.9 Design a safety system for a one-seater car given:

- (i) an indicator which provides an output of +5V when someone is sat in the car and 0V otherwise;
- (ii) an indicator which provides an output which is +5V when the seat belt is connected properly across the occupant and 0V otherwise;
- (iii) an indicator showing that the door is shut (+5V output when the door is shut and 0V otherwise)
- and (iv) a system safety condition indicator for the brakes and suspension, with output +5V when the condition is unsafe and 0V otherwise.

The safety system should provide a signal f which can be used to prevent the car proceeding ($f = '1'$ to prevent progress and $'0'$ to allow progress). The car should not proceed when a driver is not sat in it or when the driver is not wearing a seat belt or the system condition is unsafe. The car should not proceed when the door is open.

- a) draw up a truth table for the signal f ;
- b) extract the logical function which gives the signal f ; and
- c) implement the system using NOT, AND and OR gates.

Chapter 3 Logic circuit implementation

3.1 Overview

So far, we have considered how to design circuits using AND, OR and NOT gates. These certainly work correctly, but so far are rather large and need to be reduced, or minimised. To achieve this we need an algebra by which we can use logic laws to minimise the circuits. As we shall find, this becomes rather complex and so we shall then consider other minimisation tools. All through, we shall consider circuits and again we shall end with a practical logic design.

3.2 Laws of logic

We have so far defined logic without a formal algebra. Including the algebra is a bit pedantic but much logic use does depend on it. Given binary variables and logical inversion, $\overline{\overline{0}} = '1'$, $\overline{\overline{A}} = A$ we have two sets of laws, one based on OR and the other on AND.

	OR laws	AND laws
1	$A + 0 = A$	$A \cdot 0 = 0$
2	$A + 1 = 1$	$A \cdot 1 = A$
3	$A + A = A$	$A \cdot A = A$
4	$A + B = B + A$	$A \cdot B = B \cdot A$
5	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
6	$A + B \cdot C = (A + B) \cdot (A + C)$	$A \cdot (B + C) = A \cdot B + A \cdot C$
7	$A + B = \overline{\overline{A} \cdot \overline{B}}$	$A \cdot B = \overline{\overline{A} + \overline{B}}$

For both AND and OR, laws 1, 2 and 3 follow from the definitions of the logic functions (law 1 for AND is rather neatly called the Annihilation law). Law 4 shows that it doesn't matter which way round the inputs are connected to a gate. Law 5 suggests that we can make a three-input gate from two two-input gates, and for this function that it doesn't matter how the inputs are connected, as shown in Figure 3.1 for OR gates in which all three circuits are equivalent and perform precisely the same function.

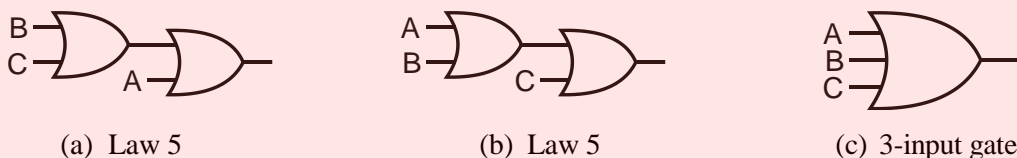


Figure 3.1 Equivalent Circuits by the Associative Law, Law 5

Law 6 is the distributive law and can be proved using the previous laws (it's a bit like algebraic multiplication) and are very useful for Boolean manipulation. The last two laws are versions of *De Morgan's law*. This law shows us how to swap logic implementation.

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

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

An easy way to remember how to use De Morgan's law is

“Break the line and change the sign. If there is no bar, add two and break one.”

The first part of the phrase is easily seen. The second part of the phrase allows use of De Morgan's law on functions which are not inverted, such as $A + B$, so

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

Note that it's the lower bar which is broken (breaking the upper bar would lead to a logic function which makes no sense since inputs can only be inverted one at a time). De Morgan's law implies *universality* of NAND and NOR gates. Since both NAND and NOR gates can be used to implement OR and AND, respectively, by inverting the inputs then any function can be implemented using just NAND or NOR gates only. This is illustrated further in Section 3.5.

De Morgan was the first professor of maths at UCL in England and invented proof by induction. Not the life and soul of a party though, he was described by a colleague as a “dry dogmatic pedant”. Note that his law is more general and you can use it to simplify complicated IF statements when programming (they are logic statements anyway!).

De Morgan's law can also be used to explain two symbols in common use for NOR and NAND. These are due to the fact that NAND is equivalent to the OR function with inverted inputs and NOR is equivalent to AND with inverted inputs. By De Morgan's law (law 7), and by using a circle on the input connection to denote logical inversion (as in the mixed logic convention) we obtain:

$$\overline{A \cdot B} = \overline{A} + \overline{B} \text{ (NAND becomes OR)} \quad \text{and} \quad \overline{A + B} = \overline{A} \cdot \overline{B} \text{ (NOR becomes AND)}$$

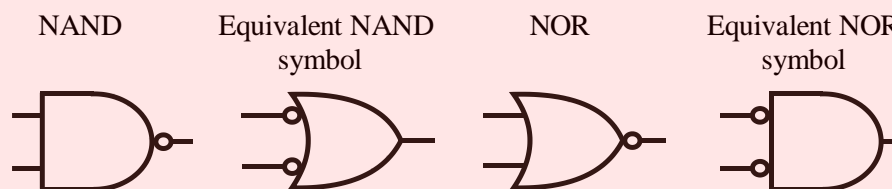


Figure 3.2 Other Gate Symbols

Two other laws can be derived from this algebra

$$A + A \cdot B = A \qquad A + \bar{A} \cdot B = A + B$$

For proof, for the first of these

$$A + A \cdot B = A \cdot (1 + B) = A \cdot 1 = A$$

for the second, by substitution for $A + A \cdot B = A$ then

$$A + \bar{A} \cdot B = A + A \cdot B + \bar{A} \cdot B = A + B \cdot (A + \bar{A}) = A + B \cdot 1 = A + B$$

Formal proof is more involved and rigorous. This is a functional proof only.

These laws lead to smaller, reduced, circuits.

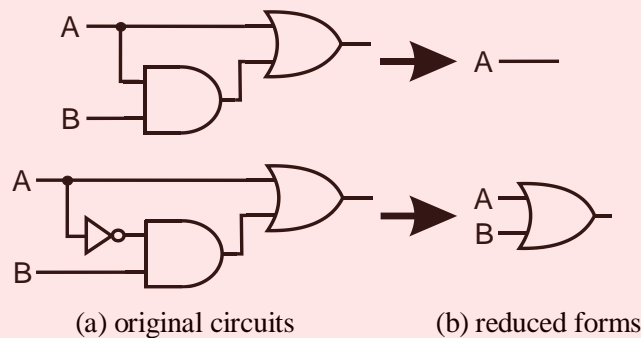


Figure 3.3 Circuit Minimisation

So in the first case we can replace two gates with a piece of wire and in the second case we can remove two gates (and their connections) and still achieve the same function. In other words, why use those OR and AND gates when they are actually redundant? Each gate costs money and has a certain reliability and lifetime associated with it. If we can reduce the number of gates we save money and effort, and make the circuit more reliable. This is *minimisation*.

3.3 Minimisation

Circuits cost money and consume power (also costing money). If we reduce the circuit, we reduce the cost (increase profit) and make it more reliable. Minimisation is therefore prompted by the desire to implement a logic function in as few gates as possible to reduce cost. Designers now use logic minimisers (implemented via computer programs) which do this automatically and this will be considered later. We shall first study some by-hand techniques, to show how minimisation can be achieved. The by-hand techniques are mainly historical now, but necessary since (automated) logic minimisers develop from them. There are *algebraic* and *graphical* techniques.

3.3.1 Algebraic minimisation

Earlier rules give the basis for *algebraic minimisation*.

$$(i) A + \bar{A} = 1 \text{ so } (ii) B \cdot (A + \bar{A}) = B \cdot 1 = B$$

To put the maths into words, in case (i) a signal OR'd with its inverted form covers all possibilities and is therefore TRUE. In case (ii), a signal ANDed with an input which is in its inverted and its noninverted form is just the signal itself (since it covers all combinations of input A). We also just demonstrated minimisation with two slightly more complicated laws: $A + A \cdot B = A$ and $A + \bar{A} \cdot B = A + B$.

We shall use the law $B \cdot (A + \bar{A}) = B$ to minimise John Wonderland's alarm circuit. The original circuit is in Figure 2.27 and needed six 4-input AND gates and one 6-input OR gate. The final extraction from the truth table was

$$f = \bar{A} \cdot B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot D + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D}$$

which using common terms gives

$$f = \bar{A} \cdot B \cdot D \cdot (C + \bar{C}) + \bar{A} \cdot B \cdot C \cdot (\bar{D} + D) + A \cdot \bar{B} \cdot \bar{C} \cdot (\bar{D} + D) + A \cdot \bar{B} \cdot \bar{D} \cdot (\bar{C} + C)$$

Note that you can consider terms more than once for the purposes of minimisation since $A + A = A$, i.e. using a term twice does not change the function at all. In this case, we have used the two terms $\bar{A} \cdot B \cdot C \cdot D$ and $A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}$ twice, so that the minimisation can be seen. Since the terms in brackets become logic '1' the minimised result is then

$$f = \bar{A} \cdot B \cdot D + \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{D}$$

which can be implemented using 2-input gates by sharing common terms as

$$f = \bar{A} \cdot B \cdot (D + C) + A \cdot \bar{B} \cdot (\bar{C} + \bar{D})$$

and the minimised circuits are

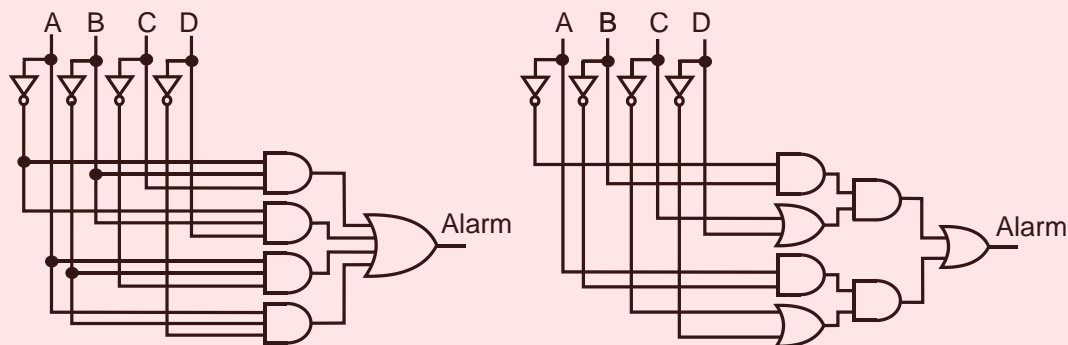


Figure 3.4 Reduced Wonderland Alarm Circuits

By De Morgan's law we could remove two of the inverters, since the last term can be replaced by a NAND gate as

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

We have then taken a function which originally required four inverters, six 4-input gates and one 6-input OR gate to seven 2-input gates plus the inverters – and it still delivers the same function. By De Morgan's law, the whole circuit can be implemented using just five 2-input gates and two inverters. Algebraic techniques are limited to small scale problems; for circuits with more than five inputs they become tedious and error prone (and you might not spot the best route to minimisation). As such we need a way which is more guaranteed to provide minimised circuits.

3.3.2 Graphical minimisation - the Karnaugh map

The *Karnaugh map* (K-map) uses the expressions $A + \overline{A} = 1$ so $B \cdot (A + \overline{A}) = B$, in a graphical way. Essentially it is a two-dimensional truth table. We first draw a box with inputs around it, which is like a map of the logical function. The input combinations point to the value of the function for that combination.

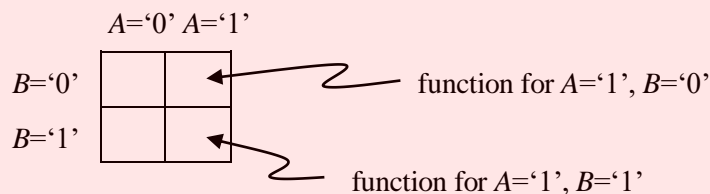


Figure 3.5 K-map Basis

Then we insert the value of the function for each input combination, so a K-map for AND is

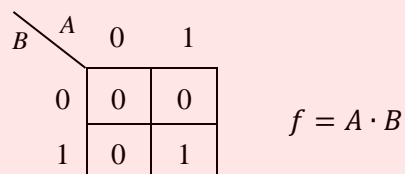
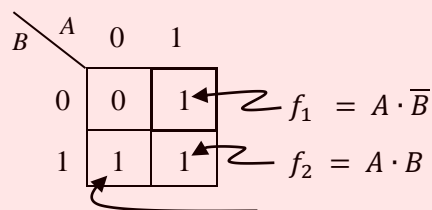


Figure 3.6 K-map for AND

This shows that the K-map is an alternative formulation of the truth table; it is now expressed in a compact two-dimensional way. But it has been introduced as a graphical minimisation technique, so how do we minimise using this? Consider the K-map for the OR function



$$f_3 = \bar{A} \cdot B$$

Figure 3.7 K-map for OR

By using algebraic minimisation based on f_1 , f_2 and f_3 (including f_2 twice) we obtain the function f that the K-map represents as

$$\begin{aligned} f &= f_1 + f_2 + f_3 = A \cdot \bar{B} + A \cdot B + \bar{A} \cdot B \\ &= A \cdot (\bar{B} + B) + (\bar{A} + A) \cdot B \\ &= A \cdot 1 + 1 \cdot B \\ &= A + B \end{aligned}$$

OK, we knew that it was OR already and that is in minimal form. The aim here is to illustrate the K-map minimisation process.

Extraction from the truth table was based on using individual terms and then reducing them algebraically. The K-map enables this reduction to be spotted **visually** - we can see terms of the form $A \cdot B + \bar{A} \cdot B = (\bar{A} + A) \cdot B = 1 \cdot B = B$, as we just did for the OR gate. We can draw loops around these common terms to minimise the circuit. Essentially, the loop is drawn around the input that changes and then removes it.

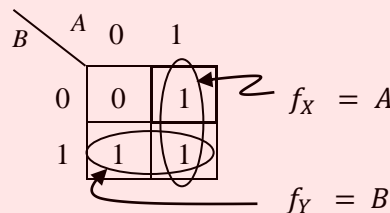


Figure 3.8 Looping Terms in a K-map

From the K-map for OR the function is then described by two loops, f_X and f_Y and since $f_X = A \cdot (\bar{B} + B) = A$ and $f_Y = (\bar{A} + A) \cdot B = B$ then the function is the sum of the two loops. The function is then $f = f_X + f_Y$, so $f = A + B$, as expected. A group of two '1's removes a single variable by covering both possible states of that variable, as in

$$f = (\bar{A} + A) \cdot B = B$$

The map was invented by Maurice Karnaugh in the 1950's. He was a telecommunications engineer at Bell Labs in the USA and he wanted to minimise communications circuits. Note that his use of the brain's ability to spot patterns largely preceded the developments of a modern computer.

We don't do analytic minimisation (that's far too painful), we just draw loops and that removes the unnecessary terms/ inputs. To achieve the minimised circuit you should take

the largest possible groups of binary size (of size 1, 2, 4, 8, ...) to reduce the number of inputs in each extracted term. Note that the extracted groups can overlap. The trick is to draw the loops the right way. Let's move on to some bigger K maps. That way we can see more about the minimisation process (a 2-input K map is of zilch use, except to explain what they are). The truth table for 3 inputs has eight output combinations, and that for four input combinations has sixteen. So their K maps have these numbers of slots, but are more compact than truth tables since they are arranged as maps.

A K-map for three inputs is given by

		AB			
		00	01	11	10
C	0				
	1				

and for four inputs

		AB			
		00	01	11	10
CD	00				
	01				
	11				
	10				

Figure 3.9 Layout of 3- and 4-input K-maps

Note the order of the inputs is **not** the same as for a truth table and, for two inputs, the last two columns (or rows) flip from the values expressed in the truth table. This then forces the order of the truth table to reflect the *unit distance criterion* where one input only changes between adjacent cells. If we were to order inputs as in a truth table then this grouping would not perform minimisation since in some places two inputs would change between adjacent cells so they could not be grouped for minimisation purposes.

Let's take a larger K-map, Figure 3.10, and illustrate the minimisation process by considering some of the groups within it. The unit distance criterion allows minimisation by looping the two cells in the top left corner to perform the operation

$$f_X = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot (\bar{D} + D) = \bar{A} \cdot \bar{B} \cdot \bar{C}$$

This is the loop at the top left hand corner, loop X.

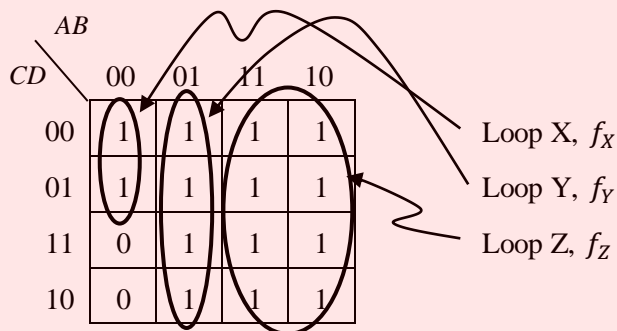


Figure 3.10 Different Sized Loops

A group of four cells removes two variables by covering all four possible combinations, loop Y which covers the four combinations of the inputs C and D , as

$$\begin{aligned} f_Y &= \bar{A} \cdot B \cdot (\bar{C} \cdot \bar{D} + \bar{C} \cdot D + C \cdot \bar{D} + C \cdot D) \\ &= \bar{A} \cdot B \cdot (\bar{C} \cdot (\bar{D} + D) + C \cdot (\bar{D} + D)) \\ &= \bar{A} \cdot B \cdot (\bar{C} \cdot 1 + C \cdot 1) \\ &= \bar{A} \cdot B \cdot (\bar{C} + C) \\ &= \bar{A} \cdot B \end{aligned}$$

If we were to loop eight cells, we can remove three inputs, as in loop Z

$$\begin{aligned} f_Z &= A \cdot (B \cdot (\bar{C} \cdot \bar{D} + \bar{C} \cdot D + C \cdot \bar{D} + C \cdot D) + \bar{B} \cdot (\bar{C} \cdot \bar{D} + \bar{C} \cdot D + C \cdot \bar{D} + C \cdot D)) \\ &= A \end{aligned}$$

Note that we just look for the terms that are not changing, rather than doing the minimisation. (That's what the K-map's for in the first place!) So by **inspection**,

$$f_X = \bar{A} \cdot \bar{B} \cdot \bar{C}, \quad f_Y = \bar{A} \cdot B \quad \text{and} \quad f_Z = A.$$

Given that the inputs are those which go through every possible combination, then we can remove loops of size 1,2,4,8... in **adjacent** cells.

This begs the question: what is adjacent? The K-map wraps round side-to-side and top-to-bottom. The left column is for inputs $AB = 00$ and the rightmost column for $AB = 10$. These then still obey the unit distance criterion and were two elements in the cells they address contain a '1' then they can be looped around the map to reduce and remove the input A leaving \bar{B} . The map also wraps round from top to bottom since only the input C changes between the top and bottom rows. Note that the map is not a sphere; it wraps round from top to bottom, and from side to side. Some find a K-map easier when it is drawn as in Figure 3.11 which is functionally identical to the map drawn before.

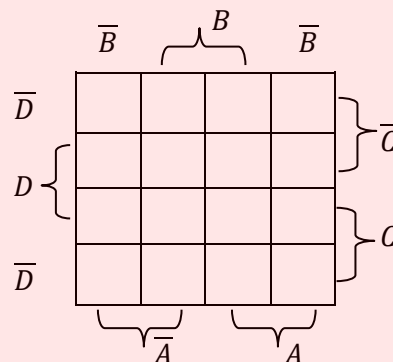
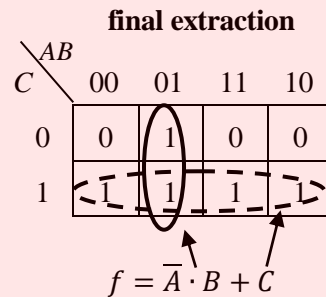
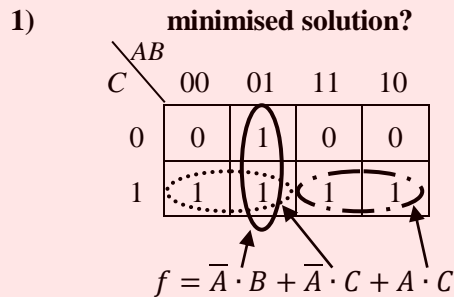


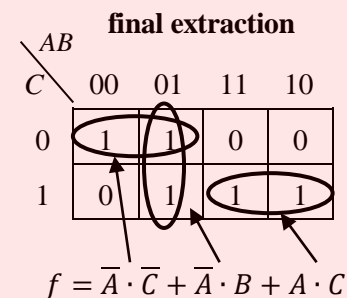
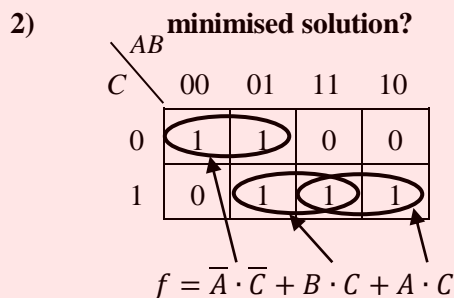
Figure 3.11 Alternative Layout of 4-input K-map

3.3.3 Examples of Karnaugh map design

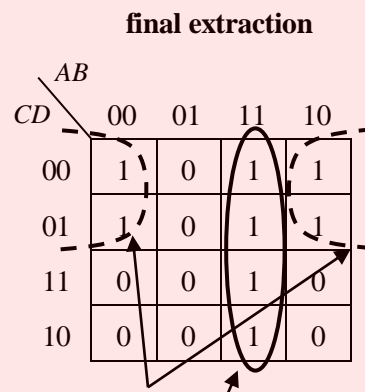
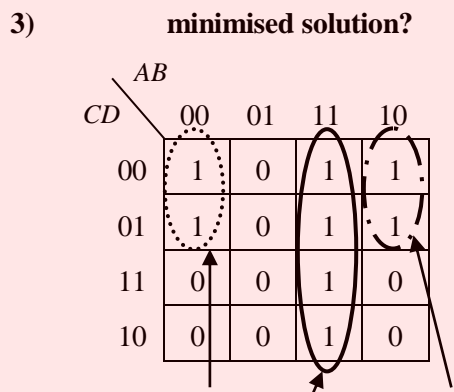
The use of the K-map for minimisation is illustrated by examples showing non-minimised and minimised solutions.



The difference in the extractions is shown by the difference between the dotted lines. Here, a larger group should have been taken in the first extraction (the left hand side) which is reflected in the possibility of algebraically minimising the K-map result. This should not occur since the K-map is designed to avoid it. The second extraction (right hand side) results in the minimised solution.



In the second example, both extractions implement the circuit in minimised form. There is no unique solution: both solutions are equally valid in terms of AND, OR and NOT extraction.

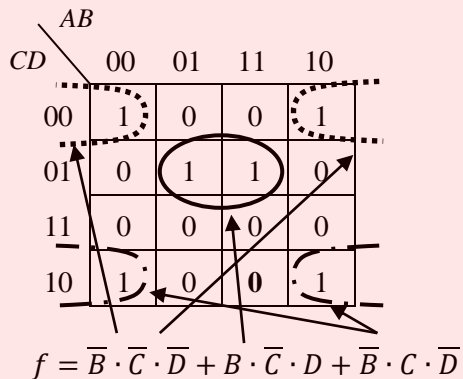


$$f = \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B + A \cdot \bar{B} \cdot \bar{C}$$

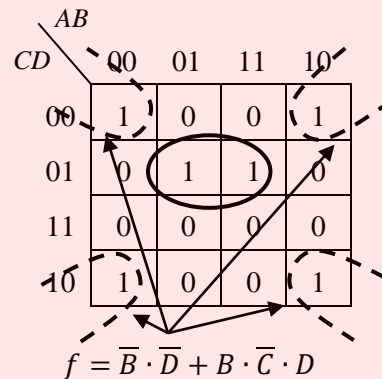
$$f = \bar{B} \cdot \bar{C} + A \cdot B$$

In the third extraction, we have not remembered that the table maps round from side to side and a better, more minimised extraction is possible.

4) **minimised solution?**

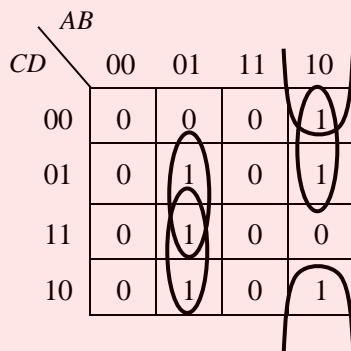


final extraction



In the fourth example, in its non-minimised form we have remembered that the map wraps round from side to side, but not also from top to bottom. The four corner elements therefore form a group that can be extracted together as the term $\bar{B} \cdot \bar{D}$ (removing the inputs A and C). Finally, consider the following K-map

5)



$$f = \bar{A} \cdot B \cdot C + \bar{A} \cdot B \cdot D + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{D}$$

This is John Wonderland's alarm circuit again. Now the minimisation is clear and easy. A minimised software description for John's alarm is:

```
IF ((NOT(alice) AND bo AND (chas OR dave))
    OR (alice AND NOT(bo) AND (NOT(chas) OR NOT(dave)))
    THEN alarm=TRUE
    ELSE alarm=FALSE
```

This clearly follows the original specification. Alternatively

```
alarm = NOT(alice) AND bo AND (chas OR dave)
        OR (alice AND NOT(bo) AND (NOT(chas) OR NOT(dave)))
```

This is clearly the same as the minimised solution.

3.4 Timing considerations and static hazards

Any practical electronic circuit takes time to respond. For gates, the *propagation delay* defines how long before the output changes state after an input. The time for the output to change LOW to HIGH, T_{PLH} , or the time for HIGH to LOW transition, T_{PHL} , are usually of the order of nanoseconds. It is therefore important to consider how many logic gates signals pass through (how much delay there is in the circuit). This is called the *level of the logic system* (not to be confused with logic levels which are the ranges of acceptable voltages for a '1' and a '0'). A 2-level logic system is one where a signal will pass through two gates at maximum from the input to the output. This can be important when considering the **delay** introduced by combinational logic via T_{PHL} or T_{PLH} . A 3-level logic system is shown in Figure 3.12. Here, the speed of the system is dictated by the longest path, which is the one for input *B*. That has at maximum three gates that an input must pass through, so the level of the logic system is three.

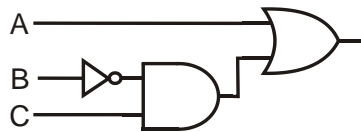


Figure 3.12 3-level logic system

A *hazard* is clearly undesirable in a logic system. It concerns a logic variable at an **incorrect** state often due to **timing considerations** (propagation delay). The propagation delay of an inverter can cause a momentary *glitch* on a circuit output, where it has the wrong value. Consider

$$f = A \cdot \bar{B} + B \cdot C$$

and we shall look at the output state for inputs $A = '1'$ and $C = '1'$ and the input B changes from '1' to '0'. We shall consider the delay from the inverter since that is what affects this circuit's operation. There is also some delay due to the AND gates and the OR gate, but that does not spell potential disaster here, whereas the NOT gate can. Essentially, the circuit output is the OR of two terms, either of which is '1' when B changes and so for $A = '1'$ and $C = '1'$ the output (in theory) should not change at all when B changes from '1' to '0'. In Fig 3.13 the horizontal axis is time and the first column is for the ideal case when \bar{B} changes to '1' precisely when the input B changes to zero. The second column is for the real case where \bar{B} changes to '1' slightly later than when the input B changes to zero

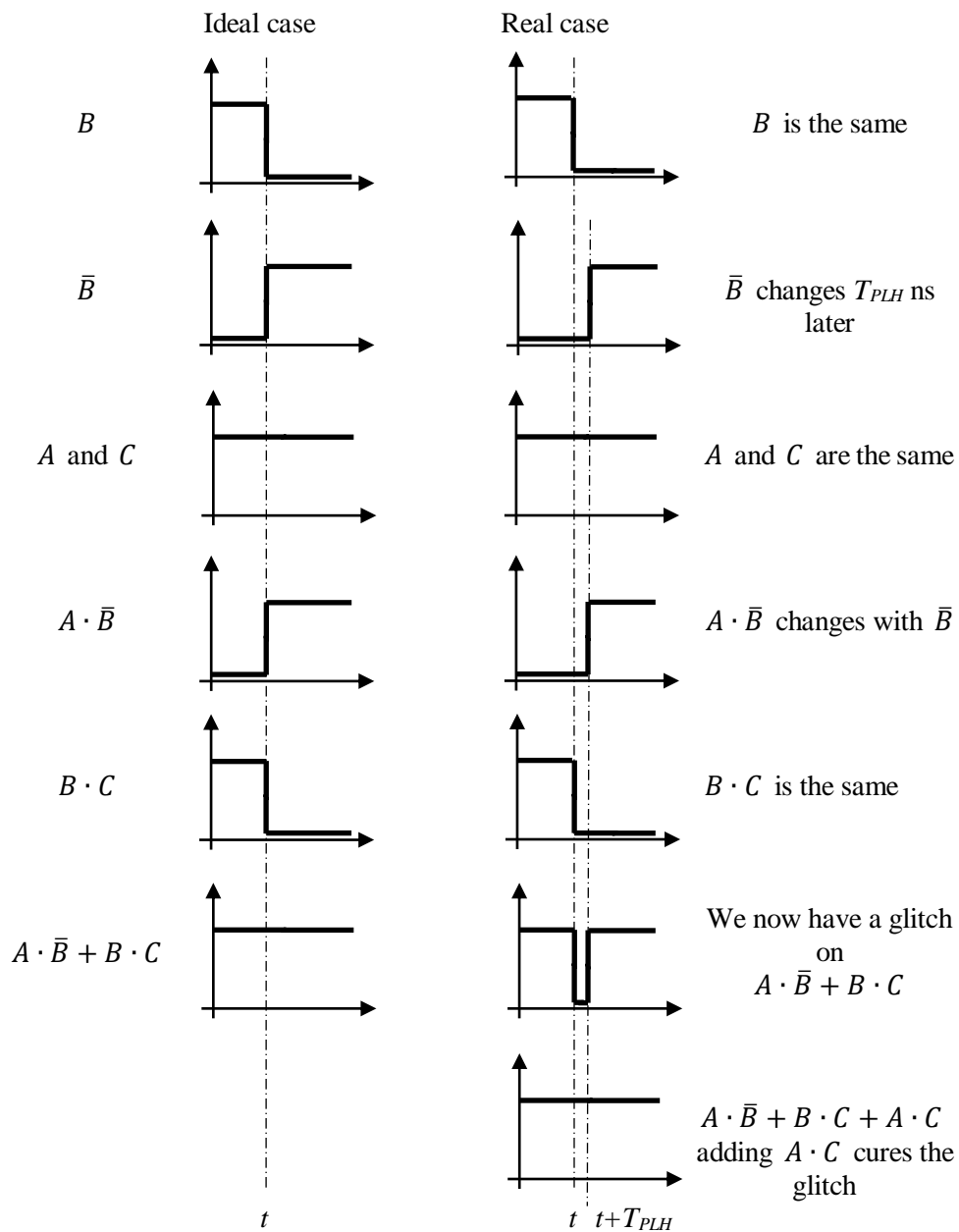


Figure 3.13 Timing and Glitch Generation

The output staying constantly at '1' is true in theory, but in practice \bar{B} is the output of an inverter whose input is B . When B changes from '1' to '0' then the output of the inverter, \bar{B} , changes from '0' to '1' T_{PLH} ns later. There is then a short time during which both $A \cdot \bar{B}$ and $B \cdot C$ are both zero. The output will then be momentarily '0' whilst the change in B propagates through the inverter. This is unwelcome but it is reality, since no circuit can

respond infinitely fast; it is called a *static hazard*. To reduce its effect we can link together all the largest groups extracted for minimisation. The extra terms we introduce are called *bridging terms*. The K-map for $f = A \cdot \bar{B} + B \cdot C$ shows how the bridging term is derived.

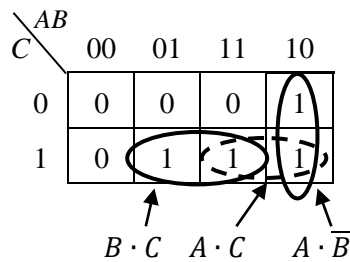


Figure 3.14 Using a Bridging Term

The bridging term $A \cdot C$ is not affected by the change in \bar{B} and its inclusion in the overall function removes the glitch due to the static hazard. The hazard-free implementation is then

$$f_{HF} = B \cdot C + A \cdot C + A \cdot \bar{B}$$

Some further examples show K-maps where bridging terms are not possible, and where they are possible.

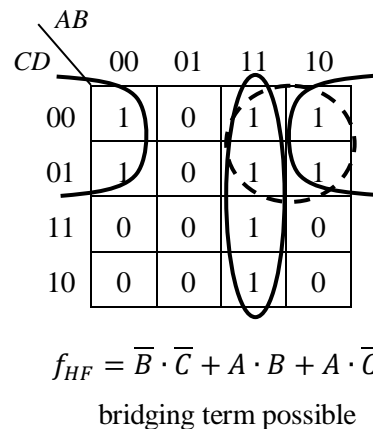
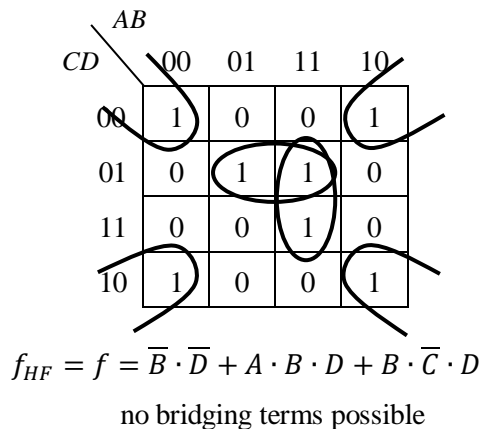


Figure 3.15 Other Bridging Terms

In practice the treatment of hazards is more complex. There are some hazards that are not exposed by determining the bridging terms in K-maps and it is often not possible to link all terms together. It is often better to **wait** until any timing components should have settled, i.e. expect propagation delay as part of life and design around it. The time for which you should wait is given by the worst-case path through the logic, i.e. the path that a signal takes the longest time to traverse.

3.5 Logic implementations

3.5.1 Using NAND or NOR to provide NOT

In Section 3.2 it was stated that De Morgan's law implies that NAND and NOR gates are universal by virtue of the possibility of exchanging AND/OR function. In fact, in a more general sense universality exists so long as a NOT gate is available. Given that for universality we want to build a circuit using NAND (or NOR) gates only, we then need to determine whether we can derive the NOT function using a NAND gate (or a NOR gate). This can be achieved by study of the truth tables.

A	B	$\overline{A \cdot B}$	$\overline{A + B}$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Table 3.1 Inversion by NAND and NOR

By inspection we can see that the output of the NAND gate $\overline{A \cdot B} = \overline{A}$ if the input B equals A (i.e. the two inputs are connected together), or if $B = '1'$ then the output is the inverted form of the input A . Similarly the output of the NOR gate $\overline{A + B} = \overline{A}$ if $B = '0'$ or when $B = A$, giving four possible circuits equivalent to NOT



Figure 3.16 Other Inverter Circuits

The circuits where one input is connected low or high are usually preferred since the input is connected to a power supply, not to the output of another gate. This takes current from the power supply (which is why it is there) rather than from the output of another gate. (Note that the consequences of interconnection on logic level are handled in greater detail in Chapter 4 where we consider the circuits that implement the gates.)

3.5.2 Extracting '0's or '1's?

The notion of universality can be further investigated by considering a K-map, wherein we have so far considered extraction by the '1's only. Naturally there are '0's and these can be grouped to determine the **inverted** function. In a K-map, extracting the '1's gives an (N) AND/OR function whereas by extracting the '0's we achieve AND/NOR. To show this, and in part to aim to try to answer the question as to which is best, we shall start with an example K-map.

$\diagdown AB$

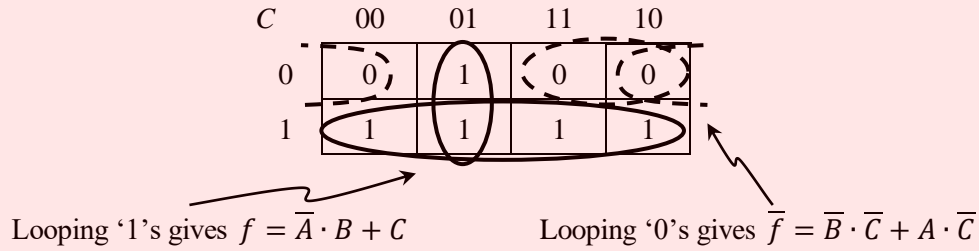


Figure 3.17 Example K-map

By **extracting the '1's** we achieve

$$f = \bar{A} \cdot B + C$$

This can be implemented using NAND gates only by application of De Morgan's law, thereby illustrating the universality of NAND gates: the circuit can be implemented without AND and without OR, just NAND.

$$f = \overline{\overline{\bar{A} \cdot B} \cdot \bar{C}}$$

This requires two 2-input NAND gates (plus two NAND gates to invert A and C as in Section 3.5.1).

By **extracting the '0's** (by looping the '0's in the same way we loop the '1's) we extract the inverse function \bar{f}

$$\bar{f} = \bar{B} \cdot \bar{C} + A \cdot \bar{C}$$

and by inverting both sides

$$f = \overline{\bar{B} \cdot \bar{C} + A \cdot \bar{C}}$$

This is an AND/NOR function. We can again achieve an entirely NOR construction by applying De Morgan's law, again re-illustrating that NAND and NOR gates are both universal.

$$f = \overline{\overline{B + C} \cdot \overline{\bar{A} + C}}$$

This requires three two input NOR gates, plus one NOR gate to invert A. To demonstrate the duality between extraction by the '0's and by the '1's, By applying De Morgan's law again

$$f = \overline{\overline{B + C} \cdot \overline{\bar{A} + C}}$$

which gives

$$f = (B + C) \cdot (\bar{A} + C)$$

by expansion we obtain

$$f = \bar{A} \cdot B + \bar{A} \cdot C + B \cdot C + C \cdot C$$

and by minimisation

$$f = \bar{A} \cdot B + C$$

This returns us to the function extracted by the '1's hence showing the duality of the two

extractions. Which is best? That rather depends on an overall system. In this case extracting the '1' leads to one gate less, but that is not always the case.

“Contrariwise”, continued Tweedledee, “if it was so, it might be, and if it were, it would be: but as it ain’t, it ain’t. That’s logic.”
(Lewis Carroll)

3.6 Terminology

The extraction of a function by grouping inputs using AND gates whose outputs are grouped using OR is called the *sum of products* (SOP) form, e.g.

$$f = \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C}$$

It conforms rather nicely with the way we think about things. There is conversely a *product of sums*, e.g.

$$f = (\bar{A} + B) \cdot (A + C)$$

A product term containing one occurrence of **every** variable is known as a *minterm* or a *canonical product term*. A sum term containing one occurrence of **every** variable is known as a *maxterm* or a *canonical sum term*. A function is expressed as OR of distinct minterms is known as *canonical sum of products*; one expressed as AND of distinct maxterms is known as *canonical product of sums*. For those with masochistic tendencies, canonical SOP is *disjunctive normal form*, canonical POS is *conjunctive normal form*. For further developments let us consider a function given by a truth table in Table 3.2.

Minterm number	<i>J</i>	<i>K</i>	<i>L</i>	<i>f</i>
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

Table 3.2 Example Logic Function showing Binary Truth Table Index

Here the decimal number corresponds to the binary representation of the input coding – its minterm number. This function can be expressed using a decimal index for each term in the truth table. The minterms are the cases for which the function is '1' and collecting these in sum of products (SOP) form is

$$f = \bar{J} \cdot K \cdot L + J \cdot \bar{K} \cdot \bar{L} + J \cdot \bar{K} \cdot L + J \cdot K \cdot \bar{L} + J \cdot K \cdot L$$

which can be expressed in a shortened manner using the decimal index as the canonical SOP

$$f = \sum_{JKL} (3,4,5,6,7)$$

These indices are a function of expressing JKL as a binary number where J is the MSB and L is the LSB, as in Table 3.2. The function can also be expressed as a product of sums

$$f = (J + K + L) \cdot (J + K + \bar{L}) \cdot (J + \bar{K} + L)$$

And in shortened form, the canonical POS is

$$f = \prod_{JKL} (0,1,2)$$

So how are these related? By extracting the '0's

$$\bar{f} = \bar{J} \cdot \bar{K} \cdot \bar{L} + \bar{J} \cdot \bar{K} \cdot L + \bar{J} \cdot K \cdot \bar{L}$$

By inversion

$$f = \overline{\bar{J} \cdot \bar{K} \cdot \bar{L} + \bar{J} \cdot \bar{K} \cdot L + \bar{J} \cdot K \cdot \bar{L}}$$

Now, by De Morgan's law

$$f = \overline{\bar{J} \cdot \bar{K} \cdot \bar{L}} \cdot \overline{\bar{J} \cdot \bar{K} \cdot L} \cdot \overline{\bar{J} \cdot K \cdot \bar{L}}$$

Again, by De Morgan's law

$$f = (J + K + L) \cdot (J + K + \bar{L}) \cdot (J + \bar{K} + L)$$

Phew, we're back to the expression of product of sums. This is the canonical product of sums specification. Note the 'inversion' in canonical POS expression and the relation between SOP and POS. You often get logic functions expressed in POS and SOP form (using summation or product symbols, respectively) - it is a very compact description, and one not prone to misinterpretation.

Some of these terms might seem a bit pedantic and tortuous. However, they do allow for unambiguous and compact specification, and so are seen in professional use. Some terms found in some textbooks have been omitted here, as they are rather unnecessary, e.g. an "isolated - cell" is what it says, it's a canonical SOP term for which no minimisation can be achieved. Erm, obvious?!

By K-map for the original function f , Fig 3.18,

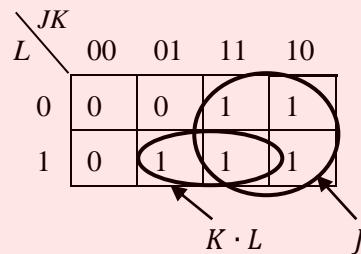


Figure 3.18 Example K-map

so $f = J + K \cdot L$ this is the condensed or minimised form (i.e. a *minimal sum*). The terms J and $K \cdot L$ are non-canonical (they do not contain all the variables) and are called *prime implicants*; these are the largest loops you can draw in the K-map. For a K-map containing four prime implicants, Figure 3.19, the four could be included in an implementation aimed to avoid static hazards.

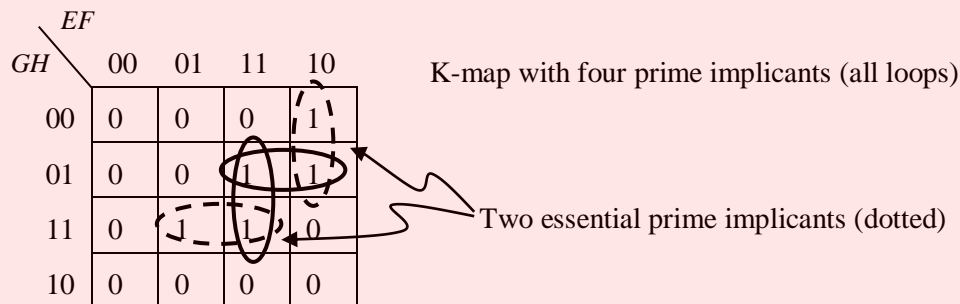


Figure 3.19 Prime Implicants and Essential Prime Implicants

Note that the minimal sum does not necessarily contain all prime implicants; a hazard-free one (probably) does. Terms which must be included in the minimal sum are called *essential prime implicants* (EPIs on the diagram, and these are the dotted groups) and are the only prime implicants which cover a particular '1' cell. Other '1's may be covered by two or more prime implicants, **one** of which must be included in the minimal sum.

3.7 Design Example

Let us consider the following design example. Essentially, it concerns designing a circuit to compare the magnitude of two binary numbers and to provide signals indicating the results.

A circuit is required which will compare the magnitude of two 2-bit binary numbers AB and CD . Each number may have a value 0 (when $A=0$, $B=0$ and $C=0$ and $D=0$), 1 (when $A=0$, $B=1$ and $C=0$, $D=1$), 2 (when $A=1$, $B=0$ and $C=1$, $D=0$), or 3 (when $A=1$, $B=1$ and $C=1$, $D=1$). The complemented outputs of A , B , C and D are available. Circuit outputs G , E and L are to be TRUE ('1') when the number AB is greater than, equal to, or less than the number CD , respectively. The outputs should be FALSE ('0') otherwise.

- produce a K-map for each output (G , L and E)
- implement G , L , and E using a minimum number of AND, OR or NOT gates
- implement G using NAND gates only
- implement E using NOR gates only
- show how L may be obtained using an 8-1 multiplexer.
- determine the relationship between G and the function

$$f = A \cdot \bar{C} \cdot (B \oplus D) + \bar{A} + B + C + D + A \cdot \bar{C} \cdot D + B \cdot D \cdot \overline{A \oplus C}$$

a) the K-maps are

CD \ AB	G			
	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

CD \ AB	E			
	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

CD \ AB	L			
	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

As a cross-check, if we were to place these K-maps on top of each other, we would find that every box has just one '1'. That's exactly as the specification. If the specification had been $G = '1'$ for $AB \geq CD$ or $L = '1'$ for $AB \leq CD$ then this would not have been the case. The K-map for E is routine: each occupied cell is where the inputs are the same and this must be the diagonal.

b) The minimal extraction for each is by the prime implicants

CD \ AB	G			
	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$$G = A \cdot \bar{C} + B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot \bar{D}$$

CD \ AB	E			
	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

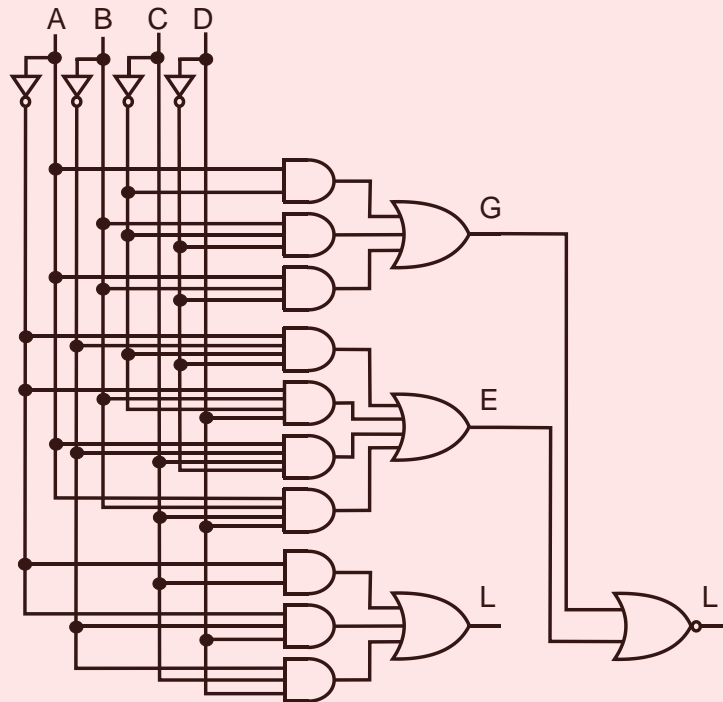
$$E = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot B \cdot C \cdot D$$

CD \ AB	L			
	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$$L = \bar{A} \cdot C + \bar{B} \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot D$$

Note that there is some symmetry between the expressions for G and for L , which is to be expected. Extracting E from a K-map appears to be overkill and we could write the expression by inspection. As usual, we often find this out afterwards.

We can check these circuits using Logisim



Here, the function L has been implemented both by the K-map extraction and by combining G and E . This is by NOR since L is not G AND not E and by De Morgan's law this becomes a NOR function.

$$L = \bar{G} \cdot \bar{E} = \overline{G + E}$$

c) To implement G using NAND gates only we use De Morgan's law

$$G = A \cdot \bar{C} + B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot \bar{D} = \overline{\overline{A \cdot \bar{C} \cdot B \cdot \bar{C} \cdot \bar{D} \cdot A \cdot B \cdot \bar{D}}}$$

This can be achieved using NAND gates only since we can use NAND to give NOT by holding one input high $\overline{A \cdot 1} = \bar{A}$. If we needed a minimised implementation we would need to consider the K-map, but that is not the case here.

d) To implement E using NOR gates we can either twice invert the function already extracted (since a NOR gate with one input LOW acts as an inverter), or extract the '0's. By inverting twice the previous canonical SOP extraction for E we obtain

$$E = \overline{\overline{\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot B \cdot C \cdot D}}$$

Then by applying inverting each term twice (ready for De Morgan's law) we obtain

$$E = \overline{\overline{\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}} + \overline{\overline{\bar{A} \cdot B \cdot \bar{C} \cdot D}} + \overline{\overline{A \cdot \bar{B} \cdot C \cdot \bar{D}}} + \overline{\overline{A \cdot B \cdot C \cdot D}}}$$

So by De Morgan's law

$$E = \overline{\overline{\bar{A} + \bar{B} + \bar{C} + \bar{D}} + \overline{\overline{A + B + C + D}} + \overline{\overline{A + B + \bar{C} + D}} + \overline{\overline{A + \bar{B} + \bar{C} + \bar{D}}}}$$

And this uses NOR gates only. Alternatively, by extracting the '0's

$$\bar{E} = B \cdot \bar{D} + A \cdot \bar{C} + \bar{A} \cdot C + \bar{B} \cdot D$$

by applying De Morgan's to each term $\bar{E} = \overline{\bar{B} + D} + \overline{\bar{A} + C} + \overline{A + \bar{C}} + \overline{B + \bar{D}}$

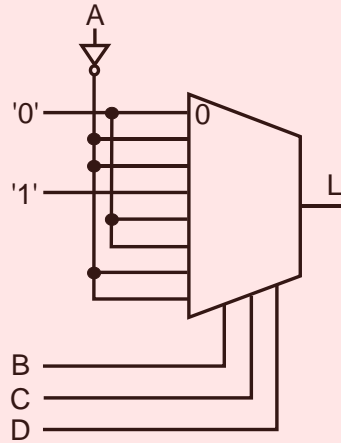
and by inverting each side $E = \overline{\overline{\bar{B} + D} + \overline{\bar{A} + C} + \overline{A + \bar{C}} + \overline{B + \bar{D}}}$

and this gives a much smaller circuit.

e) To implement L using an 8-1 multiplexer we can view the function in non-minimised form as

$$L = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{A} \cdot \bar{B} \cdot C \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot C \cdot D + \bar{A} \cdot B \cdot C \cdot D + \bar{A} \cdot B \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot C \cdot D$$

This can be implemented in a multiplexer with control lines BCD addressing the eight channels. For the channels addressed by the first four terms, corresponding to multiplexer input channels with $BCD = 001, 010, 111$ and 110 respectively, the multiplexer inputs can be fed with the inverted input \bar{A} , since this is common to all four terms. For the last term in L the output is '1' if $B = '0'$, $C = '1'$ and $D = '1'$ and so we connect a high input to the multiplexer input addressed by that term. All other input channels addressed by uncovered combinations of BCD (in the expression for L) are connected to '0'.



If we were required to use a smaller multiplexer then more logic would be used at the input stage; should a larger multiplexer be used (i.e. a 16-1 line multiplexer) then the inputs would become '0' or '1' and the input combinations would select chosen logic levels, those specified in the K-map for L .

f) Essentially, we use logic laws to determine a sum of products and we use this to complete a K-map.

$$f = A \cdot \bar{C} \cdot (B \oplus D) + \overline{\bar{A} + B + C + D} + A \cdot \bar{C} \cdot D + B \cdot D \cdot \overline{A \oplus C}$$

By the function of EXOR and EXNOR

$$f = A \cdot \bar{C} \cdot (B \cdot \bar{D} + \bar{B} \cdot D) + \overline{\bar{A} + B + C + D} + A \cdot \bar{C} \cdot D + B \cdot D \cdot (\bar{A} \cdot \bar{C} + A \cdot C)$$

And then by expanding these terms

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

By De Morgan's law

$$f = A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot C \cdot D$$

and then use these terms to make up a K-map for f and it is the same as the one for G .

3.8 Algorithmic Minimisation (Quine-McCluskey and Espresso)

Obviously, minimisation is tedious, repetitive and prone to error. That's what computers are good at, and that's why they are great at minimising logic circuits. Obviously, we need to enter the circuit data in some way into a computer to achieve this, and in modern designs this is achieved by a software specification. We shall start with the *Quine-McCluskey method*, which is a method of prime implicants. The first stage is to determine all the prime implicants, then we determine the smallest set of prime implicants which satisfy the circuit's function,

thereby minimising it. This set must include all the prime implicants. We shall illustrate the procedure using Wonderland's alarm from Section 2.6, which was

$$\text{alarm} = \bar{A} \cdot B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot D + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D}$$

and this was minimised in Section 3.3.3 as

		<i>AB</i>			
		00	01	11	10
<i>CD</i>	00	0	0	0	1
	01	0	1	0	1
	11	0	1	0	0
	10	0	1	0	1

$$\text{alarm} = \bar{A} \cdot B \cdot C + \bar{A} \cdot B \cdot D + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{D}$$

The minterms are

$$\text{alarm} = \sum_{ABCD} (5,6,7,8,9,10)$$

Listing these minterms in logic form we have

#	A	B	C	D
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

We then work out how the groups can combine, developing a list of terms that can be combined (algorithmically mimicking groups of two in the K-map) and replacing the bit that can be ignored with a X. We then get

terms	A	B	C	D
(5,7)	0	1	X	1
(6,7)	0	1	1	X
(8,9)	1	0	0	X
(8,10)	1	0	X	0

We then repeat this iteration as far as we can (first by looking for groups of 4 in the K-map, and then 16 et seq.). In this case, we do not have any further groups so our minimised function is

$$alarm = 01X1 + 011X + 100X + 10X0 = \bar{A} \cdot B \cdot D + \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{D}$$

Published papers can include a photograph of the authors (if provided). McCluskey always favoured a different hat in these photos.
<http://www-crc.stanford.edu/users/ejm/hats.html>
Well done Ed!



This procedure can be automated and then implemented as a computer algorithm. This procedure will clearly explode when there are more inputs. If $n = 32$ there could be more than 6.5×10^{15} prime implicants. Ok, storage is cheap now, and so is processing power. That tends to mean our designs are more complex so we need an optimised search procedure. The one that everyone uses is called the *Espresso* system, and it came from IBM in the 1980's. This is heuristic, but operates well. Essentially, it looks for the ON and for the OFF terms which best minimise the function. The algorithm finds pretty well standard usage, and there are several variants. Well done Robert Brayton: as ever history tends not to record gratitude or honours for engineers (remember from Chapter 1 an engineer called Shannon who invented computers). Brayton has plenty of academic awards, but as ever, people probably know the phrase “Espresso” more than “Brayton”.

3.9 Concluding comments and further reading

In this chapter we have seen the development of logic from its specification status to a circuit implementation. Many of the themes presented in this chapter will be manifest later in this book. The development of minimisation has studied some by-hand techniques and some of the algorithmic techniques. These techniques actually develop into computer programs which can be used for circuits with large numbers of inputs. The K-map is very restricted in this respect. Imagine drawing a K-map for a function of ten inputs with 1024 spaces, and then you have to start finding the loops! It is sufficient as a hand tool and gives useful insight, but there its utility ends. There again, the algebraic techniques really need a computer. Finally, we have seen some of the basic building blocks and been introduced to the design of combinational logic systems. These will be used throughout the book, particularly in the design examples. In fact, they are a notational convenience, a sort of functional shortcut in the design process. In practical designs, these will be subject to minimisation. However it is worth noting that regular layout can lead to more efficient minimisation in VLSI than can be achieved with a

fully minimised design. In this respect, design and minimisation are topics of continuing interest and research (see for example Brayton, R. K. et al, *Logic Minimisation Algorithms for VLSI Synthesis*, [Brayton84] (you just saw this name above...). For the academics (historians?), try [Karnaugh53], the Espresso method is at [McGeer93]; for Quine-McCluskey [McCluskey56]. Another topic of major interest is the specification of circuits and their operation. A software specification clearly expresses this, which is itself another avenue for further investigation.

3.10 Chapter 3 Questions (Solutions in Appendix 1)

Q3.1 Implement the function

$$f = \sum_{ABCD} (4,6,9,11)$$

using a minimum number of standard logic gates (AND, OR, NOT, NAND or NOR) which can have any number of inputs. Implement it again using NOR gates only.

Q3.2 Implement the function

$$f = \prod_{ABCD} (0,2,4,5,6,7,11,15)$$

assuming that complemented inputs are available, but without static hazards using a minimum number of standard logic gates (AND, OR, NOT, NAND or NOR) which can have any number of inputs.

Q3.3 Implement the function

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

using a minimum number of standard logic gates (AND, OR, NOT, NAND or NOR) and express your solution in sum of products form.

Q3.4 Implement the function

$$f = \overline{B} \cdot \overline{D} \cdot \overline{A} \oplus \overline{C} + B \cdot D \cdot \overline{A} \oplus \overline{C} + A \cdot C \cdot \overline{B} \cdot \overline{D} + \overline{B} \cdot (A \cdot C + \overline{A} \cdot \overline{C}) + \overline{A} \cdot \overline{C} \cdot \overline{B} \cdot \overline{D}$$

using a minimum number of standard logic gates (AND, OR, NOT, NAND or NOR) and express your solution in product of sums form.

Q3.5 What is the level of logic associated with the function

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

when implemented using standard logic gates with any number of inputs. If the T_{PLH} for each combinational logic gate is 10 ns and T_{PHL} for each combinational logic gate is 11 ns, what is the worst case propagation delay through the circuit?

Q3.6 Identify the static hazards which might be associated with a direct implementation of

the following expression

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

Using a K-map show how these static hazards might be removed. The function is required as part of a control system and the designer needs to know how long after the inputs have changed the output can be used. For all gates T_{PHL} is 9 ns and T_{PLH} is 10 ns. By considering the level of logic estimate how long after the inputs change the output of the original function and the output of your hazard-free one, can be used by the designer.

Chapter 4 Logic circuits

4.1 Logic circuit characteristics

This chapter describes what is inside an integrated circuit. Integrated circuits are made from *switching devices*. These switching devices connect together inputs to implement logic functions. An ideal switching device would respond infinitely fast, consume no power, and be physically extremely small. This is practically impossible to achieve and any switching circuit is a compromise between these factors. The *logic technology* concerns the physical implementation of the switching circuits and offers a distinct set of *performance characteristics*. We shall first investigate the performance characteristics of particular interest to integrated circuit technology, then go on to look at switching circuits, and finally consider the logic technologies which are currently available.

4.1.1 Transfer characteristics

The performance characteristics can be developed from an **ideal** logic gate. The ideal inverter connects its output HIGH to the positive supply rail for a LOW input. For a HIGH input the output is connected LOW to the negative voltage supply, usually ground. The input then acts adversely on a pair of switches; one connects the output HIGH, the other connects the output LOW, shown in Figure 4.1.

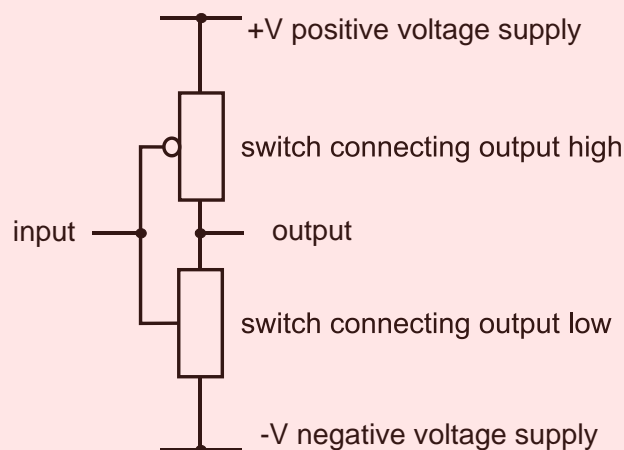


Figure 4.1 Ideal Inverter

Since the input acts adversely on either switch, the upper switch is turned ON for a LOW input which is signified by the inversion symbol attached to the input to the switch. Further

gates, such as AND and NAND, require appropriate combination of the input signals to control the output switches and an ideal logic gate is shown in Figure 4.2.

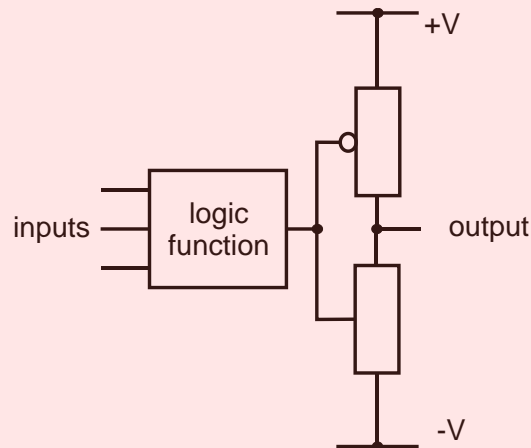


Figure 4.2 Ideal Combinational Logic Gate

Ideally, the switches connect the output (voltage) level HIGH or LOW in response to an input voltage level. We are then considering *level logic* where voltage levels are used to signify logic variables. In this introduction we shall use *positive logic* which is a convention that a HIGH voltage level, usually +5 V, represents a logic '1'. Accordingly, in positive logic a LOW level, usually 0 V, signifies a logic '0'. The *transfer characteristic* shows the relationship between inputs and outputs, and for (shown in Figure 4.3(a)) an **ideal** inverter the transfer relationship has only two possible values for the output. The slope of the line between the two output states is vertical, which implies that the ideal inverter has an infinite gain in the transition region.

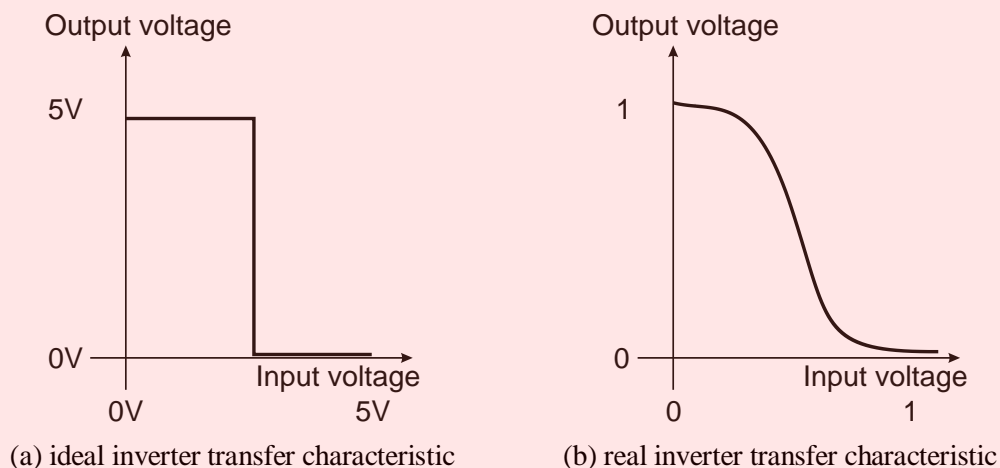


Figure 4.3 Transfer Characteristics

This gain is impossible to achieve in practice. Also, the use of a single voltage level to signify a '1' or a '0' is very difficult to achieve. Most circuits use a range of voltage levels to

represent logic signals. The transfer characteristic then relates ranges of input and output voltage levels representing '1' and '0'. The transfer characteristic of a **real** inverter then differs from that of its ideal counterpart, as shown in Figure 4.3(b).

Manufacturers have to guarantee a device's performance limits. These include the electrical and switching characteristics which can be thought of as DC and AC parameters respectively.

For the HIGH state, the logic levels most commonly encountered in logic design range from 2.0 V to 5.0 V and for a LOW state the level is in between 0 V to 0.8 V to indicate a '0'. This depends on the power supplies that are used. Any electronic circuit requires a positive and negative power supply and for a switching circuit these supplies only need to differ in magnitude. The power supply clearly affects the voltage levels used to represent a '1' and a '0'. Should two logic technologies use different power supplies then the logic levels in each will be different and if the two technologies need to be connected together (i.e. a circuit has been designed to take advantage of particular performance characteristics of each logic technology) then this will pose difficulty in *interfacing* (connecting together) the two circuits.

4.1.2 Performance characteristics

Manufacturers specify *guaranteed output levels* and *specified input levels*. This is a guarantee that if an input is within the range specified for the input levels, then the output of the circuit is guaranteed to be within a range specified for the output voltage levels. If you buy a circuit which does not conform to this specification you can return it to the manufacturer for replacement.

If an input voltage level falls outside the range specified for a valid logic level then it naturally becomes invalid and the circuit will not operate correctly. As in Figure 4.4 this might occur because noise has corrupted the transmission channel connecting two logic gates.

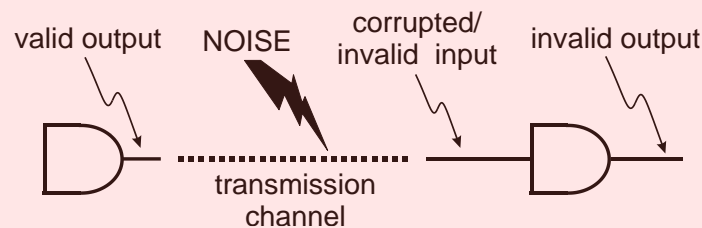


Figure 4.4 Noise in Transmission Channels

Noise can be induced by electromagnetic radiation from an external source which is of sufficient magnitude to force the voltage level on the transmission channel to become invalid. In reality we actually live in a fog of electromagnetic radiation and it is unsurprising to find that some gets picked up in logic circuits. The designers of integrated circuits accommodate this by introducing a *noise margin* into a circuit; this is a measure of by how much a valid signal can be corrupted by noise and still be interpreted correctly by the logic circuit. This can be summarised by relating the output levels to the input voltage levels.

The noise margin for a HIGH level, N_{MH} , is a measure of how much noise can be added to a signal which will still be interpreted as a valid '1', as shown in Figure 4.5. It is the difference between the minimum level for a valid output '1' and the (specified) minimum level for a valid input '1'. Note that in the HIGH state we are concerned with noise that **reduces** the output level. The noise margin for a LOW level, N_{ML} , concerns noise that **increases** an output level and is equal to the difference between a valid output '0' and a valid input '0'. N_{ML} is then a measure of how much noise can be added to a LOW logic signal whilst it is still interpreted correctly by later combinational circuitry. Logic technologies usually specify a noise margin which is equal in both high and low states. There is an uncertainty region between the maximum value for a LOW input and the minimum value for a HIGH input; if a circuit input falls within the uncertainty region then the output is not guaranteed by the manufacturer. This situation should be avoided since it can result in unreliable circuit operation. It is clearly desirable to have the largest possible noise margin, but some logic technologies can offer only a small noise margin while possessing other attributes.

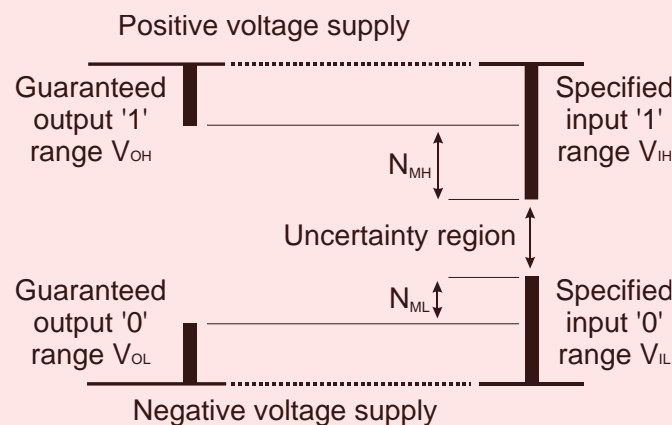


Figure 4.5 Noise Margins

The logic level might become invalid because of contamination by noise. It might also become invalid because an output has been connected to too many inputs. The maximum number of gate inputs which can be connected to the output of a single gate is usually expressed as the *fan-out* of that gate.

Some texts refer to fan-in as well and there are a number of conflicting definitions. The most convincing definition of fan-in is the maximum number of logic inputs that a gate can accept. (One egregious definition is that it is the number of input pins - and that's plain daft!).

Fan-out can be explained in terms of valid logic levels. In an ideal gate a HIGH output is connected to the positive supply through a perfect switch. A perfect switch is one which has no resistance when it is switched ON (a short circuit) and infinitely high resistance (an open circuit) when it is switched OFF. When it is ON there is no voltage drop across it since it has no resistance. Logic circuits do not use perfect switches, they use switches which

approximate the perfect switch. Since they only approximate their perfect version they have resistance. Their resistance implies that the voltage across them increases with increase in the current through them. When the voltage drop increases the output voltage falls, and when the output is connected to too many inputs the voltage will fall beneath that guaranteed as a valid output. The increase in current is due to increasing the number of gates connected to the output. Should we connect more gates than the specified maximum (the fan-out of the device), the output voltage level will not be valid as specified for the device.

If the difference between logic levels is small then the noise margin is bound to be very small as well. The advantage associated with small difference between logic levels is **speed**, since the logic technology has only to switch the output by a small amount. Speed is usually expressed in terms of *propagation delay* which is a measure of how long a change in the inputs takes to propagate through to the circuit output. The propagation delay can differ for an output changing from LOW to HIGH or from HIGH to LOW. T_{PLH} measures the time between a change in inputs and the subsequent change in output from '0' to '1'. T_{PHL} is a measure of the time taken for the outputs to change from HIGH to LOW, '1' to '0', in response to a change in the inputs. This is the time that gave rise to the static hazard in Section 3.4. The timing is illustrated in Figure 4.6 where the time is measured from when the inputs changed. The duration is measured to the 50% point, where the output is half way between its initial and its final states.

The propagation delay largely arises from the need to shift charge and that takes time: less to shift means less time is needed.

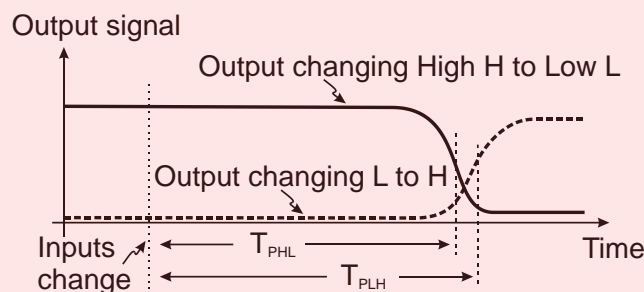


Figure 4.6 Propagation Delays

It is advantageous to have the fastest possible switching time. This will usually incur a higher *power consumption* since this is inherent in a switching process. **Speed** and **power consumption** are often the most important factors when considering different logic technologies. They are often combined in a *speed/power product* which is formulated by multiplying the worst case propagation delay (the longest time taken by a device to respond) by the power consumed on average by the device. This can then be used as a performance metric to compare logic technologies.

The **main** criteria for a logic technology are then the speed it can achieve (the smallest propagation delay), and the power consumed by the device. The secondary performance characteristics are the logic levels, noise margin and fan-out. The **secondary** criteria might dominate choice in a particular application since they dominate circuit implementation. All of

these factors are a compromise; high speed often incurs high power consumption, good fan-out usually implies wide noise margins but since the logic levels are wide the speed is then reduced. Any logic technology offers a compromise between these factors and this compromise depends on the switching elements used to implement the logic function, together with implementation criteria, in particular the logic levels.

4.2 Switching devices

4.2.1 The diode

The simplest form of electronic switch is a *semiconductor diode*. This is essentially a switch which can be turned ON or OFF. Its symbol indicates the direction in which current can pass through it, shown in Figure 4.7.

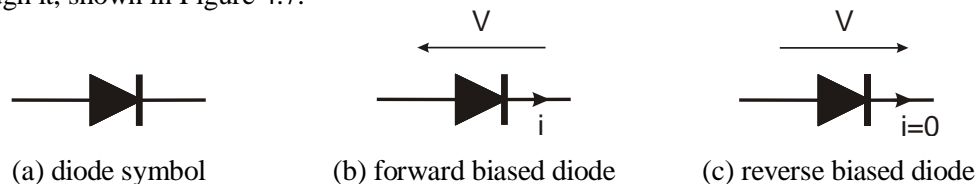


Figure 4.7 The Diode

If we apply a voltage consistent with the current direction (the positive connection, the anode, is more positive than the negative connection, the cathode) then current can pass through the device and it is switched ON: in this condition the diode is termed *forward biased*. If we apply a voltage the other way round (the cathode is more positive than the anode) no current can flow and the diode is switched OFF: in this condition the diode is termed *reverse biased*. The *equivalent circuit* is a model of the circuit and for a forward biased diode it is a short circuit. An equivalent circuit for a reverse biased diode is an open circuit (since no current can pass). There is actually a **threshold** voltage for a forward biased diode; the threshold voltage must be exceeded before current can pass and this is called the *cut-in voltage* (or *knee voltage*). This can be seen from the current/forward voltage characteristic which illustrates that a diode does not conduct before the threshold voltage is reached; after the threshold voltage is exceeded, the current increases rapidly.

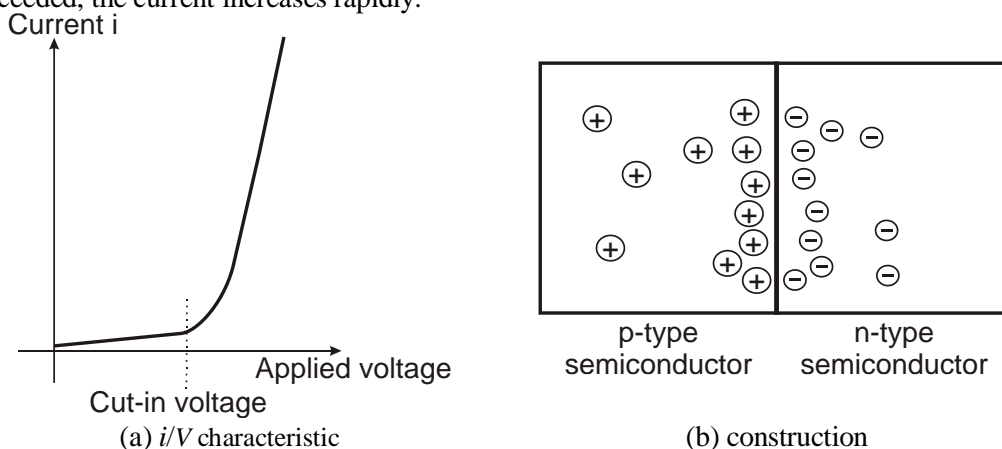


Figure 4.8 The Semiconductor Diode

One of the oldest canards about electronics is that it is the only subject whose name derives from something we have never seen (OK some, like historians, will disagree). If we could see an electron, what would it look like?

The diode is made by joining two types of semiconductor shown in Fig 4.8. One of these is *n-type semiconductor* (negative-type) which is made to have more electrons than normal giving a net negative charge. The surfeit of electrons is introduced by a process called *doping*. Doping can also cause a deficit of electrons and hence a surfeit of *holes* (essentially the hole left by the absent electron) which gives a net positive charge as in a (doped) p-type semiconductor. If we join these two types of semiconductor we now have a pair of adjacent materials, one with a net negative charge, the other with a net positive charge; this is called a *pn junction*. Even without any externally applied voltage, electrons in the n-type region will be attracted to the p-type region. There will also be a thermal effect causing electrons to jump the other way. With no externally applied voltage the current generated by thermal effects will balance the current generated by charge effects. When the diode is forward biased, the potential of the p-type region is raised, and so more electrons will migrate to the p-type region causing current to flow. When the potential between the p-type and n-type regions is below the cut-in voltage, it is insufficient to attract electrons in large numbers and no net current flows through the diode.

The current is actually exponentially related to the forward bias voltage and thus can increase dramatically. There is a limit to this current since diodes can break. Also, the reverse breakdown voltage specifies a reverse bias sufficiently large for the diode to pass current in the opposite direction.

Detailed examination of a diode's performance can be found in texts referenced at the end of the chapter. The function of a diode in this text is as a switching circuit; we want it to switch ON and OFF. A diode can turn ON if forward biased and will turn OFF if reverse biased. This is equivalent to applying a logic '1' and '0'.



Figure 4.9 The Diode as a Switching Circuit

4.2.2 Bipolar transistors

A diode has limited functionality in logic circuits and the *transistor* plays a more important role, primarily because it is a three terminal device and there are two forms:

- (i) *bipolar* - conduction uses both holes and electrons (as in a diode);
 and (ii) *unipolar* - conduction uses one type of carrier only, either holes or electrons.

A bipolar transistor can be made by fusing two regions of n-type semiconductor with one region of p-type. The connections to the n-type regions are termed the *emitter* and the *collector*, and the p-type region is called the *base*. This gives an *npn transistor*. It is made from the two types of semiconductor in three regions. Two of the regions are n-type and the other one is p-type. The arrangement is shown in Figure 4.10(a) and this can be viewed as two diodes (of the form of Figure 4.8) which are back-to-back. Its symbol, Figure 4.10(b) shows a three terminal device with a diode between the base and emitter.

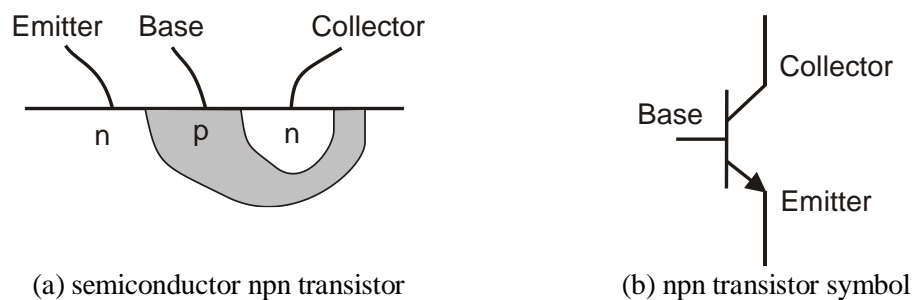


Figure 4.10 Transistors

The diode between the *base-emitter junction* effectively controls the operation of the device. If the base-emitter junction is forward biased then electrons are attracted from the emitter into the base and are swept into the collector. Forward biasing the base-emitter junction then causes a collector current to flow through the transistor. If the base-emitter junction potential is below a threshold value then there will be a small current from base to emitter due to charge effects (electrons attracted into the p-type) and a small current from emitter to base (from electrons migrating due to thermal effects). If the base-emitter junction is not forward biased then there will be no net current since the thermal effects will balance the charge effects and there will then be no collector current. We can use the base-emitter voltage to switch the collector current on or off. When we introduce a power supply for the collector current, together with a resistor (essentially to limit the collector current), we obtain a circuit which can be used as a switch, shown in Figure 4.11 where the collector, emitter and base have the abbreviated symbols C, E and B, respectively.

Note that the two pn junctions imply that the transistor can be modelled as two diodes. This is in fact the basis of the Ebers-Moll model of a transistor.

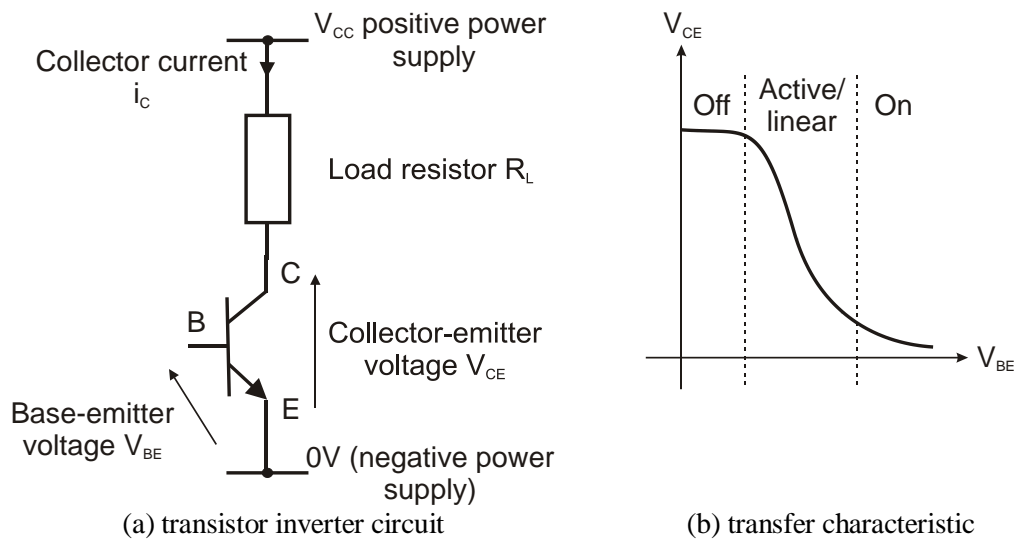


Figure 4.11 Transistor Inverter

The collector current will vary with the base-emitter voltage which in turn causes the collector-emitter voltage to vary. As we forward bias the base-emitter junction the collector current increases and so the collector voltage falls. When the transistor is turned OFF there is no collector current, so the transistor is OFF, and the collector is connected to the positive voltage supply, so V_{CE} is HIGH. If the base-emitter voltage, V_{BE} , is above a threshold value then the increase in collector current implies a reduction in V_{CE} , which reduces until V_{CE} is LOW. The collector-emitter voltage is the reduction from the power supply voltage by the voltage drop across the load resistor.

$$V_{CE} = V_{CC} - i_C R_L$$

(This is the voltage drop discussed when considering fanout earlier in Section 4.1.2.) An increase in the collector current i_C implies a voltage drop across the load resistor R_L , causing V_{CE} to be lower than the power supply V_{CC} . There are three named regions associated with the transistor's behaviour which are marked on the transfer characteristic which shows the change in V_{CE} with V_{BE} . These are:

- (i) the **off** region where the collector current is zero;
- (ii) the **active** or **linear** region where the collector voltage varies rapidly with changes in V_{BE} ; and
- (iii) the **on** region where the collector current is large.

In the *linear* region the slope of the characteristic implies that a small change in V_{BE} will cause a large change in V_{CE} . This is the performance of an amplifier and the transistor is operated in this region in linear circuits (there is a straight line approximation in this region of the characteristic). In logic circuits we will be concerned mainly with two states, the ON and the OFF state. This circuit operates as a logic inverter since when V_{BE} is HIGH ($V_{BE} = '1'$) then V_{CE} is LOW ($V_{CE} = '0'$). Conversely, the output is HIGH, '1', when the input, V_{BE} , is LOW and the transistor is switched OFF.

Circuits based on transistors led to Transistor Transistor Logic (TTL) which dominated logic circuit design at one time with the 74 series (as described earlier in Section 2.4). The

technology had reasonable performance factors, except one. And that one was power consumption. So when devices arrived that had the same speed performance and with much lower power consumption, they became the natural choice for a designer. These devices are based on a different type of transistor.

4.2.3 Field effect transistors

There are also *Field Effect Transistors*, FETs, which have one type of charge carrier controlled by the doping process, either electrons or holes, but not both (as in a bipolar transistor). The technology used to construct these transistors is *Metal Oxide Silicon*, MOS, and the transistors are often known as MOSFETs. FETs again have three terminals but the names on the connections differ from bipolar transistor technology: here we have the drain (replacing the collector), the source (replacing the emitter) and the gate (replacing the base). The aim of the device is to control the current flowing between the drain and the source. This is achieved by changing the voltage applied to the gate. The gate is insulated from the p-type substrate, as shown in Fig 4.12, and so acts as a capacitor affecting the charge. When no voltage is applied, the device is switched off and no conduction occurs between the two regions of n-type material which are the drain and the source. When voltage is applied between the gate and the source, as shown in Fig 4.12(b), electrons are attracted into a channel which forms between the drain and the source (the gate/ substrate acts as a capacitor and the substrate is the lower plate of the capacitor). This increases the channel conductivity, so current can flow from drain to source, and the device is switched on.

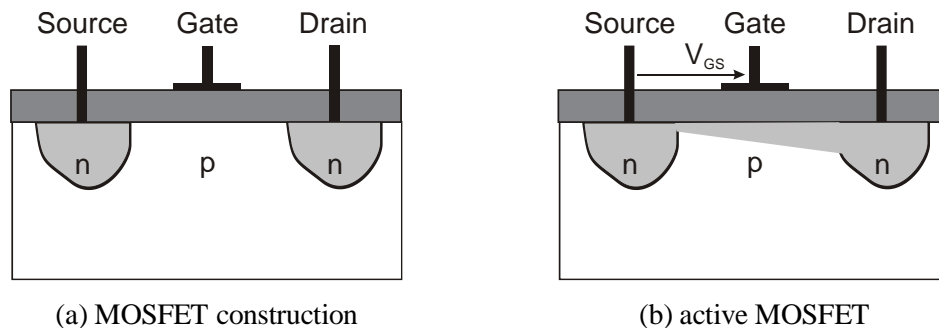


Figure 4.12 Basic MOSFET

This is a simplified picture of a more complicated position. This is an enhancement mode transistor and suffices here. Again, there are three types of activity:

- (i) the **cut-off** region where the transistor is off and the drain current is zero;
- (ii) the **linear** or **ohmic** region where the drain voltage varies rapidly with changes in V_{GS} ; and
- (iii) the **saturation** mode where the channel and the drain current are large.

There are two types of transistor consistent with the two types of carrier. One type uses (a channel made from) holes and is positive MOS, PMOS, the other type uses electrons and is negative MOS, NMOS. Essentially, the transistor in Figure 4.12 is an NMOS transistor with its n channel; conversely, if the substrate is n-type and the two connections are p-type then the

channel is p-type (and V_{GS} is negative) giving a PMOS transistor. The symbols are shown in Figure 4.13.



Figure 4.13 MOSFETs

There are actually many different types of FETs. We are using FETs here within logic technology and it is easiest to think of FETs as **voltage controlled switches (resistors)**. For both FET transistors the *drain source voltage*, V_{DS} , is controlled by the *gate source voltage*, V_{GS} . An NMOS transistor switches ON (the *drain current*, i_D , is positive) for positive V_{GS} . If V_{GS} is less than the threshold then the transistor is OFF (i_D is zero). Conversely, a PMOS transistor is ON when V_{GS} is less than a threshold value (and i_D then increases), when V_{GS} is greater than the threshold the transistor is OFF and i_D is zero. This is why an inverter symbol is included within the PMOS transistor symbol.

A bipolar transistor can be thought of as a current-controlled current source (since i_C can depend mainly on the base current i_B when V_{CE} is sufficiently high) whereas a FET is a voltage controlled current source (since i_D can depend on V_{GS} when V_{DS} is sufficiently large).

A similar circuit can be developed as with the bipolar transistor to give an inverter; here the inverter is based on an NMOS transistor

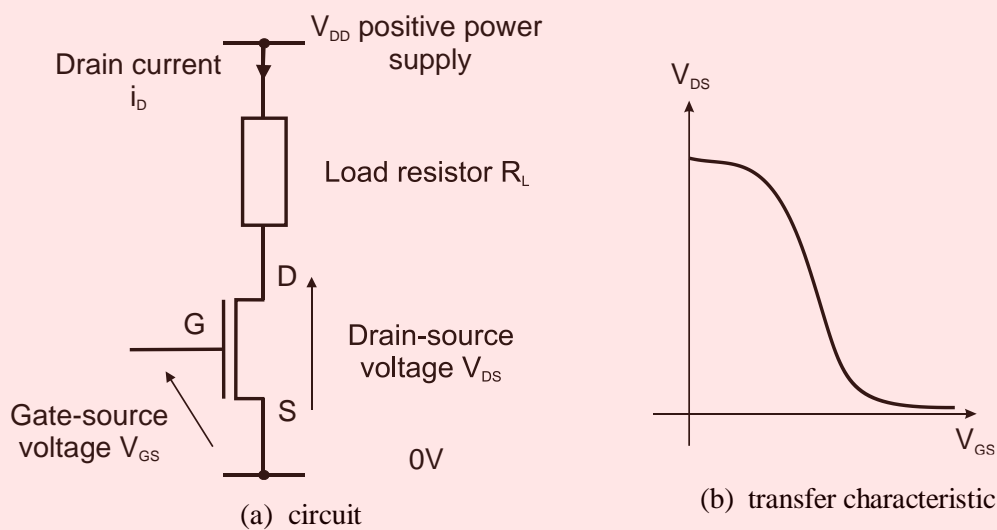


Figure 4.14 Basis of NMOS Inverter

The power supply is called V_{DD} in MOS technology. Again the circuit is that of an inverter since if an input voltage is applied between the gate and source then a HIGH input switches the transistor ON and the output is LOW, whereas a LOW input (one lower than the threshold to switch on the NMOS FET) switches the transistor OFF and the output is HIGH, connected to V_{DD} . MOSFETs have major advantages as a logic technology. When switched ON the impedance is low (of the order of 100 Ohms), whereas when switched OFF the impedance is exceedingly large (of the order of gigaOhms). The gate current is almost non-existent, except during switching, because the gate is insulated from the source/ drain. This leads to a high fan-out capability. As MOSFETs are currently the closest possible approximation to an ideal electronic switch.

Since resistors need a lot of space in VLSI circuits (they need plenty of “real estate”, but that sounds a bit too close to domestic housing), resistors can be made using transistors. We then have the circuit using a load resistor replaced with an NMOS transistor, shown in Figure 4.15. The NMOS load transistor is permanently switched on by connecting its gate to the positive power supply. When both transistors are ON then the circuit is equivalent to a potential divider. If both transistors were the same, the output voltage for a ‘0’ would be $V_{DD}/2$. To force V_{OL} to be away from the forbidden zone, we use a process called **ratioing** which changes the transistors’ relative size and thus their resistance, thus ensuring $V_{OL} \ll V_{OH}$.

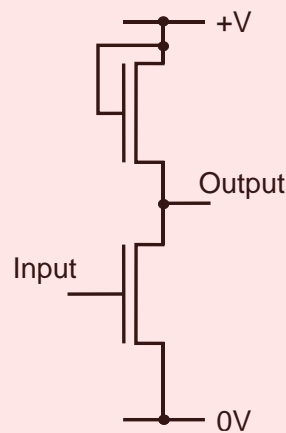


Figure 4.15 NMOS Inverter

This gives the circuit for an *NMOS inverter*. This inverter has two states, one where the input is ‘0’ and the lower NMOS transistor is OFF. When it is OFF, it is high impedance drawing no current, and so disappears from the circuit, shown in Figure 4.16(a). When the input is ‘1’ the lower NMOS transistor is ON and since its impedance is much lower than the load transistor then effectively only the lower transistor remains in the circuit. The output is then connected to ground through the switched ON lower NMOS transistor, shown in Figure 4.16(b).

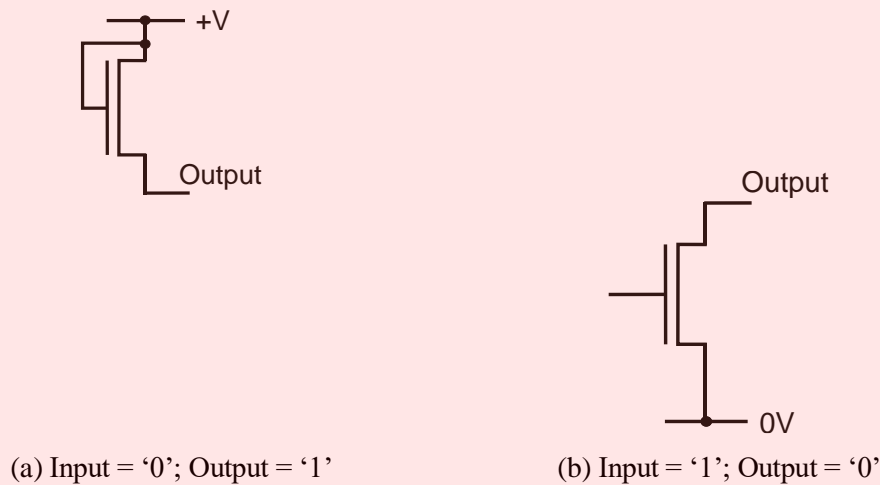


Figure 4.16 Two States of an NMOS Inverter

The circuit for a *PMOS inverter* which uses PMOS transistors is shown in Figure 4.17. The circuit using an inverter is the basis in Figure 4.17(a) and the one using just transistors where the resistor is replaced with a PMOS transistor which is permanently switched on, is shown in Figure 4.17(b).

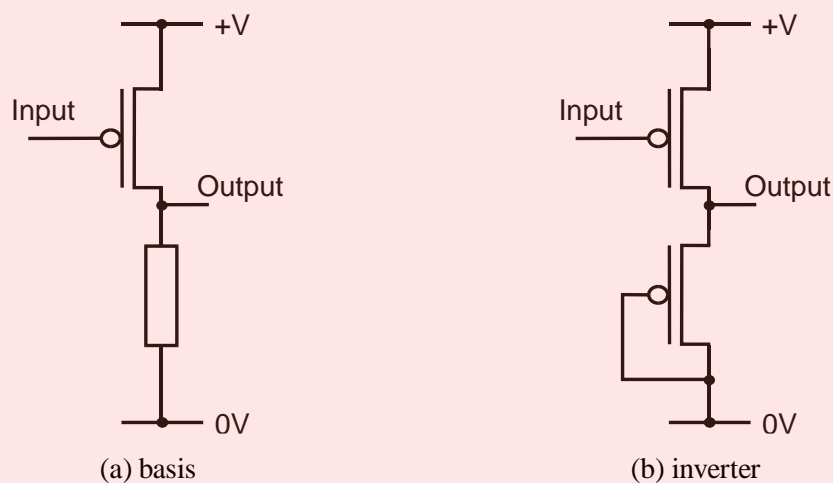


Figure 4.17 PMOS Inverter

Positive MOS depends on holes as the majority carrier which are **slower** due to the decreased mobility of holes when compared with that of electrons. This leads to PMOS being larger than NMOS. Since smaller size implies smaller capacitances and hence faster circuits, NMOS dominated early MOS technology.

The high resistance of a MOSFET input can introduce susceptibility to *static damage*. We can generate a potential, via static electricity, in excess of 5 kV just by walking across a carpet. When we touch a chip input, the resistance of the air gap between the chip input and our finger is about 1 GΩ. If the chip input circuit has small impedance compared with 1 GΩ, then the potential will be dropped across the air gap and we see a spark. If the impedance of

the chip input is large compared with the air gap, the, say, $9\text{ G}\Omega$ impedance of a MOSFET is clearly large compared with $1\text{ G}\Omega$, then the potential will be dropped across the chip input circuit. For a MOSFET this is actually larger than an internal *breakdown* voltage which specifies a maximum voltage that can be applied across parts of the device. If we exceed it we will damage the device. This is why CMOS devices are known as *static sensitive* and, even though modern designs have protection built into circuits (diodes on the inputs), it is sometimes prudent to connect yourself to 0 V (or any old gas or water pipe will do, anything that goes to ground) to avoid static damage.

4.3 Complementary metal oxide silicon (CMOS)

CMOS evolved from combining NMOS and PMOS transistors. The difficulties associated with ratioing motivated development which centred on using both NMOS and PMOS FETs. Since these switch ON for inverted inputs, their combination is called complementary MOS.

A CMOS inverter has one transistor which switches the output HIGH for a LOW input. This requires a PMOS transistor connecting the output to the positive power supply. A LOW output is caused by a HIGH input which requires a NMOS transistor connecting the output to the negative power supply.

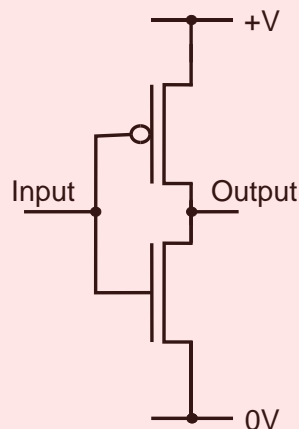


Figure 4.18 CMOS Inverter

The equivalent circuit for a CMOS inverter is a pair of switches on which the input acts in a complementary way. The CMOS inverter is at present the closest possible approximation to the ideal inverter as shown earlier in Figure 4.1.

By comparing the CMOS inverter with the ideal inverter in Figure 4.1, the FETs just replace the switches. For logic gates (such as AND and OR), CMOS circuits actually combine the output stage within the switching function as in Figure 4.19, rather than use a combinational logic function shown in Fig 4.2.

The operation of the CMOS inverter can be described by a table showing which transistors are

OFF or ON in each logic state.

Input	NMOS	PMOS	Output
'0'	OFF	ON	'1'
'1'	ON	OFF	'0'

A 2-input CMOS NAND gate is designed to give a LOW output when both inputs are HIGH. This can be achieved by two NMOS transistors in series, N1 and N2, connecting the output LOW. Conversely, if either or both inputs are LOW then the output should be connected HIGH. This can be achieved by a pair of PMOS transistors, P1 and P2, which are connected in parallel to the positive voltage supply. The combination of these circuits gives the “push-pull” nature of the output circuit: the active device connections to the power rails give rise to faster switching.

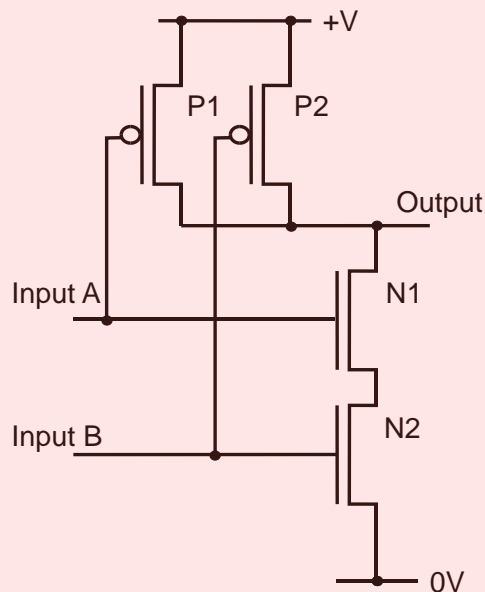


Figure 4.19 A 2-input CMOS NAND gate

The activity of the device can be described in terms of which transistors are ON or OFF according to the values of the inputs. When OFF, the transistors are open-circuit, when on, they conduct, leading to the output to be connected either to + V ('1') or to 0 V ('0'), as in Figure 4.20.

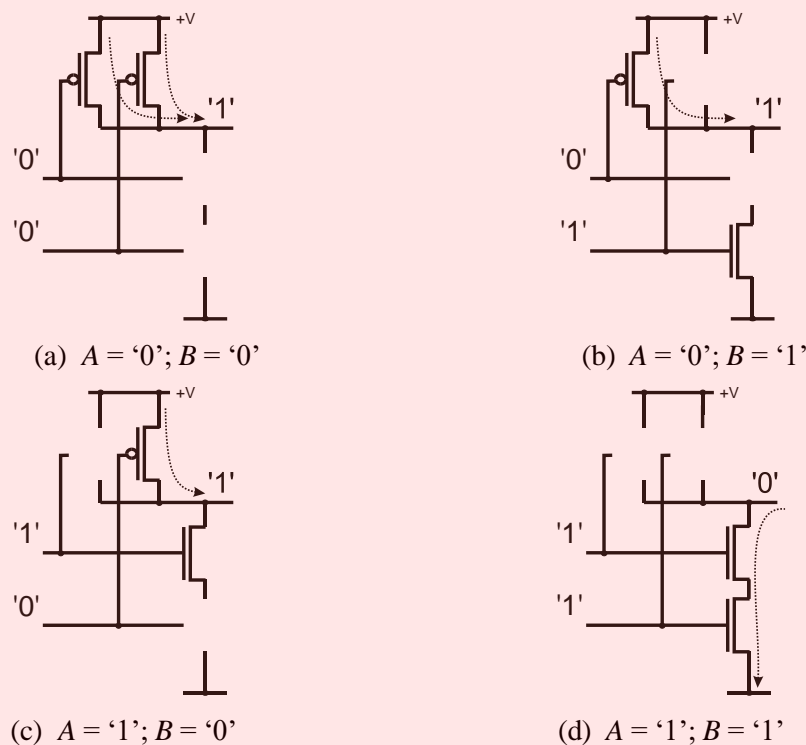


Figure 4.20 A 2-input CMOS NAND Gate Operation

It can be seen that the parallel PMOS transistors serve to connect the output HIGH if either input is LOW, and the nature of current flow is shown by the dotted lines. The parallel PMOS transistors are actually an OR structure; if Input 1 OR Input 2 is LOW then the output is HIGH. This is an illustration of de Morgan's law.

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

The NMOS transistors serve to pull the output low when they are both turned ON. This is an AND structure. If input 1 AND input 2 are HIGH then the output is LOW. The equivalent circuits for a NAND gate sourcing HIGH and LOW outputs (shown in Figure 4.20) concern only the transistors which are switched ON. The large impedance of a MOSFET which is OFF implies that it effectively disappears from the circuit. An alternative description of the circuit is a functional table

Input A	Input B	P1	P2	N1	N2	Output
'0'	'0'	ON	ON	OFF	OFF	'1'
'0'	'1'	ON	OFF	OFF	ON	'1'
'1'	'0'	OFF	ON	ON	OFF	'1'
'1'	'1'	OFF	OFF	ON	ON	'0'

The CMOS 2-input NAND gate has two circuits, one of which provides a NAND gate for LOW inputs (the PMOS transistors), the other is a NAND gate for HIGH inputs (the NMOS

transistors). This is reflected in other CMOS gates. The 2-input CMOS NOR gate provides an output which is LOW when either input is HIGH and an output which is HIGH when both inputs are LOW. The parallel OR structure is used to give a LOW output. The series AND structure is used to provide a HIGH output when both inputs are LOW.

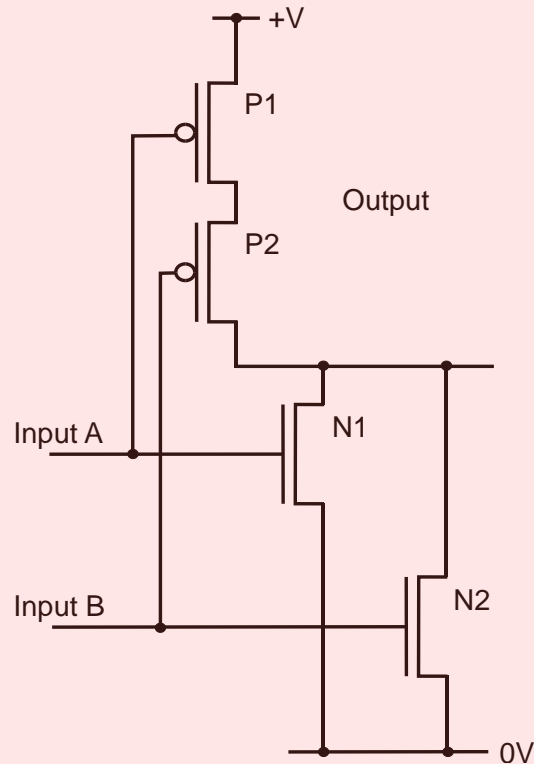


Figure 4.21 A 2-input CMOS NOR gate

The activity of the NOR gate can again be described in terms of which transistors are ON or OFF according to the values of the inputs.

Input A	Input B	P1	P2	N1	N2	Output
'0'	'0'	ON	ON	OFF	OFF	'1'
'0'	'1'	ON	OFF	OFF	ON	'0'
'1'	'0'	OFF	ON	ON	OFF	'0'
'1'	'1'	OFF	OFF	ON	ON	'0'

The main advantage of MOS technology is that it has very low power consumption. This is when the device is not switching and is termed the *static power dissipation*. That the static power is low is intimately related to the closeness of FETs to the ideal gate. The large OFF resistance implies very low power consumption. Disadvantages of MOS technology include a poor drive capability for large capacitive loads (the current sourced by MOSFETs can be less than that for bipolar transistors). Also, the power consumption increases with switching

speed. The power consumption is actually related to

$$\text{power consumption} \propto f \times V_{\text{supply}}^2 \times C$$

where f is the switching frequency, V_{supply} is the positive voltage supply and C is the capacitance driven. It can be observed that the power supply plays a major role in CMOS performance characteristics. Both the fan-out of CMOS circuits and the noise margins are good.

One advantage of CMOS is that the power supply can range between 1.5 V and up to 15 V and this can be used by designers to optimise performance. Note that low power supply logic will continue to be very important in the future.

The major advantages of CMOS are that current versions can achieve the moderate speeds required for general purpose logic with lower power consumption than other logic technologies. For these reasons CMOS now dominates much of integrated circuit design. The early CMOS *logic family* which provided a range of logic functions was the 4000 series CMOS introduced in 1972. This offered very low power consumption but with very low speed operation (about 80 ns propagation delay per gate). This family offered ranges of gates from simple combinational gates to more complex circuits. Though slow, the 4000 series dominated early implementations where power consumption was critical. Developments in CMOS manufacturing technology have led to the present position where CMOS now dominates implementation; this was achieved by supplanting a famous and popular logic technology which used bipolar transistors, TTL logic.

4.4 Concluding comments and further reading

This chapter has described what is inside integrated circuits from the viewpoint of CMOS which is now the major logic technology. As a designer it is often necessary to be aware of the properties of the circuits you are using to implement designs. You will be designing circuits to meet a wide-ranging specification. Some aspects of the specification are often more critical than others and these can dominate the logic technology chosen for implementation. Designers use less prefabricated-function circuits now (those available from logic families), but you might resort to them from time to time. Though most designers use programmable logic implementations, these naturally depend on a logic technology for implementation. There are other logic technologies; some are in a development stage, some are now obsolete, while others have not penetrated the market much. Diode logic is archaic and Transistor-Transistor logic (TTL) is very dated too (though not according to some textbooks). Amongst those which never developed is BiCMOS which aimed to take combine the advantages of speed of bipolar logic with the low power consumption of CMOS. Failure to penetrate the market might imply that the technology did not live up to its early expectations, but there are also commercial considerations which are beyond the scope of this text.

For an introduction to semiconductor physics, try Parker, G. J., *Introductory Semiconductor Device Physics*, [Parker04] or Millman, J. and Grabel A., *Microelectronics*,

[Millman88] which gives a lucid account of microelectronic technology. There are a number of reasonably priced texts which include digital electronic circuits within a complete study of electronics, see for example Sedra A. and Smith, K., *Microelectronic Circuits*, [Sedra10]. There are books dedicated to particular technologies, see for example, Elmasry, M. I., *Digital Bipolar Integrated Circuits*, [Elmasry83], Weste, N. H. E., and Harris, D., *CMOS VLSI Design: A Circuits and Systems Perspective*, [Weste10], and Alvarez, A. R., *BICMOS Technology and Applications*, [Alvarez89]. There again, the CMOS ones are of most interest now, and [Weste10] is well known for its excellent coverage. At the time of writing, the world's fastest transistor came from the University of Southampton [Kham06] (110 GHz!) though there is naturally considerable interest, and continuing advance, in this topic.

Manufacturers provide databooks on the circuits that they produce, as well as handbooks on how to use them. There are many manufacturers and many vendors (e.g. RS Components and Farnell) whose websites provide datasheets on the products that they supply. We're finished now with the main building blocks for introductory logic design. The real meat (or a tasty quiche) is logic which incorporates memory: for that we need combinational design and the circuits which implement them. Onwards to sequential logic we go.

4.5 Chapter 4 Questions (Solutions in Appendix 1)

Q4.1 Develop a CMOS circuit to implement a 2-input (positive logic) AND function.

Q4.2 Deploy a CMOS NAND gate as an inverter. Identify which transistors are redundant, and why.

Q4.3 An NMOS NAND gate is made from two transistors one with ON resistance 200 ohms and the other has an ON resistance which can be controlled. The OFF resistance for both transistors is in the order of $G\Omega$. What ratio of ON resistance must be achieved so that a LOW output approaches $V_{DD}/3$?

Q4.4 Design a CMOS (positive logic) circuit to implement the function

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

Q4.5 For a bit of history (play Leader Of The Pack by the Shangri Las when you do this), make a 2-input OR gate from two diodes. Now make an AND gate.

Q4.6 For a bit more history, changing the music to Stairway to Heaven by Led Zeppelin, develop the transistor-based inverter into a NAND gate. (Hint: there can be more than one emitter on a silicon transistor, since this just requires an extra connection to be made to the semiconductor substrate.)

Q4.7 Play the Fields of Athenry by the Dropkick Murphys to return you to the third millennium.