Activity 1.3

Memory and storage

Suggest additional reasons for using different types of memory and storage media within a computer.

1.4 Forms of computer hardware

Before the late 1930s, computing machines had taken the form of mechanical and electromechanical systems. This was soon to change and the quest for the 'all-electronic' computer became the subject of intensive research (the first machine of this sort was the ENIAC – the Electronic Numerical Integrator and Calculator – which was developed during WWII and was primarily used for solving ballistics problems. The machine was some 33m long, 2.8m high and a metre or so wide – and employed 18,000 vacuum tubes.

Two of the most important breakthroughs that have occurred, and which made possible the modern computer, were the invention of both the *transistor* and the *integrated circuit*. These technologies enabled the development of more powerful and more reliable computer systems. A major driving force behind the development of the integrated circuit was the Apollo space programme that led to the voyages to the Moon in the late 1960s and early 1970s. Rockets making such journeys had to be equipped with computer technologies that could perform a multitude of tasks, including navigation (although it is amazing how rudimentary the computer systems used actually were). In parallel, from the late 1950s through the 1960s, the nuclear missile programme – which centred upon the development of intercontinental ballistic missiles – necessitated the use of computers for the detection of missiles in flight and for missile management and navigation.

The 1960s was the period in which large computers evolved. These computers were known as **mainframes** and such machines became bigger and bigger. Computers of this type take the form of large installations, and can support many users who share the computational capacity of the machine. It was at this time that the philosophy of 'economy of scale' came into being. Put simply: if you spend twice as much on a new mainframe, you get more than four times the processing power. Given this idea, it was natural for mainframes to become ever bigger. Unfortunately, bigger also meant more complicated and it became increasingly difficult to equitably share the processing power between users. The result was that operating systems became ever-more complex and used up more of the processing power of the machine – leaving less available to the users.

Mainframes developed during the 1960s and 1970s took the form of large computers located within large, clinical, air-conditioned facilities. Unfortunately, the very size of such machines ultimately impacted on their performance. Here, it is important to note that electrical signals do not travel instantaneously along wires – they propagate at a finite speed. Consequently, as connections become longer, it takes more time for signals to pass along the wires and this reduces computer performance. In order to increase performance, we endeavour to minimise the lengths of the interconnections between critical components – which leads to the design of compact computers. As mainframes grew in size, electrical signals had further to travel (propagate) and so performance was compromised.

Minicomputers are much smaller, and were intended to support the computing requirements of smaller organisations, rather than the large institutions serviced by mainframe machines. As with the mainframe, the processing power of the minicomputer is shared between a number of users. Within a business, typical tasks that would have been carried out in the 1970s by minicomputers included staff payroll calculation and stock inventories. As with any centralised machine, reliability is paramount. Should the minicomputer develop a technical problem, then all users are stopped in their work. Furthermore, when a computer is shared between users, one can never guarantee performance – this being determined by the number of programs running on the machine at any one time.

In the early 1970s, Xerox undertook pioneering work in an effort to develop the **personal computer**. Essential to the vision was the development of a desktop machine where each user has access to their own computer, and its resources are not shared. Consequently, performance is controlled by the user and not by others using the machine. Xerox also pioneered new interaction techniques, particularly the incorporation of the mouse (invented in the mid-1960s), and the graphical user interface that we use today. They also pioneered the development of the **WYSIWYG** wordprocessor ('What You See Is What You Get' – i.e. the document that appears on the computer screen is identical to the form of the document when printed).

The development work at Xerox took some time, and it was not until the early 1980s that the personal computer (PC) that we know today became readily available. Soon a variety of personal computers were produced by vendors such as Xerox, IBM, Commodore and Macintosh. Other companies appeared on the scene, and a wide range of products became available.

The 1980s denoted a period of rapid advancement in computer development and performance. It was also a period of the survival of the fittest, and intense corporate warfare broke out within the computer industry; excellent products seemed to disappear almost overnight. Occasionally, companies were taken over simply in order to suppress products. This was also a period when standards began to evolve and the issue of backward compatibility became paramount. In this context, support for backward compatibility means that when a new computer architecture is developed, it must still be able to execute older software products. The availability of DOS within current versions of Windows provides an example of adherence to backward compatibility.

Since the mid-1980s there has been a reduction in the diversity of computer architectures and operating systems. The main advances that have taken place are in the areas of computer performance, the increased capacity of storage media such as hard disks, and networking technologies (particularly the Internet). In later chapters we will be considering these issues in some depth.

1.5 Basic computer operation

Computers are digital machines that employ binary (number base 2) arithmetic. It is important to remember that whatever task you perform on a computer (including, for example, playing a CD or video, undertaking a word-processing activity or accessing Internet sites), the computer is operating on numbers: it is a mathematical machine.

The diversity and nature of the applications that can be supported by the modern computer often makes it difficult to imagine that the operation of the machine is underpinned by mathematics. For example, consider a word-processing activity. Within a document we may insert text (comprising alphabetic characters), diagrams and photographs. However, within the computer each of these forms of material is represented by sets of numbers. Each word-processing operation that we carry out results in the computer performing mathematical calculations on these numbers – in fact, the computer knows only about numbers.

Even when we create words and sentences in a document, these are not stored and manipulated by the computer as a series of characters, but as a set of numerical values – the operation of the computer is indeed based on mathematics.

A computer is said to be a 'digital' machine because all the components, wires, etc, operate only on discrete signals. This is rather like a conventional light switch. The light can only be turned on or off – it cannot be a half on, a quarter on, or a quarter off; the bulb has one of two states: illuminated or otherwise. This analogy with a light bulb serves us quite well, because not only is a computer a digital machine, but all computation is performed not in our conventional base 10 system (in which we represent numbers by means of ten symbols, 0 to 9), but in base 2 (for which we have only two symbols '0' and '1'). In the next section we briefly consider the base two number system.

1.6 Binary (number base 2)

In our everyday lives, we employ the decimal number system (base 10). Here, we make use of ten symbols ('0' through to '9'). The modern computer, however, operates in base 2 (binary), and in this number system we have only two symbols ('0' and '1'). In this section we provide a brief review of the binary system. It is important that you have a clear understanding of this number base, because it is critical to the operation of modern computer technologies.

Let us begin by considering how we represent numbers in base 10. As indicated above, we have at our disposal ten symbols (0,1,2,3,4,5,6,7,8,9). Consequently, if we want to represent a number (integer) up to the number nine, we simply make use of one of these symbols. But what happens if we wish to represent a number greater than nine? Suppose that we want to represent the number thirteen. We now allow our symbols to take on different meanings according to their position (placement). For example, when we write 13, we actually assign to the digit '1' a new meaning. Here it represents the number of tens. Thus, 13 really means one 10, and three units (1s).

Consider another example. Suppose we want to represent numerically three thousand four hundred and sixty-seven – we do so as follows:

Thousands	Hundreds	Tens	Units
3	4	6	7

As you can see in the above, the digit '3' indicates the number of thousands, the digit '4' the number of hundreds, the digit '6' the number of tens, and the digit '7' the number of units.

Each digit in the number has a different significance depending on the column in which it is located. Now look at the values ascribed to the columns. We have the number of ones (units), the number of tens, the number of hundreds, and the number of thousands. The next column would be the number of ten thousands, etc. From this pattern it should be clear that the columns can actually be represented in terms of 'powers' of ten:

Units = 10^0 (remember that 10^0 is defined as equalling 1)

Tens = 10^1

Hundreds = 10^2

Thousands = 10^3

Because we are working in base 10, the values ascribed to each column in our numeric representation increase by a factor of 10 as we move along the number.

In base 2 (binary), the values of the columns increase by a factor of 2. Let us take an example: but in order to understand this you need to remember that, just as in base 10 we have ten symbols at our disposal, in base 2 we only have two symbols (represented as 0 and 1). Suppose we write the binary number 1100. What does this represent in our base 10 number system? Examine the following table:

Eights (2^3)	Fours (2^2)	Twos (2^1)	Units (2^0)
1	1	0	0

The binary number 1100 represents one 8, one 4, zero 2s, and zero units. Adding 8 and 4 together gives us 12. In short, the binary number 1100 corresponds to the base 10 number of twelve.

Consider a further example – the binary number 1111.

Eights (2^3)	Fours (2^2)	Twos (2^1)	Units (2^0)
1	1	1	1

As you can see, this corresponds to one 8, one 4, one 2, and one unit (1). Adding these together gives us 15. Thus the binary number 1111 corresponds to the base 10 number of fifteen. You can now convert binary numbers to base 10.

Activity 1.4

Converting from binary to base 10

Convert the following binary numbers to base 10.

(a) 1101 (b) 11111 (c) 1010101

The next step is to convert base 10 numbers into binary. As long as you remember the values assigned to the columns in both base 10 and base 2, you should encounter no problems with this.

Let us begin with a simple example. Consider the base 10 number 'five'. In order to convert this into binary, we write down the values of a few binary columns:

8s	4s	2s	1s

Now we remember that a binary number can only be expressed using 0s and 1s. Therefore, all we have to do is to insert 0s and 1s in the appropriate columns:

8s	4s	2s	1s
	1	0	1

Thus the binary representation of the base 10 number five is 101. When expressing a number in binary we are simply looking at what 'column values' we need to use so as to form the base 10 number (five can be expressed as a 'four' plus a 'one'). Consider another example – the base 10 number twenty-seven. As this is a larger number, extra columns are needed:

64s	32s	16s	8s	4s	2s	1s

Here we have provided more columns than are necessary: naturally the 32 and 64 columns cannot assist us in representing twenty-seven as placing a '1' in either of these columns would give rise to a binary number larger than 27! However, if we place a '1' in the 16 column, a '1' in the 8 column (this would add to 24), then we have 3 left to represent. This can be achieved by placing a '1' in the 2 column and a '1' in the 1 column – as follows:

16s	8s	4s	2s	1s
1	1	0	1	1

Thus the binary representation of the decimal number twenty-seven is 11011.

Activity 1.5

Converting from base 10 to binary

Convert the following base 10 numbers to binary:

(a) 9

(b) 36

(c) 73

Activity 1.6

Binary arithmetic

Most scientific calculators support binary arithmetic, and a 'mode' key is usually provided to enable the calculator to operate in different number bases. This facility is also available in the calculator program supplied with Windows. The calculator can be accessed by selecting 'programs', 'accessories', and then 'calculator' under the 'start' menu icon (at the bottom left-hand side of the screen). Once you have accessed the calculator, you need to select 'scientific' mode (this is available under the 'view' icon). To enable the calculator to operate in base 2, check the 'bin' (binary) option. You will now find that only the '0' and '1' numbers on the keypad operate (as we have seen, in binary we only have these two numerical symbols). Use this calculator to perform the following binary arithmetic. You should check the results that you obtain by manually converting each of the binary numbers to base 10, performing the arithmetic in base 10 and converting the result back to base 2.

(a) 11 + 101 =

(b) 1110 -11 =

(c) 100101 * 11 =

Note that * represents the multiplication operation.

1.7 Bits and bytes

Each binary digit ('0' and '1') is referred to as a 'bit', and a group of eight bits is called a 'byte'. Thus 16 bits constitutes two bytes, 64 bits are eight bytes, etc. It is important that we have a clear understanding of the number of values that can be represented by a certain number of bits. This can be readily determined by examining some simple cases. For example, consider two bits – they may take on the following values: 00, 01,10 and 11 Here we are simply counting – in binary and the binary numbers given here represent 0, 1, 2 and 3 in base 10. Thus two bits can represent four different values. Consider now the case of three bits. These may take on the following binary values: 000, 001, 010, 011, 100, 101, 110 and 111 Again we are simply counting in base 2 and you should confirm that these numbers represent 0, 1, 2, 3, 4, 5, 6 and 7. Thus three binary digits can represent eight different values.

Activity 1.7

Counting in binary

Consider the case of four binary digits. List all possible values that may be represented by four bits, and state the number of values that they may be used to represent. As we have seen, two bits can be used to represent four different values (=2^2); three bits, eight different values (=2^3); and four bits, 16 values (=2^4). From this pattern it is readily apparent that an arbitrary number of n bits can be used to represent 2^n different values.

Activity 1.8

Binary representation

Determine the number of binary values that may be represented by:

(a) 16 bits (b) 32 bits (c) 3 bytes

1.8 Summary

In this first introductory chapter we have introduced a number of key ideas that will be used as the basis for discussion in the coming chapters. As we have seen, although the computer is able to process different forms of media (such as graphics, video, audio and text), within the computer itself these are all represented by sets of numbers and processed by means of mathematical calculations. Additionally, although in our everyday lives we generally employ the decimal number base, computers operate in base 2 (binary). This is not an essential characteristic of the computing machine but greatly facilitates aspects of their implementation. Since operations are performed by the computer in base 2, a clear knowledge of base 2 arithmetic is an essential precursor to gaining a sound understanding the internal operation of computer-based technologies.

Despite the flexibility and power of modern computer systems, computer hardware is based on the interconnection of large numbers of very simple circuits ('logic gates'). In the next chapter we introduce a number of these 'gates' and show how they can be used to construct more complicated devices.

1.9 Review questions

Question 1.1 State four essential characteristics of a computer.

Question 1.2 A computer is said to be a 'digital machine'. What do you understand by the term 'digital' when used in this context?

Question 1.3 Convert the base 10 number 21 to binary, and the binary number 1001 to base 10.

Question 1.4 State two key features of the 'stored program' computer architecture.

Question 1.5 How many different values may be represented by (a) 4 bits and (b) by 2 bytes?

Question 1.6 What does the abbreviation 'CPU' stand for?

Question 1.7 Multiply the binary numbers 101 and 110.

Question 1.8 What do you understand by the phrase 'a general purpose programmable machine'?

1.10 Feedback on activities

Activity 1.2: The essence of a computer

A conventional calculator is able to implement a number of predefined functions (we exclude from our discussion programmable calculators). On the other hand, a computer is a far more powerful tool as it is able to execute a series of instructions by sequence (one after the other), by selection (decision-making during the execution of the instructions) and by iteration (repeatedly executing a series of instructions until some form of condition occurs).

Consequently, the order of execution of instructions is not necessarily predefined, but is generally determined by the results obtained during the computational process.

Activity 1.3: Memory and storage

One main reason for providing different forms of memory/storage device within a computer concerns the balance we seek to obtain between speed, storage capacity, and price. Higher-speed memory devices are more expensive in terms of their cost-per-unit of storage capacity. Therefore storage devices able to store larger volumes of data tend to be less expensive, but operate at a lower speed. Main memory (RAM) is more expensive (in terms of cost-per-unit of storage) than, for example, the hard disk drive. On the other hand, main memory operates at a higher speed.

Activity 1.4: Converting from binary to base 10

(a) The binary number 1101 corresponds to 13 in base 10, i.e. one unit, zero 2s, one 4, and one 8.

(b) The binary number 11111 corresponds to 31 in base 10, i.e. one unit, one 2, one 4, one 8 and one 16. Adding these up gives us 31.

(c) The binary number 1010101 corresponds to 85 in base 10.

Activity 1.5: Converting from base 10 to binary

(a) 1001. Meaning that 9 can be represented by writing a '1' in the 8 column, and a '1' in the 1 column.

(b) 100100. Meaning that 36 can be represented by placing a '1' in the 32 column, and a '1' in the 4' column.

(c) 1001001.

Activity 1.6: Binary arithmetic

(a) 11+101= 3+5=8. This corresponds to the binary number 1000

(b) 1110-11= 14-3=11. This corresponds to the binary number 1011

(c) 100101*11=37*3= 111. This corresponds to the binary number 1101111

2.1 Introduction

Computers are constructed through the interconnection of very large numbers of simple circuits (these are referred to as logic gates). In this chapter, we introduce several types of logic gate, focusing our attention on the most basic and most widely used devices. In Section 2.3 we briefly introduce two general types of digital circuit – these being referred to as combinational and sequential logic circuits. In the case of the former, a circuit's output is entirely determined by the combination of signals currently supplied to it. In contrast, sequential logic circuits produce output which is determined not only by the current logical state of the input signals, but also by the previous state of the circuit. In short, sequential logic circuits exhibit 'memory' and underpin the construction of, for example, a computer's main memory.

In Sections 2.4 and 2.5 we provide exemplar circuits that demonstrate the implementation of both combinational and sequential circuits and therefore provide an insight into the design of digital electronics.

Finally, in Section 2.6 we discuss the hexadecimal number base (base 16). As we discussed in the previous chapter, modern computers perform their computations on binary (base 2) numbers. Although binary may be the language of machines, from our point of view it is certainly a far from convenient method of expressing and writing down large numbers. However, binary numbers can be easily converted into base 16 and this provides a compact and convenient means by which we can express and represent strings of 1s and 0s.

2.2 Logic gates and truth tables

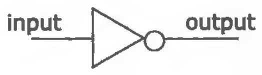
When we think of a computer, we often imagine it to be constructed from immensely complex circuits. In fact, a computer is essentially constructed through the use of many instances of simple building blocks. These building blocks are referred to as logic gates, or simply as 'gates'. A gate consists of a simple electronic circuit which has one or more inputs and one or more outputs. We interconnect these gates on a vast scale, and each is responsible for reacting in some way to the binary input values presented to it.

Perhaps it is surprising that something as complicated as a computer can be created by the interconnection of only a small number of different types of gate. In fact, as we will see later in this chapter, a computer can (in principle) be constructed using only one type of gate. Nevertheless, different types of gate are manufactured, and facilitate both the construction of a computer and its speed of operation. The availability of different types of gate relates therefore to convenience and performance rather than absolute necessity. The functionality of seven key logic gates is briefly outlined below.

- **The inverter – (NOT gate)**
- **The AND gate**
- **The NAND gate**
- **The OR gate**
- **The NOR gate**
- **The exclusive OR gate**
- **The buffer.**

The inverter – (NOT gate)

The inverter (also commonly referred to as the NOT gate) is the simplest of all gates. As the word 'inverter' implies, the function of this gate is to invert the signal presented to it. It has one input and one output. If a voltage corresponding to a binary number 1 (this is also called a 'logic high') is presented to the input of an inverter, then the output will be a binary 0 (this is also referred to as a 'logic low'). Conversely, if a logic low is presented to the input, the output will be a logic high. In this way the inverter simply inverts the binary value presented to it. All gates are represented in circuit diagrams by means of means of different symbols. The symbol for an inverter is shown in Figure 2.1. You will note that there is a small circle at the output of the gate. Whenever you see such a circle, it indicates that the gate has an inverting function. In the small table provided on the right-hand side of Figure 2.1 we illustrate the functionality of the gate. This shows the two possible logic states that can be applied to the input, and the corresponding logic states produced at the output. A table that illustrates the logical function of a gate is referred to as a 'truth table'. Such tables are very convenient and can be used not only to show the logical operation of individual gates, but also to summarise the logical operation of circuits that are constructed from any number of gates.



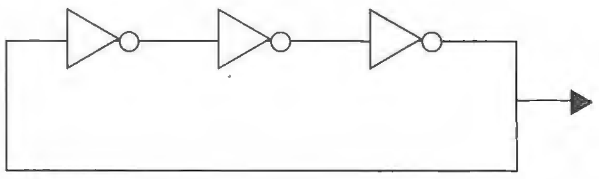
symbol for invertor

input	output
0	1
1	0

truth table

Figure 2.1: The symbol and truth table for an inverter

A gate is constructed on a silicon chip and consists of simple electronic components such as diodes and transistors. However, when the input state applied to the gate changes, it takes the components within the gate a certain time to react to these changes and so produce an output. Thus, if the input applied to a gate is changed, it takes a certain (very small) amount of time for this change to be reflected at the output. This delay in the reaction of a gate to an input change is referred to as the gate's 'propagation delay'. These are very small; typically a small number of nanoseconds (1 nanosecond equals 10^{-9} seconds). It would seem that such delays are of no significance, but we must remember that computers operate at extremely high speeds, and signals within a computer pass through enormous numbers of gates. Therefore even very small delays can be significant. It is not the delay associated with a single gate that ultimately counts, but rather the sum of all the delays associated with many gates that form circuits within the computer. In Figure 2.2 a simple circuit diagram is presented showing three inverters connected together. As may be seen, the output of the third inverter is connected back to the input of the first inverter. Suppose that when we turn the power on to this circuit, wire A happens to be at a logic high (you may equally assume that it happens to be at a logic low). The output from inverter 1 will then be at a logic low, consequently the output from inverter 2 will be at a logic high, and in turn the output from inverter 3 will be at a logic low. If we now work our way through the circuit again, we will see that wire A will again change state. In short, the circuit oscillates – each wire switches between being at a logic high, and a logic low. In principle, this circuit will produce a waveform as indicated in Figure 2.2. (This explanation of the action of the circuit is somewhat simplified. As we will later discuss, signals do not instantaneously propagate through gates – their passage takes a finite time. This complicates the operation of this simple circuit – although if certain requirements are met, the circuit will oscillate and produce the form of waveform indicated in Figure 2.2.).



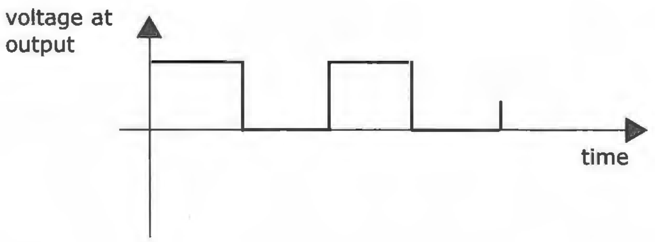


Figure 2.2: A simple oscillator that may, in principle, be formed using three inverters

Activity 2.1

Propagation delay

Consider the circuit presented in Figure 2.2. Suppose that rather than using three inverters, five inverters are connected in a similar way. What effect is this likely to have?

Activity 2.2

The action of an inverter

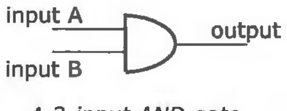
Consider the circuit shown in Figure 2.2. Suppose that it is constructed using an even rather than an odd number of gates. For example, you may consider a circuit comprising two, or four inverters, connected together in the manner indicated in Figure 2.2. How will this circuit behave?

The AND gate

The function of the AND gate is slightly more difficult to understand than the inverter. The AND gate has more than one input (typically between two and eight inputs), and a single output. If any of the inputs are in a logic low state (binary 0), then the output will also be a logic low. This can be easily remembered as:

- **Any low gives a low.**

The symbol for an AND gate is given in Figure 2.3, together with a corresponding truth table. Here, an AND gate with two inputs (labelled A and B) is shown, and as can be seen from the truth table, if either of these inputs is a logic low then the output is also a logic low. Consequently, the output from the gate can only be a logic high if the inputs are all logic highs. As with an inverter, this functionality does not seem particularly complicated and it is difficult to imagine that such a simple circuit can be of such importance in the implementation of computer systems. As with the inverter and all other gates, there is a propagation delay associated with the operation of the AND gate.



A 2-input AND gate

A	B	output
0	0	0
0	1	0
1	0	0
1	1	1

Truth table

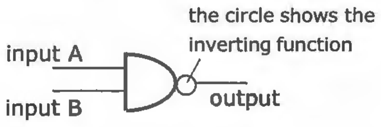
Figure 2.3: A two-input AND gate and its truth table

The NAND gate

In Figure 2.4 the symbol for a NAND gate is provided, together with its associated truth table. Silicon chips containing NAND gates are readily available. However, if one does not have access to a specifically manufactured NAND gate, it can be implemented by combining an AND gate with an inverter, see Figure 2.5. The diagram illustrates the way in which a NAND gate can be implemented using an AND gate connected to an inverter (recall this is also referred to as a NOT gate). In fact, the word NAND is created by bringing together the words 'NOT' and 'AND'. Thus, a NAND gate is a 'NOT AND' gate. As you will see from the truth table provided in Figure 2.4, if any of the inputs of the NAND gate are at a logic low, then the output is a logic high. This can easily be remembered as:

- **Any low gives a high.**

Consider for a moment the implementation of the NAND gate using a NOT and an AND gate, as shown in Figure 2.5. Here, we have simply inverted the output from the AND gate, and a comparison of the truth table provided in this illustration with the one presented in Figure 2.4 shows that both circuits perform the same logical function. However, since electrical signals must now travel through two gates this circuit will exhibit a greater propagation delay than would be the case if a single NAND gate were used.



A 2-input NAND gate

A	B	output
0	0	1
0	1	1
1	0	1
1	1	0

Truth table

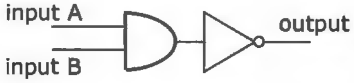
Figure 2.4: A two-input NAND gate and its truth table

As with the AND gate, a NAND gate has two or more inputs (typically between two and eight), and a single output. Referring to the symbol for the NAND gate which is illustrated in Figure 2.4, you will see that this symbol is the same as that used for the AND gate, but it has a small circle at the output. As indicated above when discussing the inverter, this circle indicates that the gate has an inverting function. By connecting together the inputs of a NAND gate, an inverter may be constructed. This is illustrated in Figure 6.6.

Activity 2.3

A three-input NAND gate

Draw up a truth table for a 3-input NAND gate.



A	B	C	output
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Truth table

Figure 2.5: A NAND gate implemented by using an AND gate and inverter

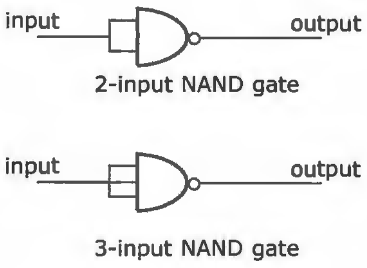
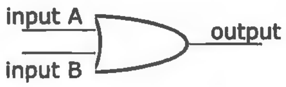


Figure 2.6: When the inputs of a NAND gate are connected together it acts as an inverter

The OR gate

As with the AND and NAND gates, the OR gate has two or more inputs. The symbol for a two-input OR gate is given in Figure 2.7 together with the corresponding truth table.



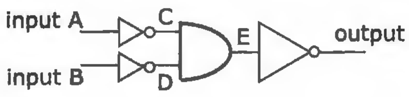
A 2-input OR gate

A	B	output
0	0	0
0	1	1
1	0	1
1	1	1

Truth table

Figure 2.7: The symbol for a two-input OR gate and its truth table

In fact, an OR gate can be implemented using an AND gate and some inverters. Consider the circuit and truth table presented in Figure 2.8. As may be seen, each of the input signals passes through an inverter before being applied to the AND gate, and the output from this is also passed through an inverter. You should verify the truth table presented in this illustration, and with reference to the truth table given in Figure 2.7 confirm that this circuit performs the logical OR operation.



A	B	C	D	E	output
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

Truth table

Figure 2.8: A two-input OR gate implemented using an AND gate and three inverters

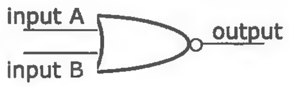
Activity 2.4

Building a two-input OR gate by means of NAND gates

Draw a circuit diagram showing how a two-input OR gate could be implemented using only NAND gates.

The NOR gate

The NOR gate derives its name from the 'NOT OR' designation. This means that it is an OR gate whose output is inverted. The symbol for a two-input NOR gate is provided in Figure 2.9. Here, you will see that this symbol is the same as that used for an OR gate, but is followed by a small circle which indicates its inverting function. A NOR gate has two or more inputs – most commonly between two and eight inputs are provided.



A 2-input NOR gate

A	B	output
0	0	1
0	1	0
1	0	0
1	1	0

Truth table

Figure 2.9: The symbol for a two-input NOR gate and its truth table

Activity 2.5

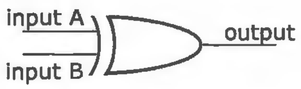
A three-input NOR gate constructed using only NAND gates

Draw a circuit diagram showing how a three-input NOR gate could be implemented using only NAND gates.

The exclusive OR gate

The name 'exclusive OR' is usually abbreviated to 'XOR' or 'EOR'. Again, this has two or more inputs and in the case of the two-input XOR gate, its functionality is the same as the two-input OR gate other than when the two inputs are a logic high. In this case, the output is a logic low. The symbol for a two-input XOR gate and its corresponding truth table are given in Figure 2.10.

As with other gates mentioned above, the XOR function can be implemented using other gates – as illustrated in Figure 2.11. Here, an XOR gate is constructed using a NAND gate, an OR gate and an AND gate. However, since we know that the OR and AND gates may be constructed using NAND gates, it follows that the XOR gate can be constructed using only NAND gates. In fact, most gates can be implemented using only NAND gates – and, in principle, a computer can be constructed from this single and very simple 'building block'.



A 2-input XOR gate

A	B	output
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

Figure 2.10: The symbol for an XOR gate and its truth table

A deeper insight into the functionality of an XOR gate can be obtained from the information provided in Figure 2.12. Here, one input is labelled as 'A' and the other as 'control'. As may be seen from the truth table provided in this illustration when the 'control' input is at a logic low, the output simply follows the input signal applied to 'A'. Thus, in this case if 'A' is a logic low – so too is the output. If we now apply a logic high to 'A', then the output also takes on this same logic state.

However, if the input labelled 'control' is a at logic high, then the gate acts as an inverter – the output state is a logic low when 'A' is a logic high and vice versa.

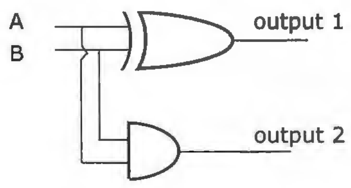


Figure 2.11: An XOR gate may be implemented using an AND gate, a NAND gate and an OR gate

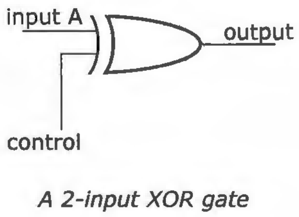
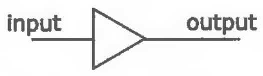


Figure 2.12: The XOR gate may be considered to act as a 'programmable inverter'. When the input labelled 'control' is a logic low, then the output state is the same as input 'A'. However, if the 'control' input is a logic high, then the gate acts as an inverter.

The buffer

The symbol and truth table for this gate are provided in Figure 2.13. As may be seen, the symbol used for the buffer has the same form as that used for the inverter, although the small circle used at the output of the inverter is missing. (Recall that this circle is used to indicate that the gate performs an inverting function.) In fact, as may be seen from the truth table, the buffer does not have any logical function – the input and output have identical logic states (a buffer has a single input and single output). It therefore appears that this device has little purpose – it simply relays to the output the logic state applied to the input.



symbol for buffer

A	B
0	0
1	1

truth table

Figure 2.13: The buffer – here we show the circuit symbol and truth table.

However, within digital circuits the buffer has a very important function which relates to the interconnection of gates.

Gates are electronic devices which have one or more inputs to which we apply logic low and logic high signals. Typically, the former is represented by voltages close to zero volts and the latter by voltages close to 5 volts. (For example, in the case of a particular type (family) of logic gates (called the TTL LS series (Transistor-Transistor Logic Low-power Schottky)) a logic low applied to the input of a gate is represented by voltages in the range 0 – 0.8v, and a logic high by 2.8 – 5v. Another form of logic (CMOS) employs somewhat different voltages to represent logic low and logic high states). When we apply these voltages to the input(s) of a gate, a small electrical current will flow into the gate. This current must come from somewhere – it is in fact provided by the output circuit of the gate to which the input(s) of other gates are connected. Consider a circuit where the output of a NAND gate is connected to the inputs of six inverters. So as to correctly operate, each of these inverters will draw a small current from the NAND gate. However, there is a maximum current that the NAND gate can provide and if this is exceeded, the circuit will fail to operate. This limits the number of gates that can be connected to the output of a single gate and in this context we make use of the term 'fan-out' which refers to the maximum number of gates that can be directly driven by a single gate. For the TTL LS family of gates mentioned earlier, the maximum fan-out is 20.

A buffer is commonly used in situations in which we wish to exceed the maximum fan-out that can be supported by a single gate. By way of an example, suppose that we are using some form of logic gates that have a maximum fan-out of four (this would not be a very attractive form of gate) and that we wish a NAND gate to drive six other gates (inverters). If we simply connect these then we would exceed the fan-out limitation and the circuit would not operate correctly. However, by means of two buffers, we can solve the problem whereby the NAND gate drives the two buffers and each of these drives three inverters and so the maximum fan-out specification is not exceeded.

It is important to note that although the buffer serves no logical function, as with all logic gates, it does exhibit a propagation delay and therefore when the logic state applied to the input is changed, a short time will elapse before this change appears at the output.

2.3 Combinational and sequential circuits

The gates introduced in the previous section provide the essential building blocks from which the computer is constructed. These gates are fabricated in large numbers on the CPU chip and interconnected to provide the required functionality. They are also used in the implementation of memory and other devices within the computer.

Gates may be used to create circuits that are broadly divided into two categories:

- **Combinational logic circuits**
- **Sequential logic circuits.**

These are briefly summarised below:

- **Combinational logic**: A combinational logic circuit has outputs that are completely defined by the combination of input signals applied to the circuit. Thus, given a certain set of binary input values, the circuit will produce a certain result (output). For example, the AND, OR and NOT gates provide examples of combinational logic circuits.

As we will see in the next section, these may be interconnected to provide us with more complex combinational logic circuits

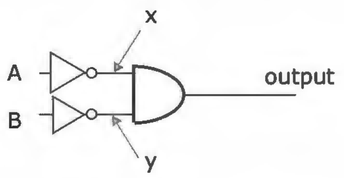
- **Sequential logic:** A sequential logic circuit differs from combinational logic in that the output(s) depend not only on the combination of inputs applied to the circuit but also on the sequence in which they occur (i.e. on some previous state). The concept of sequential logic will be familiar to you – but the name may not be. For example, a TV may have a single ON/OFF button, and the TV may be in one of two states: ON or OFF. The effect of pressing the ON/OFF button will depend on the state of the TV before the button is pressed. If, for example the TV is already turned on, then the effect of pressing the button will be to turn it off. Similarly, if the TV is turned off, then pressing the button will turn it on. Thus the output resulting from the input depends on where we are in the on/off sequence. Sequential logic circuits are constructed using the combinational logic gates of the sort described in the previous section.

2.4 Example combinational logic circuits

In this section we briefly examine some simple circuits constructed by means of the logic gates introduced in Section 2.2. Here, we will examine several circuits and draw up truth tables for them. All circuits discussed in this section fall within the 'combinational' category mentioned in the previous section.

Example circuit 1:

Consider the circuit provided in Figure 2.14. For all possible combinations of input, let us determine the corresponding outputs. This may be readily achieved by drawing up a truth table and including within this table the logic levels that appear on the connections between the gates (in this case there are two – labelled x and y in the illustration.



A	B	x	y	output
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

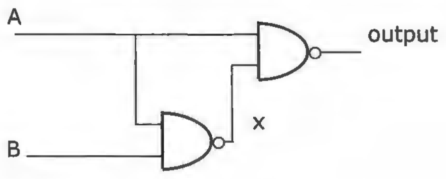
Figure 2.14: Combinational logic example circuit

If we compare the input and corresponding outputs given in this table to the truth tables presented in Section 2.2, we will see that this circuit is acting as a NOR gate. Thus a NOR function may be implemented by means of two inverters and an AND gate.

Activity 2.6

Circuit behaviour 1

Consider the circuit provided in Figure 2.15. Complete the truth table so as to show the output from the circuit for all possible combinations of input.



A	B	x	output
0	0		
0	1		
1	0		
1	1		

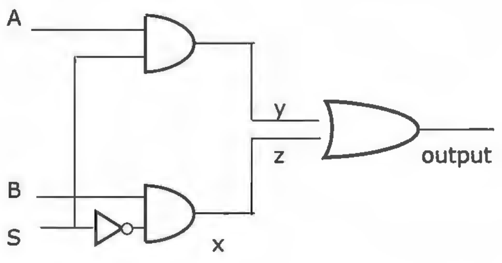
Truth table

Figure 2.15: Combinational logic example circuit

Activity 2.7

Circuit behaviour 2

Consider the circuit provided in Figure 2.16. For all possible combinations of input, determine the corresponding outputs. Hint: you should employ the same approach that was used in the previous activity and complete the truth table provided.



A	B	S	x	y	z	output
0	0	0				
0	1	0				
1	0	0				
1	1	0				
0	0	1				
0	1	1				
1	0	1				
1	1	1				

Truth table

Figure 2.16: Combinational logic example circuit

Examine the truth table that you have drawn for Activity 2.7. Here, you will see that when the input 'S' is a logic low, then the output corresponds to whatever input signal is applied to 'B'. Conversely, when 'S' is a logic high, the output follows whatever signal is applied to input 'A'.

The type of circuit is referred to as a multiplexer and is widely used in digital electronics. In fact, in circuit diagrams a special symbol is often used to represent the multiplexer – as illustrated in Figure 2.17(a). In Figure 2.17(b) a diagram is presented which shows the functional behaviour of the multiplexer.

We can consider this device to act as a switch, with the position of the switch being determined by the value of the 'select' signal. Consequently, if the 'select' signal has one particular binary value, the output will be governed by the signal applied to one of the inputs (say 'input 1') and when the value of the 'select' signal is changed, the output will be driven by the binary values applied to the other input ('input 2'). In fact, multiplexers often have more than two inputs – perhaps eight or sixteen – in which case more than one 'select' signal is required. For example, in the case that a multiplexer has eight inputs, then three 'select' signals are required. This is because the three signals may take on $2^3=8$ different binary values and each of these values determines the particular input that will drive the output.

Activity 2.8

A 16-line multiplexer

Consider a multiplexer with 16 inputs and a single output. How many select wires (lines) are required to enable any of the inputs to be routed through to the output?

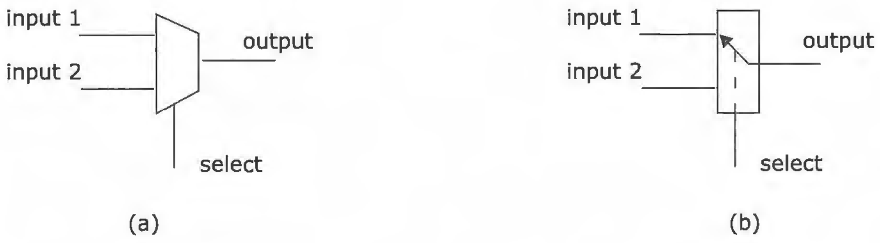


Figure 2.17: In (a) the symbol for a multiplexer is given and (b) indicates the effective action of this device.

The half-adder

Here we briefly consider the creation of a circuit able to add together two binary digits (bits). Before we do this, let us briefly look at the process of adding binary numbers.

First, consider adding two 3-bit binary numbers. For example:

$$\begin{array}{cccl} 1 & 1 & 0 & + \\ 0 & 1 & 1 & \\ \hline \end{array}$$

We start the addition process with the rightmost column (the least significant bits) and work our way to the left.

$$\begin{array}{cccl} 1 & 1 & 0 & + \\ 0 & 1 & 1 & \\ \hline & & 1 & \end{array}$$

As we add, we may need to carry. In this example, we begin by adding 0 and 1. What should we carry? Technically, you do not have to carry anything. However, when this process is implemented in hardware, we always need to define a carry value, which in this case is zero.

We therefore carry a 0 into the next column, and then add that column.

$$\begin{array}{cccl} & \mathit{0} & & \\ 1 & 1 & 0 & + \\ 0 & 1 & 1 & \\ \hline & 0 & 1 & \end{array}$$

This time, when we add the middle column, we get 0 + 1 + 1 which sums to 0, with a carry of 1.

$$\begin{array}{ccccl} & \mathit{1} & \mathit{0} & & \\ & 1 & 1 & 0 & + \\ & 0 & 1 & 1 & \\ \cline{2-4} (1) & 0 & 0 & 1 & \end{array}$$

The final (leftmost) column adds 1 + 1 + 0, which sums to 0, and also generates a carry. We put the carry in parentheses on the left.

Typically, when we perform an addition of two k-bit numbers, we expect the answer to be k-bits. If the numbers are represented in binary, the result can be k+1 bits. To handle that case, we have a carry bit (the one written in parentheses above).

It makes sense to design a circuit that adds in 'columns'. Let us consider adding the rightmost column. We are adding two bits. Therefore the adder we want to create should have two inputs. It generates a sum bit for that column, plus a carry bit. So there should be two outputs. This device is called a half-adder.

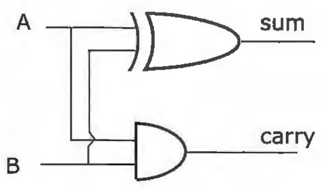
- **Data inputs:** 2 (call them A and B)
- **Outputs:** 2 (call them SUM, for sum, and CARRY, for carry).

A circuit for a half-adder together with its associated truth table is presented in Figure 2.18 and uses an AND gate together with an XOR gate.

Activity 2.9

The half-adder

Verify the accuracy of the truth table presented in Figure 2.18 for the half-adder.



A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2.18: A half-adder and its associated truth table

As may be seen, a 'carry' value of 1 is generated only when both inputs are a logic high, and the 'sum' is zero when we add together two zeros or two ones (in this latter case a 'carry' is produced). This circuit is called a half-adder because although it produces a 'carry out' of the current arithmetic operation, it does not permit a 'carry in' from a previous arithmetic operation.

2.5 Example sequential logic circuits

In Section 2.3 we introduced the concept of the 'sequential logic' circuit. Here, the logic state(s) of a circuits output(s) depends not only on the present values of the inputs applied to the circuit but also the circuits previous state. In this section we provide several examples of circuits of this type.

Consider the circuit presented in Figure 2.19. As may be seen, this circuit employs 'feedback' since the outputs of the two NAND gates are 'fed back' to act as inputs. The two outputs are labelled Q and $\overline{Q}$(generally expressed as 'Q bar'). The use of the 'bar indicates that the two outputs are always (or should be) in opposite logic states (if, for example Q is a logic low then $\overline{Q}$ will be a logic high).

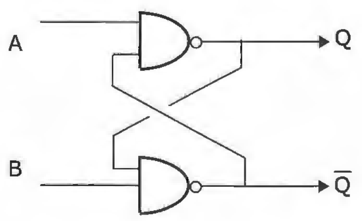


Figure 2.19: A simple example of sequential logic. This circuit is known as the 'RS bistable'.

Consider the case that A is a logic low and B a logic high. Using the same approach that was adopted in the previous section, we can readily determine the output states (recall that for a NAND gate 'any low gives a high'). Since input A is low, it follows that the output from the top gate will be a logic high (thus Q is high). Both inputs to the lower NAND gate are high, and so its output will be a logic low. Thus Q=high and $\overline{Q}$ =low. If we now change input A to a logic high state (leaving B unaltered), then this will have no impact on Q and $\overline{Q}$ (in fact we can say that the circuit 'remembers' its previous state).

Now consider the case that A is a logic high and B a logic low. Using the same reasoning as was used in the previous paragraph, it follows that Q will be a logic low and $\overline{Q}$ a logic high. If we now change input B to a logic high and leave A unaltered this will have no effect on the outputs. Again the circuit 'remembers' its previous state.

Unfortunately, if a logic low is applied to both inputs, then any subsequent change to either one of the inputs has an undefined impact on the circuit. This can be understood by examination of the circuit. If both inputs are a logic low, then the outputs from both NAND gates will be high. If we now change A and B to a logic high, we cannot predict the states of Q and $\overline{Q}$ – this will simply depend on slight differences in the propagation delays of the two gates and so the circuit cannot be viewed as being stable. Consequently, we must avoid simultaneously applying a logic low to both inputs.

The operation of this circuit can be summarised as follows:

- If A=0 and B=1 then Q=1 and $\overline{Q}$ =0. Call this state 1
- If A=1 and B=0 then Q=0 and $\overline{Q}$ =1. Call this state 2
- If we now make A=1 and B=1 then the circuit will retain its previous state (state 1 or state 2) – i.e. it will 'remember' its previous state.

Note that when A=0 and B=1 then Q=1 and $\overline{Q}$ =0. Similarly when A=1 and B=0 then $\overline{Q}$ =1 and Q=0. Thus the output from the circuit corresponds to the inverted input. When Q=1 and $\overline{Q}$ =0 the circuit is said to be 'set' and in the converse case (Q=0 and $\overline{Q}$ =1) the circuit is referred to as being in the 'reset' state. Additionally, when both inputs are a logic high the circuit is said to 'latch' (remember) the previous input.

As we have seen, the circuit can latch two input states (1,0 and 0,1) and the output is the inverted input. Since the circuit can exhibit two stable states it is referred to as a 'bistable' device and is generally called an 'RS latch' or 'RS bistable'. Here, the R and S indicate the set and reset conditions in which the circuit is able to exhibit stability.

Activity 2.10

The RS bistable

Consider the circuit presented in Figure 2.20.

Here, the two NAND gates employed in Figure 2.19 have been replaced with two NOR gates. For this circuit, if A and B are both a logic high the input is said to be invalid. Thus there are three valid combinations of input 0,0; 0,1; 1,0. When the two inputs are a logic low, the circuit will latch the previously applied values (0,1 or 1,0).

When A=1 and B=0 and when A=0 and B=1, determine the logic states of Q and $\overline{Q}$.

Does this device invert the input values?

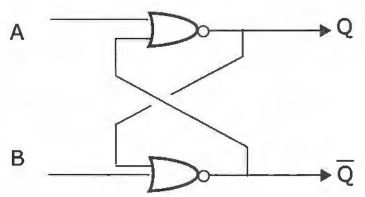


Figure 2.20: The implementation of a bistable using NOR gates

The level-triggered D-type bistable

The simple form of circuit described above in which output is fed back to the circuit input underpins the operation of various forms of computer memory. Unfortunately, the circuit operates incorrectly when both inputs are in a logic low state, and so this condition must be avoided. One solution is to make use of two additional gates as illustrated in Figure 2.21. Here, we have only a single data input (D) and an additional input that is labelled 'clk' (a frequently used abbreviation for 'clock').

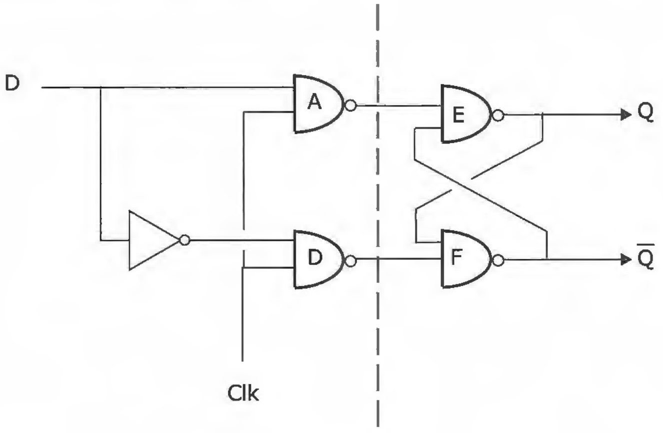


Figure 2.21: The level-triggered D-type bistable

Although at first sight this circuit may look complicated, its operation can be readily understood by remembering that the bistable (located to the right of the dashed line) may only be presented with three binary combinations (0,1; 1,0; 1,1). We aim to prevent it ever being simultaneously presented with two logic lows.

The clock ('clk') is used to enable data to be presented to the bistable and subsequently latched. When the clock is a logic low, then the output from gates A and B will be a logic high (recall that for a NAND gate 'any low gives a high') – in this situation input D has no effect. Thus both gates within the bistable (gates E and F) are presented with a logic high – this corresponds to the latched state.

Now consider the case in which the clock ('clk') is a logic high. If the input D is a logic low, the output from gate A will be a logic high and the output from gate B a logic low (this is because in the case of gate B, the value of D has been inverted and so both its inputs are in a logic high state). Thus the bistable receives 1,0 and so Q=0 and $\overline{Q}$ =1 (the reset condition).

If input D is a logic high, the output from gate A will be a logic low and from gate B a logic high. Thus the bistable is presented with 0,1 and so Q=1 and $\overline{Q}$ =0 (the set condition).

Summary

- We apply either a logic high or low to input D when the clock is in a logic high state. The value of D sets or resets Q and $\overline{Q}$. If now the clock changes to a logic low (before we remove the value applied to input D), the bistable will latch (remember) the value of the signal that was applied to input D. We have a memory device able to store one binary digit (bit).

This circuit is referred to as the 'level-triggered D-type bistable' and there are no circumstances (other than circuit failure) in which both of the inputs of the bistable will be simultaneously presented with a logic low. Thus the 'invalid' state is avoided.

The device is said to be 'level-triggered' since it is the binary value (level) of the clock that determines when the bistable is in the latched state. As we will see, many digital circuits employ an alternative approach in which they react to changes in the state of a signal (rather than to the absolute binary value). Such devices are said to be 'edge-triggered'. This can be readily understood by reference to Figure 2.22, which shows the variation of a digital signal (waveform) with time. As indicated, a 'level-triggered' device responds to the absolute binary value of a signal. This contrasts with the 'edge-triggered' device in which changes in the state of the waveform are of importance.

The waveform presented in this illustration is idealised in as much as the changes in state appear to occur instantaneously. However, in the case of a real signal these changes occupy a finite (although very small) time. The transition from a logic low to a logic high corresponds to the waveform's 'rising edge' and the converse to the 'falling edge'.

A register

As we will discuss in Chapter 3, within a microprocessor there are a number of registers. These are each able to store one or more bytes, and support both the operation of the processor and the implementation of arithmetic and logical operations. Since the registers are implemented on the processor chip, they operate at very high speeds.

For our present purposes it is sufficient to note that registers are simply storage locations able to 'remember' one or more bytes. We can implement such a register by means of a set of D-type bistables (the number of bistables needed corresponds to the number of bits that are to be stored within the register). For simplicity, we will assume that a register is to hold (store) four bits, and a suitable circuit is illustrated in Figure 2.23. Here, you will notice that each bistable device is simply represented by a rectangle, each of which has two inputs ('D' and 'clk') and two outputs Q and $\overline{Q}$. For the moment we will assume that these are 'level-triggered' devices of the sort that were illustrated in Figure 2.21.

4.1 Introduction

In the first two sections of this chapter we consider the representation of several types of number within a computer. Up to this point we have focused on the representation of positive integer values (positive 'whole' numbers). In order to perform useful and wide-ranging functions, a computer must also be able to store and process negative integers and numbers that have a fractional (decimal) component. Furthermore, the computer must also be able to store (with a sufficient degree of precision) numbers that are either extremely small or extremely large. Such issues are discussed in Sections 4.2 and 4.3.

In Section 4.4 we turn our attention to the representation of symbols (such as letters of the alphabet), and here we introduce the ASCII character codes. This leads on to brief discussion in Section 4.5 relating to the means by which the processor is able to determine the manner in which operands and data should be treated. In this context it is important to remember that the processor has no inbuilt knowledge of what we are trying to achieve, nor the meaning that we assign to the operands and data that are to be processed. After all, the digital world knows only of the logic high and logic low states – the binary ones and zeros – and it is up to the programmer to ascribe meanings to these states and ensure that they are operated upon in the desired manner.

In Sections 4.6 and 4.7 we introduce the use of 'interrupts' within a computer. These provide a system whereby the processor is able to efficiently determine when peripheral devices such as the hard and floppy disk drives require attention and underpin computer communication.

4.2 Representing positive and negative integers

In previous chapters we confined our discussion to the representation of positive integers. For example, the base 10 number 27 may be represented as the binary value 11011. If this number were to be stored within an 8-bit register, then we would simply pad it out by inserting leading zeros, i.e. 00011011. By way of a further example, consider the base 10 number 15. This would be represented within an 8-bit register as the binary value 00001111.

If we have a binary number comprising N bits, then this may represent 2^N different values. For example an 8-bit number may represent 2^8 (=256) different values ranging from zero through to 255.

A computer must be able to deal with both positive and negative numbers, and one simple approach to the representation of negative numbers is to dedicate one bit to indicate a number's sign. This may be achieved as indicated in Figure 4.1 where we show one bit (the left-most bit) as indicating the sign (positive or negative), and the remaining bits as indicating the magnitude (size) of the number. In this scenario, we could adopt the convention that if the sign bit is a zero then the number is positive, and if the bit is a one, then a negative number is indicated. (This convention is arbitrary. Equally, a zero could be used to indicate a negative number, and a one to indicate a positive number. Two examples of this approach are illustrated in Figure 4.2.)

The use of a sign bit gives rise to the number zero being represented in two ways – see Figure 4.3. Consider a binary number comprising N bits (one of these bits representing the number's sign, and N-1 bits the number's magnitude). This enables 2^N-1 different binary values to be represented (the '-1' arises because the number zero is represented in two different ways).

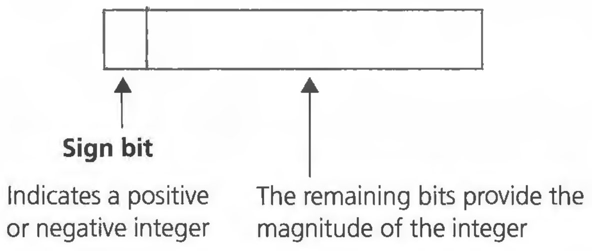

Sign bit
Indicates a positive or negative integer
The remaining bits provide the magnitude of the integer

Figure 4.1: The representation of positive and negative numbers by means of a 'sign' bit. We may, for example, adopt the convention that if the sign bit is a binary *zero* then the integer is *positive*, whilst if the sign bit is a binary *one* then the number is *negative*.

In the early days of computing, the implementation of the electronic circuits that comprise the computer was a difficult and laborious task. Consequently, one key objective of the computer designer was to minimise the number of logic gates needed to perform computational (arithmetic) operations. Although the representation of positive and negative binary numbers in machine code programs using a 'sign' bit (and associated magnitude bits) is logical, it means that addition and subtraction operations have to be carried out by largely separate hardware. In essence, it becomes necessary to implement hardware for the addition of numbers, and different hardware for their subtraction. Designers therefore sought alternative solutions and developed number representation schemes that would permit both addition and subtraction operations to be performed by the same hardware. Below we review the use of the 2s complement approach.

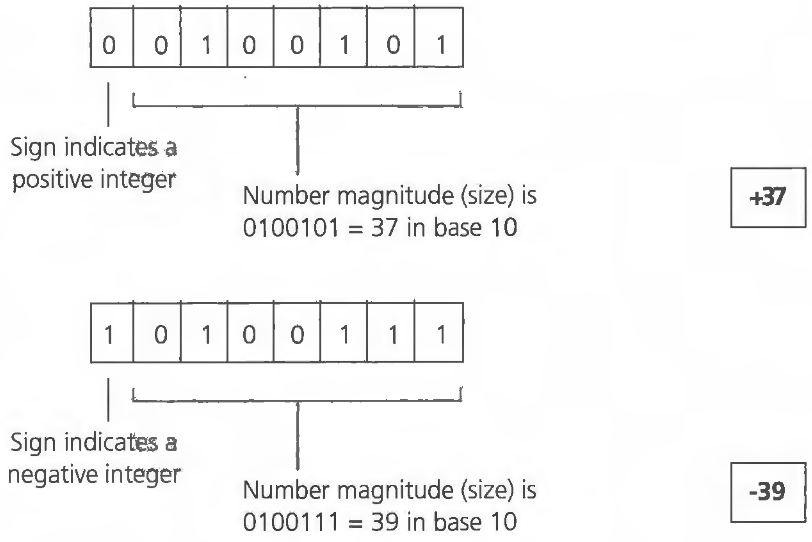

0 0 1 0 0 1 0 1
Sign indicates a positive integer
Number magnitude (size) is 0100101 = 37 in base 10
+37
1 0 1 0 0 1 1 1
Sign indicates a negative integer
Number magnitude (size) is 0100111 = 39 in base 10
-39

Figure 4.2: The representation of positive and negative integers using a 'sign' bit

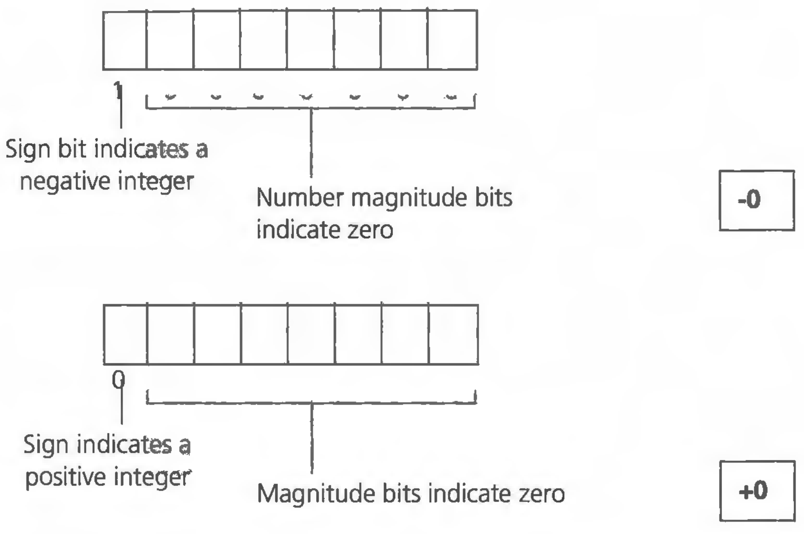

1
Sign bit indicates a negative integer
Number magnitude bits indicate zero
-0
0
Sign indicates a positive integer
Magnitude bits indicate zero
+0

Figure 4.3: The use of a 'sign' bit results in the number *zero* being represented in two different ways (as +0 and –0). This is non-optimal.

Twos complement arithmetic

Rather than employing a 'sign' bit to indicate whether a number is positive or negative, an alternative approach is to encode the negativity of a number within the number itself. In this context and when working in the binary number base, we make use of what is called a number's '2s complement'.

Although at first sight this may seem complicated, in practice the technique is very simple. Below we provide two techniques that may be used to obtain the 2s complement of an integer:

The first approach may be summarised as follows:

- Change all 1s in the number to 0s; change all 0s to 1s; and add 1.

By way of example, consider the binary number 10101. We begin by changing 1s to 0s and 0s to 1s (i.e. inverting the bits). (The process of swapping 1s to 0s and 0s to 1s is referred to as calculating the 1s complement of a binary number.) Thus we obtain 01010. We then add 1 and this gives 01011. Thus, the 2s complement of 10101 is 01011.

The second approach is even simpler and can be summarised as follows:

- Inspect the number starting at the right-hand side (least significant end). For all bits to the left of the first '1' that you encounter, change all 1s to 0s and all 0s to 1s.

Taking the first example used (10101), the first digit is a one. We leave this unchanged and swap all ones and zeros located on its left-hand side. This gives the same result as above – 01011.

Let us try employing these two approaches on two more example numbers:

Example 1

Find the 2s complement of the binary number 11001100.

- Using the first approach, we swap 1s and 0s – thereby obtaining 00110011. We then add one – this gives: 00110100.
- Using the second approach, we move along the number for the right-hand side and make no changes until we encounter the first '1'. Subsequently, we swap (invert) 1s and 0s. Thus the first three bits are not changed and all others that are a '1' become '0', and all that are '0' become '1'. This gives: 00110100.

Example 2

Find the 2s complement of the binary number 11000000.

- Using the first approach, we swap 1s and 0s – thereby obtaining 00111111. We then add one – this gives: 01000000.
- Using the second approach, we move along the number for the right-hand side and make no changes until we encounter the first '1'. Subsequently, we swap 1s and 0s. Thus the first seven bits are not changed and the eighth bit becomes a zero This gives: 01000000.

NOTE: If you calculate the 2s complement of a number and then repeat the process on this result, you obtain the original number. This provides a convenient method of checking your arithmetic. For example, consider the number 1100110. If we take the 2s complement of this number (using either of the above methods), we obtain 0011010. If we take the 2s complement of this number, we obtain 1100110 – the original value!

Activity 4.1

The 2s complement technique

Calculate the 2s complement of each of the following binary numbers:

(a) 1101010

(b) 1000011

(c) 1110000

(d) 1010100

It is recommended that in each case you employ both of the methods introduced above.

In order to subtract one binary number from another, we can make use of 2s complement arithmetic. Let us suppose that we wish to perform the following calculation:

$$A - B = C .$$

Where A and B represent two binary numbers each comprising N bits. We simply calculate the 2s complement of B (we will refer to this as B') and then compute:

$$A + B'$$

In other words, by first taking the 2s complement of B, the subtraction of B from A can be performed using addition!! There is however one small complication – when performing subtraction operations, the result is sometimes negative. However, when we use the 2s complement approach the sign of the answer (positive/negative) may be determined from the value of the carry (overflow) that occurs. Let us take a couple of examples and see how this works in practice:

Example 1: Calculate the binary subtraction 11001 – 10011, using the 2s complement approach

Before we begin, it is convenient to convert both these binary values to base 10 and see what answer we should obtain for the subtraction. Recall from Chapter 1 that 11001 equals 25 (base 10) and 10011 equals 19 (base 10). The subtraction is 25 - 19 = 6 and therefore in base 2 the answer that we obtain should equal 110. (For convenience, we may add two leading zeros so that the result comprises the same number of bits as the two original numbers, i.e. 00110).

Now let us try the 2s complement approach. In this question A = 11001 and B = 10011. We begin by finding the 2s complement of B (10011) which is 01101 (we will call this B′). We now add A and B′ together, see Figure 4.4.

A		1	1	0	0	1
B′		0	1	1	0	1
C	*1*	0	0	1	1	0

This bit is the carry (overflow) from the addition

Figure 4.4: Performing subtraction using 2s complement arithmetic

As may be seen, we obtain a result of 00110 with a carry (overflow) of one. If the overflow bit is a one, then this indicates that the result is positive – if there is no overflow (i.e. the carry/overflow bit is zero) then the result is negative.

Recall from above that when we performed this calculation in base 10, we obtained an answer of 6 (binary 110). Thus the 2s complement approach provides the correct result. (Note: to obtain the magnitude of the answer, we discard the overflow bit).

Example 2: Calculate the binary subtraction 110010 – 111010, using the 2s complement approach

As with the previous example, it is instructive to convert both of these binary values to base 10 and see what answer we should obtain for the subtraction. Recall from Chapter 1, 110010 equals 50 (base 10) and 111010 equals 58 (base 10). The subtraction 50 - 58 = -8 (i.e. a negative answer).

Now let us try the 2s complement approach. In this question A = 110010 and B = 111010. We begin by finding the 2s complement of B (111010) which is 000110 (we will call this B′). We now add A and B′ together, see Figure 4.5.

A		1	1	0	0	1	0
B′		0	0	0	1	1	0
C		1	1	1	0	0	0

Here, the carry (overflow) is zero

Figure 4.5: Performing subtraction using 2s complement arithmetic

As may be seen, we obtain a result of 111000 with a carry (overflow) of zero. As shown above, this indicates that the result is a negative number.

Recall from above that when we performed this calculation in base 10, we obtained an answer with a magnitude of 8 (binary 1000), and negative in sign. However, the 2s complement method that we have used gives us an answer of 11100. Something seems to have gone wrong!

The answer that we have obtained is a negative value and as such, in order to find the actual magnitude (size) of the result we need to take its 2s complement. The 2s complement of 111000 is 001000 and in base 10 this equals 8. Thus the 2s complement technique has provided the correct answer.

For convenience, the overall technique is summarised in Figure 4.6 and an important point to remember about this method is that it permits both the addition and subtraction of binary integers to be performed by a single hardware unit that performs only addition operations.

4.3 Dealing with non-integer numbers

So far we have confined our discussions to the representation within a computer of positive and negative integers ('whole' numbers). In this section, we briefly consider ways in which we can represent and manipulate numbers that have a fractional/decimal component (e.g. 3.142). We begin by summarising some basic maths.

A little mathematics: Exponents – base 10

Consider the following expression:

$$3.24 \times 10^2.$$

The power (2) is referred to as the exponent whilst the number appearing on the left-hand side (3.24) is called the mantissa. This expression evaluates as follows:

$$3.24 \times 10^2 = 3.24 \times 100 = 324.$$

An alternative way of viewing this is to say that the exponent (power to which 10 is raised) indicates the number of places the decimal point should be shifted (a positive exponent indicating that the decimal point should be moved to the right). Thus, in the case that the