



School of Design, Engineering & Computing

B.Sc. (Hons) Software Engineering Management
B.Sc (Hons) Computing
HNC Computing

Systems Architecture
Term 1 Lab Book

Stephen Welsh
18 September 2003

Contents

1	<i>Introduction</i>	1
1.1	The Aims of this Unit	1
1.2	Overview of Digital Computer Systems	1
1.3	Introduction to Digital Works	2
1.4	Exercises	5
2	<i>The Digital Computer and the Binary System</i>	7
2.1	What is 'Digital'	7
2.2	Binary Numbers	7
2.3	Binary Arithmetic	8
2.3.1	Binary Addition	8
2.3.2	Binary Subtraction	9
2.4	The Hexadecimal System	10
2.5	Two State Devices	10
2.6	Digital Signals/Media	11
2.7	The Digital Computer and Microprocessor	13
2.8	The Internet	15
2.8.1	Digital Equipment and Systems	17
2.9	Exercises	17
3	<i>Combinational Logic</i>	19
3.1	Basic Gates	19
3.1.1	The AND logical function	19
3.1.2	The OR logical function	21
3.1.3	NOT logical function	23
3.1.4	The NAND logical function	24
3.1.5	The NOR logical function	24
3.1.6	The Exclusive-OR logical function	25
3.1.7	The exclusive-NOR logical function	25
3.1.8	Gates with inverted inputs	25
3.2	Multiplexers	26
3.3	Encoders/Decoders	26
3.3.1	Encoders	26
3.3.2	Decoders	27
3.4	Adders	27
3.5	Macros in Digital Works	29
3.6	Fan-In and Fan-Out	32
3.7	Exercises	32
4	<i>Karnaugh Map</i>	35
4.1	Sum-of-Product Equations	35

4.2	2-Variable K-Maps	35
4.3	3-Variable K-Maps	36
4.4	4-Variable K-maps.....	37
4.4.1	Inverse of a function	38
4.4.2	Don't care/Not possible states	38
4.4.3	Further Simplification.....	39
4.5	Exercises.....	39
5	<i>Sequential Logic/Flip-Flops.....</i>	<i>41</i>
5.1	The R-S Latch.....	41
5.2	The D-Type Flip-Flop	42
5.3	The J-K Flip-Flop.....	43
5.4	Clocking Types	43
5.4.1	Level sensitive	43
5.4.2	Rising edge/Falling edge.....	45
5.4.3	Master/Slave Flip-Flop	45
5.5	Exercises.....	45
6	<i>Useful Circuits</i>	<i>47</i>
6.1	Memory	47
6.2	Memory in Digital Works.....	49
6.3	Registers	50
6.4	Shift Registers.....	51
6.4.1	Serial in/serial out	52
6.4.2	Serial In/Parallel Out	53
6.4.3	Parallel In/Serial Out	53
6.4.4	Parallel In/Parallel Out.....	54
6.4.5	Bi-directional Shift Registers.....	54
6.4.6	Universal Shift Registers	55
6.5	Counters	56
6.5.1	Asynchronous Counters	57
6.5.2	Synchronous Counters	58
6.6	Exercises.....	60
7	<i>Binary Arithmetic</i>	<i>63</i>
7.1	Sign and Magnitude Representation	63
7.2	Ones Complement	63
7.3	Twos Complement.....	64
7.4	Subtraction using Ones Complement and Twos Complement	65
7.5	Exercises.....	66
8	<i>Floating Point Numbers.....</i>	<i>67</i>
8.1	Scientific or Exponential Notation.....	67
8.2	Floating Point Format	68
8.3	Bias	68

8.4	Normalisation	69
8.5	Special Values	70
8.5.1	Denormalised.....	71
8.5.2	Zero	71
8.5.3	Infinity	71
8.5.4	Not a Number (NaN).....	71
8.6	IEEE Expressions	71
8.7	Exercises.....	72
1.1.	Decimal System	74
1.2.	Hexadecimal System	74
1.3.	Octal System	74
1.4.	Binary Coded Decimal	74
1.5.	Converting between Number Systems	75
1.5.1.	Decimal to Binary	75
1.5.2.	Binary to Decimal	76
1.5.3.	Decimal to Hexadecimal.....	77
1.5.4.	Hexadecimal to Decimal.....	78
1.5.5.	Hexadecimal to Binary	79
1.5.6.	Binary to Hexadecimal	79
1.5.7.	Decimal to Octal	80
1.5.8.	Octal to Decimal	80
1.5.9.	Octal to Binary	80
1.5.10.	Binary to Octal.....	81
1.6.	Binary and Hexadecimal Fractions.....	81
1.6.1.	Binary Fractions.....	81
1.6.2.	Hexadecimal Fractions	83
1.7.	Exercises.....	83

1 Introduction

1.1 The Aims of this Unit

By the end of this first term unit, students should:

- *Understand* fundamental concepts in the design of digital circuits and systems.
- *Understand* and have a working knowledge of Boolean algebra and its application to combinational logic circuits.
- *Understand* and have a working knowledge of sequential circuitry and how to implement such circuits in the laboratory.
- *Understand* and have a working knowledge of real world timing problems and alternative solutions in both combinational and sequential circuits.
- *Understand* and have a working knowledge of numbering systems, and their relationship to hardware and software implementations.

1.2 Overview of Digital Computer Systems

The ubiquitous spread of digital computer systems over the last three and a half decades is astonishing. Every aspect of our lives is touched by it, whether we are aware of it or not. From shops, banks, petrol stations and offices to mobile phones, the Internet, cameras, washing machines, traffic lights and myriad other facets of our lives.

Many students are confronted with hardware issues when they undertake their industrial placement, and a good understanding of the basics of computer systems architecture stands them in good stead with their placement employer. A typical scenario would be that on the first day of placement employment, the student is given a bare PC, a new hard disk, a digitiser card, a network card and a copy of Windows XP and told – ‘Make that work!’.

All of these computer systems rely upon certain basic principles, and this term is intended to give you an insight into those principles. This, in its turn, will give an appreciation of other aspects of the course: why software is designed as it is, why mathematics is studied in the way that it is, how we interact with computers. All have a basis in the relationship between the software and hardware of systems.

Take for example a PC and a mobile phone. These are, both on their own or together with other equipment, digital computer systems. Both include a user interface comprising a combination of hardware and software. In the case of the PC there will probably be a display, a keyboard and a mouse. There may be other items like joysticks, drawing tablets and so on. The mobile phone has a small monochrome or colour LCD display, and various control buttons. It may also have peripherals such as a hands-free kit or an add-on digital camera.

Both include microprocessors, and both will have temporary and permanent storage. A PC’s temporary storage would typically be random access memory (RAM), and its permanent

storage would include (but is not confined to) hard disks and read only memory (ROM). The mobile phone would again have RAM, but its permanent memory is in the form of the SIM card.

So far so good. We have a device of some sort, with its relevant user interface, its temporary and permanent storage. But what have we got? Without one vital ingredient, all we have got is a useless collection of parts – wires, batteries, displays, microprocessors and so on. All totally useless. What is the missing ingredient? Software!

Here is the major division in digital computer systems. What is hardware? Hardware is the actual physical pieces of equipment. You can touch it, see it, kick it around the room. So what is software? Software cannot be seen. It can't be touched or smelled. Software is the machine code instructions and other data files that make the collection of pieces that is hardware actually do stuff.

Originally, the software had to be manually loaded into the memory of the machine. This was done with a series of switches that were set to the bit pattern to be entered and a push button that actually caused that 'line' of code to be stored in the machine. Obviously this was slow, laborious and error prone (imagine getting to line 995 of a thousand line program and making a mistake!).

John von Neumann developed the concept of the stored program computer, and most of the computers in use today use this architecture. In this concept, a copy of the program is stored in binary form where the machine can access it relatively quickly and easily. This can be on a local hard disk, on tape, on another machine connected via a local network, or nowadays on a machine remote in any part of the world via an Internet connection. When required, this binary code is loaded into the memory of the machine that is to run it.

As the computing world has advanced, much effort has been placed into attempting to make the production of machine code a simpler and less error prone process. As stated, the machine code can be directly entered into the computer. The next step is assembler language, in which human readable mnemonics are substituted for machine code instructions. These instructions are then *assembled* by an *assembler* into the binary code. Assembler language will form the basis of term 2 of this unit. Higher still are the so-called third generation languages such as C, C++ and so on. These languages are compiled and linked to form an executable file. Alternatively, they may be compiled to byte code, and linked at run time, as in the case of Java. Finally there are the so-called fourth-generation languages such as Structured Query Language (*SQL*) that attempt to get as close as possible to a human readable source.

All this is aimed at making computer programming easier. It has been said that writing non-trivial software is one of the most difficult undertakings attempted by man. The understanding of computer architecture that this unit is seeking to impart, will assist in creating a greater awareness of what we are seeking to achieve when writing computer software.

1.3 Introduction to Digital Works

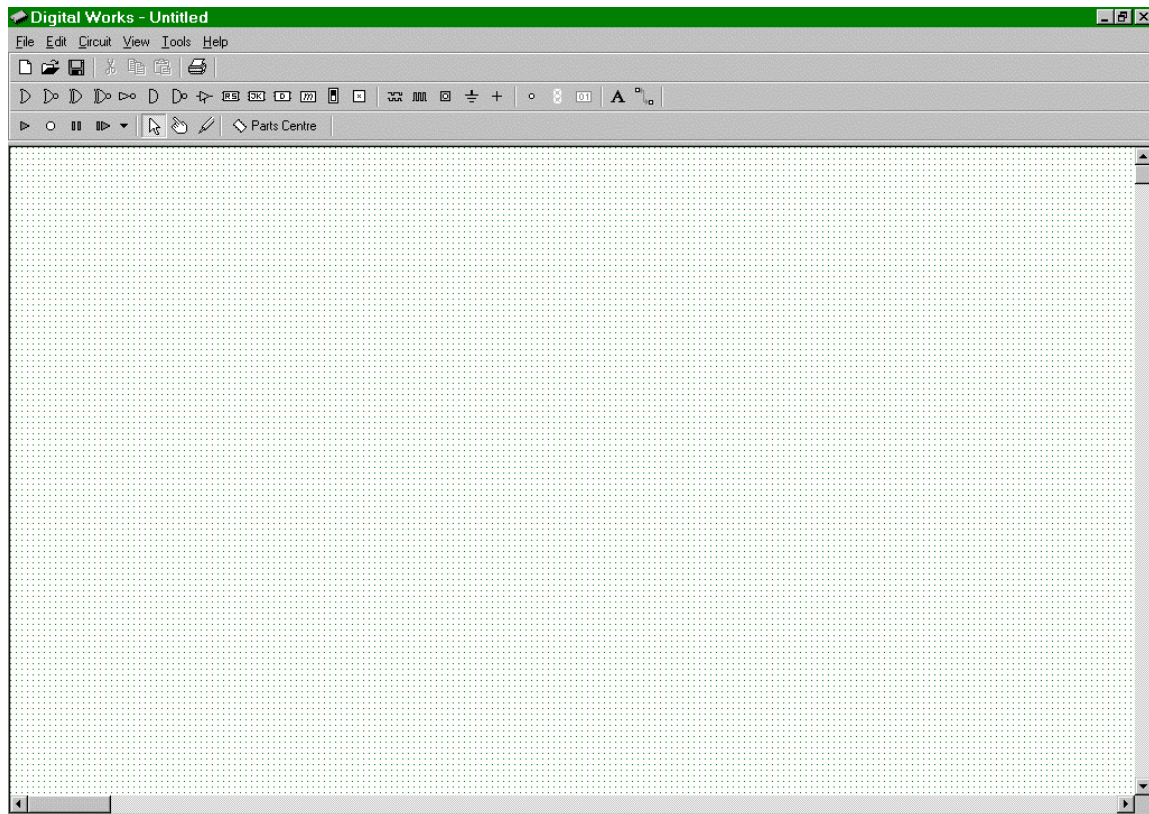
During the first term, the software tool that will be used is Digital Works V3.04. This is a tool that allows you to simulate the building and running of hardware at its lowest level i.e. individual gates, flip-flops, LEDs, clocks, etc. It will be used for the first Assignment in this unit.

A Word of Caution

If you install Digital Works V2.0, which comes on the CD included with Clements, The Principles of Computer Hardware, on your own machine, you will be able to build components at home and they will work on DW V3.04 at the University. However, V3.04 is **not backwardly compatible**, and if you have opened files with the later version, **you will not be able to open them again** with V2.0.

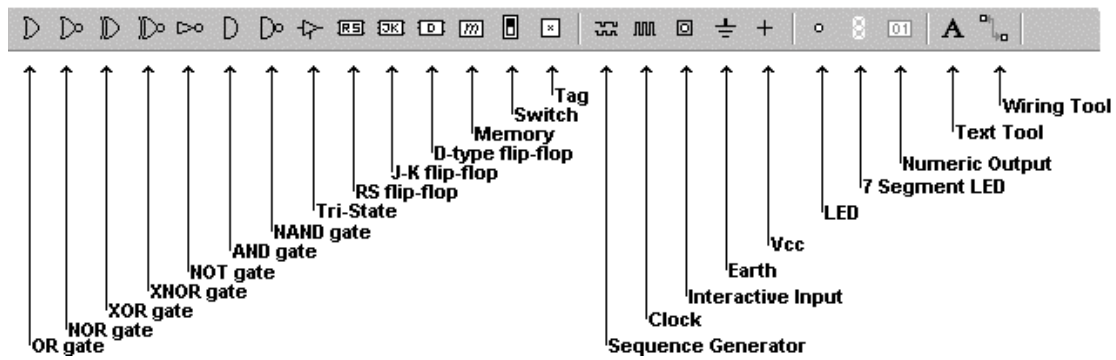
The interface of Digital Works is shown below as Figure 1.1.

Fig. 1.1



Below the menu bar are three toolbars. The top one is a fairly standard type of GUI toolbar with open new, open folder, save, cut, copy, paste and print buttons. The next toolbar allows the user to select and drop components onto the canvas. Multiple copies of an item may be dropped by holding down the <Ctrl> key. The meaning of the icons on this toolbar are shown in Figure 1.2. It can be seen that they are grouped into four sections. These sections represent: devices, inputs, outputs and 'other'.

Fig. 1.2



The only buttons on this toolbar that are not select and drop are the text tool and the wiring tool.

The text tool allows text annotations to be placed anywhere on the canvas.



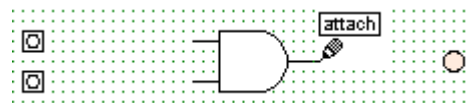
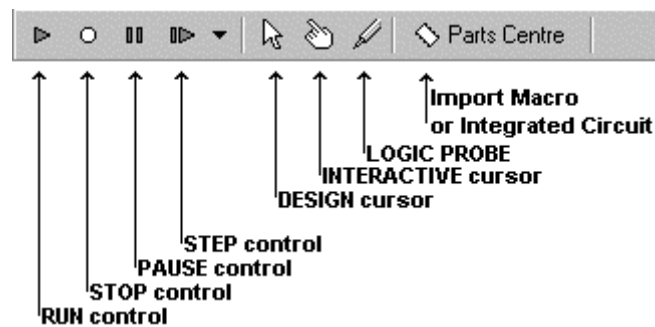
The wiring tool allows components to be ‘wired’ together. When it is selected and the cursor placed over the canvas, the cursor icon will change to . When the cursor is in a position where it is possible to connect a wire, the cursor will then change to  , as shown in Figure 1.3.

Fig. 1.3






The next toolbar down is the run-time toolbar, as shown in Figure 1.4

Fig. 1.4



The RUN control will set a simulation running, the STOP control will stop it, the PAUSE control will leave the simulation running but will stop any clocks contained in it. The STEP control advances the simulation HALF A CLOCK CYCLE each time it is clicked by default, but there is a drop-down menu that allows you to change this to a full clock cycle.

It is not recommended that you make design changes whilst running a simulation. ALWAYS STOP your circuit before making changes. You will also always have to select the INTERACTIVE CURSOR before you can interact with those components that permit it. Notice how the cursor changes to a hand when it enters the canvas area.

The LOGIC PROBE allows you to check the logic status of a connection whilst running a simulation. If you select it, you will notice that the cursor changes to  when it enters the canvas area. When positioned over a 'live' component, if the cursor changes to  this indicates a logic 0, and if it changes to  this indicates a logic 1. This facility is of limited use, however, since in this version of Digital Works, the 'wires' indicate their logic status anyway. A black 'wire' indicates logic status 1, and a greyed-out 'wire' indicates logic status 0.

The package allows for more sophisticated handling of circuits, for instance the use of ready built integrated circuits and macros, and the building of purpose made macros. This functionality will be explained during the weekly seminar sessions.

1.4 Exercises

1. Consider the terms 'computer' and 'microprocessor'. List 6 differing applications of these for discussion.
2. You are now on an undergraduate course in a computer related discipline. Think of a number of reasons why computers are 'a good thing'. Think of a number of drawbacks to a computer-dominated world.
3. With Digital Works
 - (a) Click on the 'Open' button
 - (b) Select the 'Sample Circuits' directory
 - (c) Select the 'Bus Contention' circuit
 - (d) Click on the 'Run' button to run the circuit. What is it doing?
 - (e) Stop the simulation by clicking the 'Stop' button
 - (f) Click on the Macro in the middle of the circuit (labelled 8 bit Register) to select it
 - (g) Right click on the device to bring up the context menu
 - (h) Select 'Open Macro' from the context menu. This will show the circuit inside the IC. Note the use of the Macro Tag to represent the pins of the IC
 - (i) Open either of the two '4 bit Register' macros
 - (j) Bring up the context menu for one of the D-Type flip-flops. You will see that the 'Edit Macro' item is missing. This is because the D-Type flip-flop is considered to be a basic design unit
 - (k) Click on the 'Close Macro' button that has appeared on the bottom toolbar to close the '4 bit Register' macro
 - (l) Click on it again to close the 8 bit Register macro

You will not be asked to design a circuit quite like the bus contention circuit, and you will not be asked to use this IC, but your Assignment will ask you to design circuits of this complexity. You may find the macro facility useful for this.

4. With Digital Works

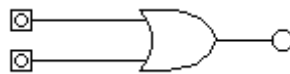
Click on 'New' to generate a clean canvas. Now draw the following circuit:



Note the Vcc is obtained by using the '+' control. Now call up the context menu for the Interactive Switch. Select the 'Set Switch Type' option. You will see that there are types of 'Toggle', 'Push to Make' and 'Push to Break'. Investigate the operation of the circuit in all three modes, noting the results, by selecting the 'Run' button and using the 'Interactive Cursor'. Do not forget to stop the circuit before making design changes.

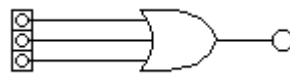
5. With Digital Works

Click on 'New' to generate a clean canvas. Now draw the following circuit:



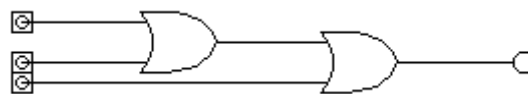
Run this circuit and interact with the 'Interactive Inputs'. What does it do?

Now stop the simulation and call up the context menu. Select the 'Inputs' option and then select '3 – Three'. The gate will change from a two input to a three input gate. Confirm this by drawing up the following circuit and running it:



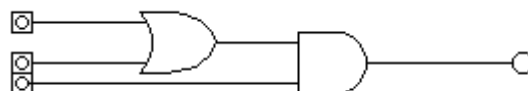
6. With Digital Works

Click on 'New' to generate a clean canvas. Now draw the following circuit, remembering that holding down the <Ctrl> key allows you to place multiple copies of an item:



What does this circuit do? How does it differ from the previous circuit?

7. Draw the following circuit. What does it do, and how does it differ from the circuit in Q.6?



2 The Digital Computer and the Binary System

2.1 What is 'Digital'

Before we examine the binary system and other numbering systems which follow in this chapter, let us first consider exactly what we mean by expressions such as analogue, digital and discrete.

The Free On-line Dictionary of Computing defines digital as:

“A description of data which is stored or transmitted as a sequence of discrete symbols from a finite set, most commonly this means binary data represented using electronic or electromagnetic signals.”

Using this definition, there is immediately a clear distinction to be drawn between this and any system that can be regarded as 'analogue'. An analogue system by its nature deals with infinitely variable values. The type of computer that we shall be dealing with here cannot do this, which is why they are referred to as 'digital'. You will also immediately see a link to the discrete mathematics that you will study this year, since the definition given uses this term when referring to the description of data.

This notion of 'digital' has important implications which will be explored both in the sections that follow, and particularly in the chapter on the handling of real numbers at the end of this book.

2.2 Binary Numbers

The decimal numbering system used by humans, in which numbers have a base or *radix* of 10, appears to have originated over 5000 years ago in India, and is believed to have its origins in the fact that we have 10 digits (fingers/thumbs). It is unusual in a numbering system of that antiquity in having an appreciation of the number zero so that every number consists of the sum of 0-9 units, 0-9 tens, 0-9 hundreds, 0-9 thousands and so on. Therefore a number such as 2387 means $(2 \times 1000) + (3 \times 100) + (8 \times 10) + (7 \times 1)$, or $2 \times 10^3 + 3 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$. Since only digits 0-9 are used in the decimal system, the number 10 requires two digits, i.e. indicating $1 \times 10^1 + 0 \times 10^0$.

It is not essential for the base, or *radix*, of a numbering system to be 10; the base may be any chosen number. Remote shepherding communities in this country still count in base 5 (yan, tan, tethera, methera, pip). For our purposes on the course it is necessary to have an understanding particularly of the *binary* system (base 2), and the *hexadecimal* system (base 16). The *octal* system (base 8) may also be encountered from time to time. When indicating numbering systems, if there is any chance of misunderstanding the number base in use, the number should be subscripted with its base e.g. 1010_2 is 1010 binary, 1010_{10} is 1010 decimal and 1010_{16} is hexadecimal, all of which indicate totally different numbers. Sometimes a hexadecimal number will be prefixed with the letter H, and many programming languages support the representation of hexadecimal numbers as 0x1234 (1234_{16}). Throughout this lab book, hexadecimal numbers are indicated by the latter convention (e.g. 0xFF).

The importance of the binary system in respect of digital electronics is that it is an ideal representation of the two state devices that form the vast majority of the inner workings of

computers and microprocessors. A binary digit can only be a '1' or a '0', closely corresponding to states of 'On' or 'Off' found in digital circuitry. In fact the term 'bit' comes from *binary digit*. Although binary numbers are ideal in this respect, they are not ideal for people to use. For example, the 16 bit binary number 1101011100110011_2 is equivalent to 55091_{10} or $D733_{16}$.

You should at this stage become familiar with the binary number system, and be able to do basic counting in it, so the following sections will explore the binary system and its relationship to other numbering systems. This chapter gives an introduction; more complex and esoteric topics are covered in a later Chapter.

2.3 Binary Arithmetic

Let us begin by understanding the relationship between the binary and denary systems. The following tables will demonstrate that relationship.

Table 2.1 Powers of 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Table 2.2 Binary Numbers

3	0	0	0	0	0	1	1
11	0	0	0	1	0	1	1
32	0	1	0	0	0	0	0
77	1	0	0	1	1	0	1
99	1	1	0	0	0	1	1
127	1	1	1	1	1	1	1

In accordance with our normal understanding of numbering systems, when written this way the leftmost bit is the *most-significant bit* and the rightmost bit is the *least-significant bit*. You should understand that this is not necessarily the way that the data is represented within the computer or microprocessor, as the bytes may be physically stored in either direction. It is necessary to know which way round they are stored before you can safely perform operations at the 'bit' level. (See <http://www.hive.speedhost.com/Programmers/code5.html> for a gentle introduction).

Binary arithmetic in itself is a very simple and straightforward operation so long as it is realized that the only numeric representations allowed are the '1' and the '0'. Addition and subtraction follow the normal rules that we use every time we do these operations, of 'carry' and 'borrow'. Once this operation is understood, we will consider how positive and negative numbers may be represented in the binary system, with particular reference to how this can be dealt with at the bit level within a computer. You should understand that you should be able to perform all the different manipulations of binary numbers without recourse to calculators. At this stage, you should get a firm grasp upon the various methods that computers use to represent negative numbers in the binary system. You should be aware that number representation has frequently formed the basis of an examination question!

2.3.1 Binary Addition

The easiest way to see how binary addition works is by using some examples, but note the use of the 'carries'. Always remember that:

$0 + 0 = 0$, with no carry,
 $1 + 0 = 1$, with no carry,
 $0 + 1 = 1$, with no carry,
 $1 + 1 = 0$, and you carry a 1.

Example 2.3.1

Let us start with a simple example of addition of unsigned numbers. Consider $6_{10} + 7_{10}$ ($0110_2 + 0111_2$).

Decimal	Binary
06	0110
07	0111
carry <u>1</u>	carry <u>11</u>
13	1101

The only problem that we have with this is the matter of overflow. Consider $9_{10} + 7_{10}$:

Decimal	Binary
09	1001
07	0111
carry <u>1</u>	carry <u>111</u>
16	<u>1</u> 0000

If we are dealing with 4-bit unsigned binary numbers we know that we can only represent 0_{10} to 15_{10} , so as shown above the result of adding $9 + 7$ will give the answer 0, *with a fifth bit* that is unaccounted for in our calculation (shown in the example as 1). This is arithmetic overflow.

Example 2.3.2

Now let us do a couple of more complex examples. For instance, let us add 0011011101_2 to 0111100101_2 :

Operand 1 +	0	0	1	1	0	1	1	1	0	1
Operand 2	0	1	1	1	1	0	0	1	0	1
Carry	1	1	1	1	1	1	1		1	
Result	1	0	1	1	0	0	0	0	1	0

and a further example:

Operand 1 +	0	0	1	1	1	1	0	1	0	1
Operand 2	0	0	1	0	1	0	0	1	1	1
Carry		1	1	1			1	1	1	
Result	0	1	1	0	0	1	1	1	0	0

2.3.2 Binary Subtraction

Binary subtraction follows exactly the same rules as represented for addition, so:

Decimal	Binary
7	0111
− 3	− 0011
= 4	= 0100

Our normal mathematical rules apply regarding ‘borrows’ apply when dealing with binary numbers, but remembering that in binary $10 - 1 = 1$. So:

Decimal			*		
	Borrow	10	10	10	
8		1	0	0	0
– 3				1	1
	Repay	1	1	1	
= 5		0	1	0	1

(not forgetting that in the column marked * there is a repay and a 1, so 1 + 1 in binary is 10. Thus we are subtracting 10 from the 10 that is borrowed from the next column giving 0)

2.4 The Hexadecimal System

At this stage it is intended that you should also acquire a basic understanding of the hexadecimal system, and its relationship to binary numbers. As we have already written, the hexadecimal system uses a radix of 16. This has an important implication in the fact that there is a direct mapping between the binary system and the hexadecimal system because 16 is 2^4 .

The hexadecimal system uses the digits 0 .. 9, and the letters A .. F in its representation of numbers as shown in Table 2.3 below. There are several good reasons why you should, at this stage, become proficient and comfortable with the use of hexadecimal. Included in these are the fact that most debugging programs represent numeric values in the hexadecimal system, and also the environment in which you will be working in the second term (assembly language programming) is almost entirely reliant on the hexadecimal system.

Table 2.3

Decimal	Binary	Hexadecimal
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

2.5 Two State Devices

The following characteristics would characterize a theoretical ideal two state device: (a) turning ON or OFF would represent binary 1 and 0 logic levels, (b) an instant change from one state to the other, and (c) have infinite resistance when OFF and zero resistance when ON.

A manual switch will satisfy the requirements of (c), but is large, slow and suffers from contact bounce which means that it does not satisfy (b), due to the fact that voltage pulses will occur for a time after switching and could affect the accuracy of digital circuits.

An applied voltage can switch the types of transistor used in these devices very rapidly between states. A logic 1 voltage will turn the transistor ON and a logic 0 voltage will turn the transistor OFF. The advantage of this high speed switching is that it minimises the power dissipated during switching. Whilst in a stable 1 or 0 state the power dissipated is very small since either the current through, or the voltage across, the device is very small.

If a circuit employs *positive logic* a HIGH voltage indicates a logical 1 and a LOW voltage indicates a logical 0. Most digital circuits employ this system. In a *negative logic* circuit, the opposite is true – a HIGH voltage indicates logic 0 and a LOW voltage logic 1.

2.6 Digital Signals/Media

A binary digital signal takes the form of a stream of bits, and this stream may be either *unipolar* or *bipolar*. A unipolar signal uses only one polarity voltage, either positive or negative, and it switches between this and a nominally zero voltage, and a representation of a positive unipolar stream is shown in Figure 2.1. A bipolar signal switches between positive and negative voltages, as shown in Figure 2.2.

Fig. 2.1

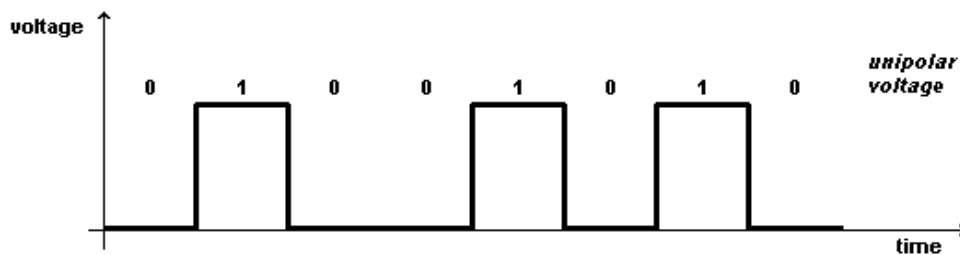
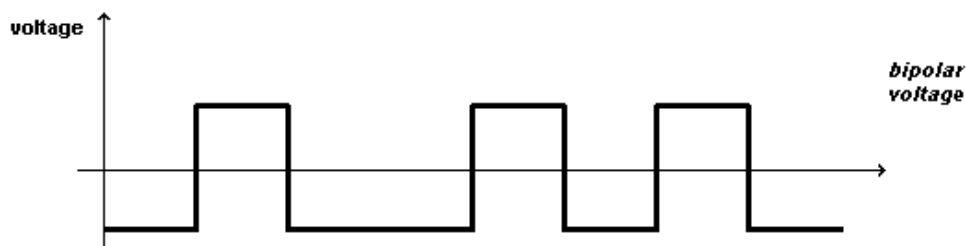
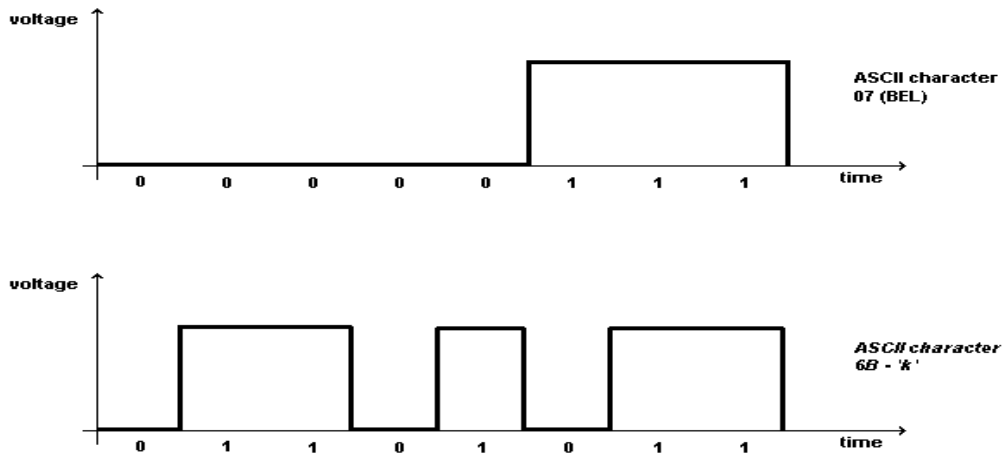


Fig 2.2



A digital signal consists of a number of logic 1s and 0s that represent numbers, letters, symbols and control characters. A combination of 8 bits (logical 1s, 0s) is a byte, which can represent one character in the *American Standard Code for Information Exchange* (ASCII) system. Note that the original ASCII standard uses 7 bits for the representation of the character; the eighth bit would often have been used as a parity bit to give a simple form of error checking. Two examples of ASCII data representation are shown in Figure 2.3.

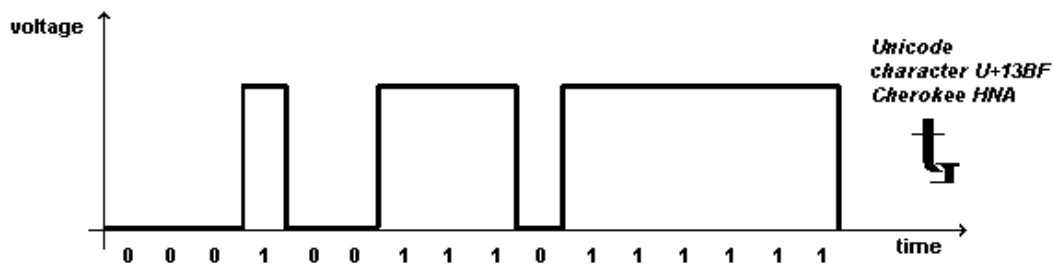
Fig. 2.3



The original ASCII system allows only for 127 symbols (2^7 or 7 bits). Extended ASCII, also known more correctly as ISO_8859-1 uses 8 bits, and can therefore represent 255 characters, and the 16 bit (2 byte) UNICODE UTF-8 system is now becoming more important, as many more languages need to be represented, some with hundreds or thousands of symbols (Han comprises 27484 characters!). The UNICODE UTF-8 representation of the Cherokee Letter HNA would be 0x13BF, which is shown in its binary form in Figure 2.4. UNICODE UTF-8 allows for the representation of 2^{16} or 65536 symbols, and UTF-16 allows for a further million symbols.

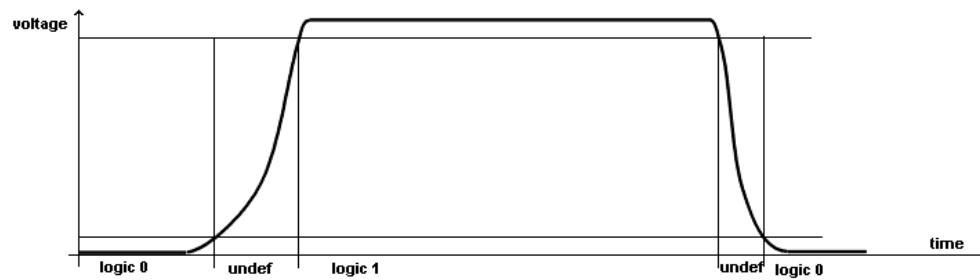
ASCII and Unicode are not the only data encoding schemes, and you will come across the *Gray Code*, and the *Manchester encoding* schemes.

Fig. 2.4



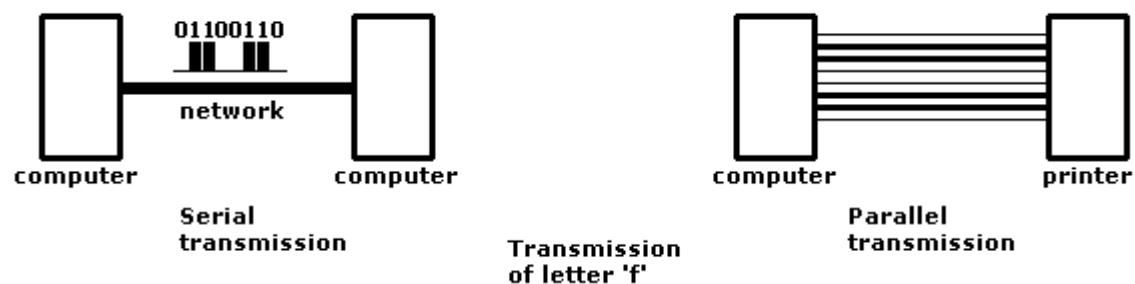
The examples shown are theoretical, in that they show instantaneous rise and fall of the voltage levels. In practise, a digital waveform cannot do this and the actual rise and fall of the voltage is as shown in Figure 2.5. This has important implications when we come to discuss circuits that operate by being 'clocked'. Manufacturers give details of the voltages at which their equipment operates (e.g. a recent AMD processor gives a logical 0 between -0.5v and $+0.5\text{v}$, and a logical 1 between $+1.65\text{v}$ and $+1.85\text{v}$). Any voltage outside these ranges gives a result of 'undefined' – neither a logical 1 nor a logical 0. So from this it can be seen that for a short period during each voltage transition, the logical state cannot be determined.

Fig. 2.5



In addition to this factor of how data is represented, how it may be transmitted can also be considered. Data can be transmitted either by serial or parallel transmission. In serial transmission a single carrier is used (wire, fibre-optic, electromagnetic waves) and one bit at a time transmitted. This is suitable for transmission over great distances, and is relatively cheap. In parallel transmission, many conductors are used (a *bus*). This allows a number of bits to be transmitted simultaneously, one per conductor. This is suitable for transmission over short distances, e.g. a local printer, from motherboard to hard disk, or internally between the components of the processor. It is more expensive, but many times faster, as typically 8, 16, 32 or 64 bits are transmitted simultaneously. This is shown in diagrammatic form in Figure 2.6.

Fig. 2.6



2.7 The Digital Computer and Microprocessor

A typical digital computer of today is a 'stored program' or 'Von Neuman' machine that stores the *program* and loads it into computer memory when it is required to be run. It will take input data, either from some peripheral device or from its own internal storage, act upon that data, and then output results either to internal storage of some sort, or to an external device such as a printer or display unit. A program is a set of instructions that tells the computer what to do.

The digital computer has become an indispensable part of day-to-day operation for everything except the smallest organisation; Government departments, commercial companies like banks and insurance companies, industrial firms in all branches of science and engineering and even small companies all use the computer as a normal part of their business dealings. The calculation of wages, taxes, pensions, bills and accounts; stock control, the storage of medical, scientific and engineering data; the rapid booking of theatre tickets, holidays and aircraft seats are all performed by computers. Rapid and complex scientific and engineering calculations, controlling engineering processes, telephone exchanges and military equipment and weapons, modelling weather systems and many other purposes are easily answered by the use of digital computers.

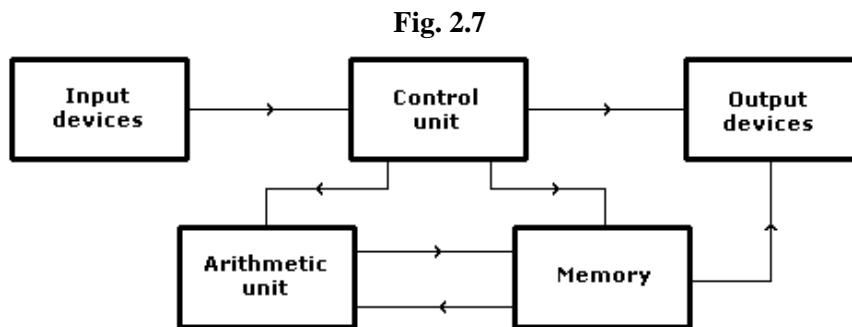
Recent figures show that PC ownership in some European countries is over 65% of households; the UK lags somewhat at 55%. Nevertheless, this is an indication of the extent

to which access to digital computers has grown. Even those people who do not own a computer cannot have escaped using the cash dispenser that is mounted in the wall outside banks and building societies. The ATM is linked by a line to a mainframe computer, and when a cash request is made the computer will check the account of the cardholder and either authorise the machine to pay out the cash, refuse the request or retain the card.

A block diagram of a digital computer is shown in Figure 2.7. Most of the memory is provided internally to the computer, either as ICs such as RAM or ROM or on hard disks, and sometimes it is external to the computer such as DAT tape, or optical disk such as CD or DVD.

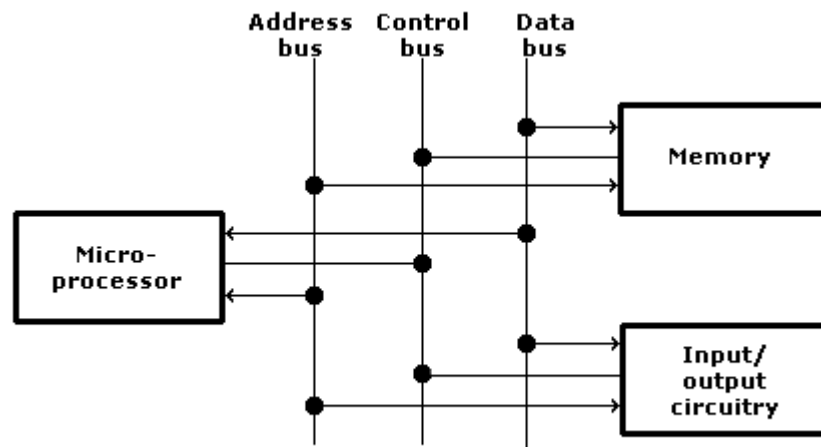
A computer program contains a series of instructions that the internal circuitry of that type of machine can understand. These instructions are taken in by the control unit, and as a result data will be moved to or from memory, or various calculations will take place. The resulting data may be returned to memory, or it may be retained in *registers* within the processing unit. Later in this unit you will learn how a register is constructed, and what actions and calculations the various types of register can perform.

When a calculation has been completed the control unit will transfer the results either to a memory device or to an output device, which could be a printer, a VDU or a network interface. The most usual input devices used to feed information into the computer are keyboard, disk drive or network interface, but could now include such items as CCDs, virtual reality gloves or even more esoteric devices.



A microprocessor is an integrated circuit that is built into, and is able to control the operation of a piece of equipment. A wide variety of equipment is controlled in this way now. Figure 2.8 shows the basic block diagram of a microprocessor system; the microprocessor contains various registers, an arithmetic unit and control circuitry. The memory and the input/output interface circuits are also provided by integrated circuits.

Fig. 2.8



2.8 The Internet

The Internet is a series of networks of computers linked by devices such as routers so that easy communication between persons or organisations is possible, no matter where they are located. Any organisation connected to the Internet has access not only to a vast fund of informative documents; increasingly it is also becoming a very strong part of business.

The Internet is now becoming so powerful, reliable and fast that it is possible to replace standard services such as the telephone and FAX, by services such as 'Voice over IP', which allows exactly the same person to person speech contact as a normal telephone call, but is actually using the facilities of the Internet to send and receive the information.

The Internet has three main parts: (a) Electronic Mail, (b) Newsgroups, and (c) the World Wide Web. There are also many other facilities that many people use on a daily basis; chat rooms, where people can send messages to one another in 'real time', and peer to peer file sharing are just two of the facilities that are available.

Electronic mail (e-mail) is now so commonplace as to hardly need describing here. A person wishing to send e-mail types the message into their computer, specifies the e-mail address of the recipient(s), and then sends the message onto the Internet. The message will be deposited rapidly into the mailbox of the recipient. E-mail does not require an immediate response, but can be read or re-read, and responded to at the time of the recipient's choosing, and in that respect resembles a letter sent by normal mail, but with the advantage of almost instant transmission.

E-mail can be on-line or off-line. With off-line operation, messages can be written and will remain stored on the local computer until a connection to the Internet is established and the instruction to send the messages is given; similarly, inbound messages are held on an Internet server belonging to the subscriber's supplier until they are requested to be downloaded. With on-line operation, messages are typed and transmitted using browser software such as Internet Explorer or Netscape. A familiar version of this is the Hotmail service.

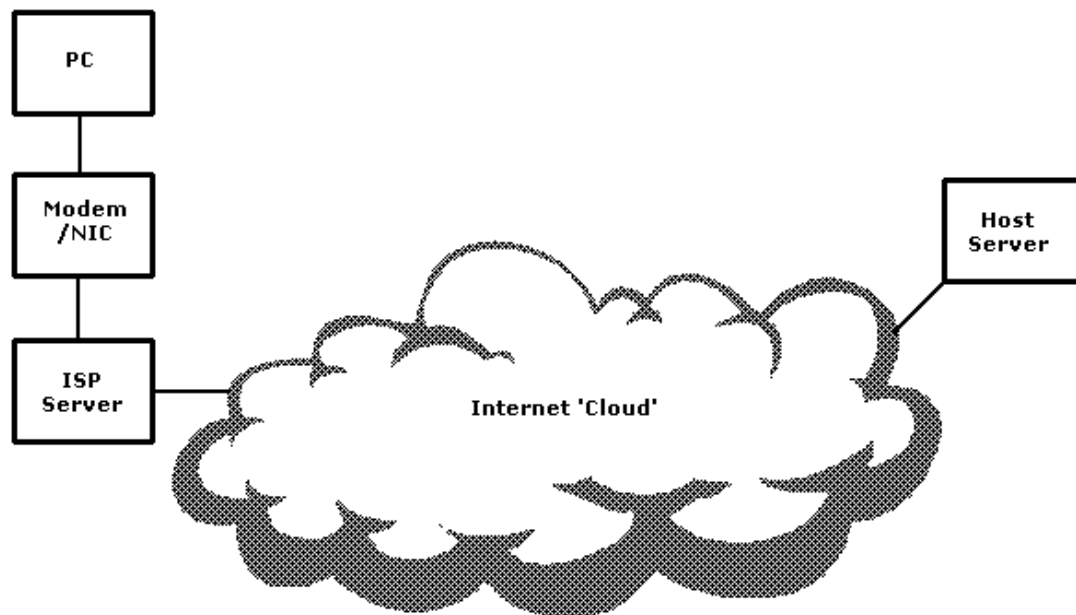
The address format of an e-mail is always [name]@[domain], e.g. bill.gates@microsoft.com, such that there must always be a '@' sign and there must always be a full stop in the part after the '@'.

Newsgroups (also known as *Usenet*) is an electronic bulletin board system where people may exchange views and seek or present information for others. All submissions are visible to everyone who subscribes to that discussion area, as are any replies. World wide there are now many thousands of newsgroups ranging from mainstream to bizarre.

With browser software one can obtain pages of information from the World Wide Web; many millions of pages are now freely available using *Uniform Resource Identifiers* (URIs). A URI consists of a short string such as 'http://dec.bmth.ac.uk/staff/swelsh/index.html'. This allows a particular page to be identified and retrieved using the correct protocol (in this case the Hypertext Transfer Protocol – 'http:') from anywhere in the world. The hypertext mark-up language (*html*) allows links to other pages, and other URIs to be embedded into web pages. Many sites are now using scripting systems that allow web pages to be built and sent dynamically, so that only the most current information is seen.

The basic block diagram of the Internet is shown in figure 2.9. A 'backbone' of large servers connected by very high-speed data links exists now in most countries of the world. These servers operate for 24 hours a day 7 days a week (24/7). To access the Internet some form of connection to these servers is needed. This can vary through a modem connection via the public telephone system, a broadband ADSL connection to a permanent fibre-optic or microwave link connection such as is used at the University.

Fig. 2.9



The idea of the 'Internet cloud' may seem somewhat strange, but its basis is in the fact that there may be many different routes from the ISP server at one end and the destination host server at the other. Indeed, this is the whole idea of the Internet, having been originally designed as a military concept that would enable communications to continue even in the event of a nuclear attack. The sad events of September 11th proved how effective this has been. In many cases on the day of the attack, the only method of communication that continued to work in some areas was e-mail.

2.8.1 Digital Equipment and Systems

Hand calculators that carry out the basic mathematical functions are now extremely cheap and widespread: many are able to carry out more advance mathematical calculations, and some are programmable. It is only the advent of Integrated Circuits allowing the complex circuitry needed that has made these devices practicable.

EPoS (Electronic Point of Sale) equipment is common, and many supermarkets have linked bar-code scanners, printers, weighing scales and cash registers that allow customers to have a complete printout of their purchases including weights of commodities, the amount tendered, change given, method of payment etc. These systems are also linked to central computers that control stock levels, reordering and delivery to the supermarket on a 'just-in-time' basis.

In an engineering context, digital readouts are more convenient than analogue readings. Whilst they may not necessarily be any more accurate, there is less possibility of error when recording a digital reading than when trying to interpolate an analogue scale. In the field of transport, computers can control everything from monitoring the movement of a single vehicle or consignment, to controlling traffic flows over an entire city.

Telephone systems are becoming increasingly digital. In this country, the transmission of voice from your telephone to the local exchange is still analogue based. The introduction of ISDN brings a complete digital service. Voice is translated into digital data using *Pulse Code Modulation* and voice, fax, images and any kind of data may be sent across the same network.

2.9 Exercises

- For each of the following state whether it is analogue or digital:
 - The number of pages on the Internet
 - The humidity in a greenhouse
 - The channel selector on a video recorder
 - The milk left in a milk bottle
 - A mobile phone
 - The weight of an elephant
 - The indicator on a garage petrol pump
- A digital signal voltage varies between the values +9 V and +0.2 V. Is this:
 - A unipolar or a bipolar signal?
 - If the +9 V signal represents logic 1, is this an example of positive or negative logic?
- Discuss the reasons why a digital computer employs digital electronic circuitry and not analogue circuits. Give a reason why binary digital circuitry is employed and not digital decimal.
- Which of the following can be classified as being analogue quantities?
 - The thickness of the polar ice-caps
 - The number of milliseconds in a second
 - Octal numbers
 - A digital watch
 - A light switch
 - A dimmer switch
- Use the Internet to search for sources that you think might be of assistance to you for this unit. Be prepared to share what you have found with the rest of your seminar group during the seminar.

3 Combinational Logic

The logic gate is the basic building block from which most digital circuits are constructed. Most gates have a number of inputs, and produce a single output. The signals at the input and output terminals are either at a HIGH voltage level or at a LOW voltage level, representing the logical '1' or '0' conditions. Gates will produce one output level according to the combinations of input levels and the type of gate, and the opposite output when any other combination is present at the inputs. In this chapter you will be introduced to the various types of gates and the ways they can be interconnected to perform different logical functions. The types of gate we will be using are the AND, OR, NOT, NAND, NOR, XOR and XNOR. We will be using the International standard symbols for the gates, as per Digital Works, and we will assume positive logic, i.e. a higher voltage is a logical 1 and a lower voltage is a logical 0.

Truth tables list all the possible combinations of the inputs applied to a gate. As we are dealing with 2-state devices, each input can only be at the logic 1 level or the logic 0 level. The output of a logic gate defines its function, so it is normally labelled as 'F'. The number of combinations is always 2^n , where n is the number of inputs. They may be listed in the binary counting order, that is from 0000 to 1111, alternately they may be listed in the reverse order. This is a matter of personal preference, but you should make sure that whichever scheme you chose you are consistent in its use. From a completed truth table a *Boolean* expression can be created to represent the function of a logic circuit.

Combinational logic circuits are those whose output(s) are determined by the logical states of their existing input(s). This is in contrast to *sequential logic* circuits, whose output is set by both the present input *and* the previous output state of the circuit.

3.1 Basic Gates

3.1.1 The AND logical function.

The AND function can be demonstrated by the circuit in Figure 3.1. This shows a lamp connected in series with two switches S1 and S2 and a D.C. voltage supply. For a current to flow through the circuit and light the lamp, both switches must be closed. The operation of the circuit can be described by its truth table that is given in Table 3.1, where a 0 indicates the switch is open (or off) and a 1 indicates that the switch is closed (or on).

Fig. 3.1

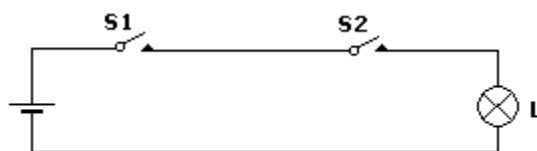


Table 3.1

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

The Boolean expression for the output F of a 2 input AND gate is given by the equation

$$F = A \bullet B$$

The dot \bullet is the Boolean symbol for the AND logical function but it is often omitted, hence this could have been written

$$F = AB$$

Table 3.2

A	B	C	F
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

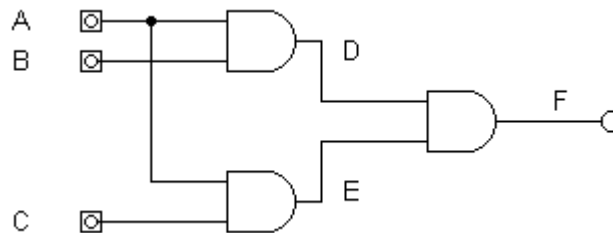
The output F of an AND gate with 3 inputs A, B and C is

$$F = A \bullet B \bullet C$$

The truth table of a 3-input AND gate is given in Table 3.2 and shows that the output is at logical 1 only when A *and* B *and* C are at 1. Always remember that the number of combinations of the input variables is 2^n , where n is the number of variables. So a 4 input gate requires 2^4 (16) rows in the truth table.

Example 3.1

Figure 3.2 shows a digital circuit constructed using AND gates. Write down the truth table for the circuit and use this to simplify the circuit.

Fig. 3.2

The truth table for the circuit shown is given in Table 3.3. The final column for this truth table ($F = D \bullet E = A \bullet B \bullet C$) is the same as the final column in Table 3.2 for a three input AND gate. Thus, the circuit shown in Figure 3.2 can be replaced by a single 3-input AND gate. This is the first indication that it is often possible to simplify a logic circuit.

Table 3.3

A	B	$D = A \bullet B$	C	$E = A \bullet C$	$F = D \bullet E = A \bullet B \bullet C$
0	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
1	0	0	0	0	0
1	0	0	1	1	0
1	1	1	0	0	0
1	1	1	1	1	1

3.1.2 The OR logical function

Current will flow in the circuit shown in Figure 3.3 if either switch S1 OR switch S2 OR both are closed, or ON. The only time current will not flow is when BOTH switches are open, or OFF.

An OR gate has two or more inputs and a single output. The output is LOW only when all the inputs are LOW. The truth table for a two-input OR gate is show in Table 3.4.

Fig. 3.3

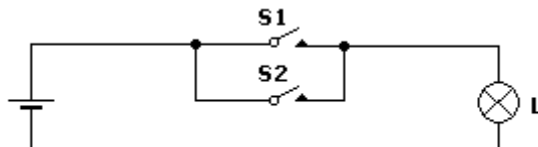


Table 3.4

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

The Boolean expression for the output F of a 2 input OR gate is given by the equation

$$F = A + B$$

The plus sign + is the Boolean symbol for the OR logical function.

Table 3.5

A	B	C	F
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

The output F of an OR gate with 3 inputs A, B and C is

$$F = A + B + C$$

The truth table of a 3-input OR gate is given in Table 3.5 and shows that the output is at logical 1 only when A or B or C are at 1.

Example 3.2

Write down the truth table for the circuit given in Figure 3.4 and from this show that the circuit can be obtained in a simpler manner.

Fig. 3.4

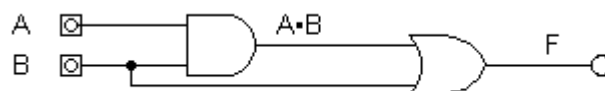


Table 3.6 shows the truth table for this circuit.

Table 3.6

A	B	$D = A \cdot B$	$F = D + B = A \cdot B + B$
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

The output F of the circuit is always the same as the input B , therefore no gates are required since input B may be directly connected to output F .

Example 3.3

Determine the logic circuit corresponding to the truth table given in Table 3.7

Table 3.7

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

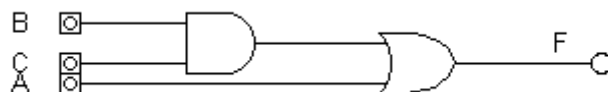
The output F is at logic 1 either if A is 1 OR if B AND C are both 1. This means that inputs B and C are connected to a 2-input AND gate, and the output from this, together with input A , are connected to a 2-input OR gate. The circuit is shown in Figure 3.5.

$$F = A + (B \cdot C)$$

Note that the AND has a higher priority than the OR so this expression could also be written:

$$F = A + BC$$

Fig. 3.5



Example 3.4

Write down the Boolean equations that describe the logic circuits given in figures 3.6 and 3.7. Write down the truth table for each circuit and compare them, and comment on the results.

Fig. 3.6

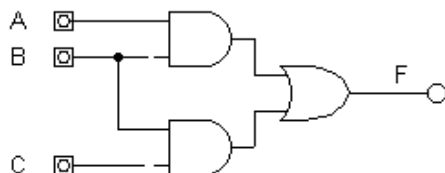


Fig. 3.7

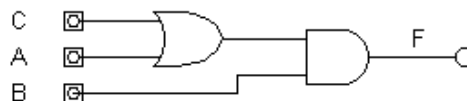


Table 3.8

A	B	$A \cdot B$	C	$B \cdot C$	F
0	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
1	0	0	0	0	0
1	0	0	1	0	0
1	1	1	0	0	1
1	1	1	1	1	1

$$F = A \cdot B + B \cdot C$$

Table 3.9

A	B	C	$A + C$	F
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

$$F = (A + C) \cdot B$$

Comparing the columns for F in the two truth tables, it can be seen that these two circuits give identical outputs. Thus, different combinations of gates can produce the same output. In this example circuit 3.7 uses one less gate than circuit 3.6.

3.1.3 NOT logical function

A NOT gate has a single input and a single output that is always the *inverse* of the input. If the input is at logic 1, the output will be at logic 0 and *vice versa*. The truth table of the NOT gate is given in table 3.10. The NOT gate is often known as an inverter and can be said to *complement* a digital signal. The complement of an 8 bit binary number is shown in Table 3.11.

Table 3.10

A	F
0	1
1	0

Table 3.11

Number	01001101
Complement	10110010

We denote the logical inverse of a value by placing a bar over the value. Hence \overline{A} is the logical inverse of A, and is pronounced as "A bar". As you will no doubt be aware the normal keyboard does not allow for the overbar and you have to use Word's Equation Editor to obtain this feature. Because of this, you will often find people placing a star after the name, as in A^* , to indicate the inverse. Both the overbar and the trailing bar are acceptable.

Example 3.5

Write down the truth table of the circuit shown in Figure 3.8, then write down the Boolean equation representing the circuit.

Fig. 3.8

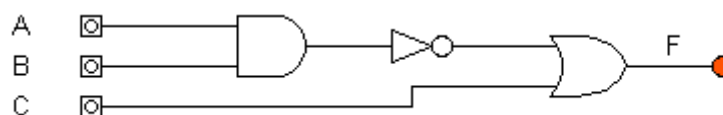


Table 3.12

A	B	$A \bullet B$	$\overline{A \bullet B}$	C	F
0	0	0	1	0	1
0	0	0	1	1	1
0	1	0	1	0	1
0	1	0	1	1	1
1	0	0	1	0	1
1	0	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1

The truth table for the circuit is given in Table 3.12. The output F is HIGH when the inputs A and B are both LOW OR if C is HIGH. Hence the Boolean expression for the circuit is:

$$F = \overline{A \bullet B} + C$$

This could also be written:

$$F = (A \bullet B)^* + C$$

3.1.4 The NAND logical function

The AND gate and the NOT gate in the previous example may be replaced by a single NAND gate, which performs the same logical function. That is, it performs the inverse of the AND function, so its output is LOW only when all its inputs are HIGH. The truth table for a 2-input NAND gate is given in Table 3.13 and that for a 3-input NAND gate in Table 3.14.

Table 3.13

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

Table 3.14

A	B	C	$A \bullet B \bullet C$	$\overline{A \bullet B \bullet C}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

The NAND gate can be manufactured more cheaply than other logic gates, as it requires fewer transistors in its construction. It is also possible to construct all other gates by using only NAND gates. In the real world therefore, many ICs are constructed using only NAND gates.

For instance, a NOT gate may be constructed by using a 2-input NAND gate with the 2 inputs connected together. Try this with Digital Works, and show that it is correct.

3.1.5 The NOR logical function

The output of a NOR gate is the same as that of an OR gate whose output has been inverted. The NOR logical function can be produced by an OR gate followed by a NOT gate, but more usually a NOR gate would be used. The truth tables for 2-input and 3-input NOR gates are given in Tables 3.15 and 3.16 respectively

Table 3.15

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

Table 3.16

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

3.1.6 The Exclusive-OR logical function

The exclusive-OR (or XOR, or occasionally EOR) gate has only two inputs. Its output is HIGH when one or other of its inputs is HIGH, but not when they are both HIGH or both LOW. It performs the logical function:

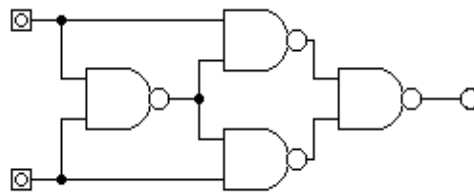
$$F = A \bullet \overline{B} + \overline{A} \bullet B = A \oplus B$$

Table 3.17

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

Its truth table is shown in Table 3.17.

The XOR gate can be made by combining various other types of gate, and one such combination of gates that gives the XOR logical behaviour is shown in Figure 3.9.

Fig 3.9

The XOR is frequently used to test if two inputs are different.

3.1.7 The exclusive-NOR logical function

The coincidence, or XNOR gate, has two inputs and one output. It produces a logical 1 only when both its inputs are the same, either HIGH or LOW. If the inputs are different, it produces a logical 0. The Boolean equation that describes an XNOR gate is therefore:

$$F = (A \bullet B) + (\overline{A} \bullet \overline{B})$$

This is the complement of the operation of the XOR gate.

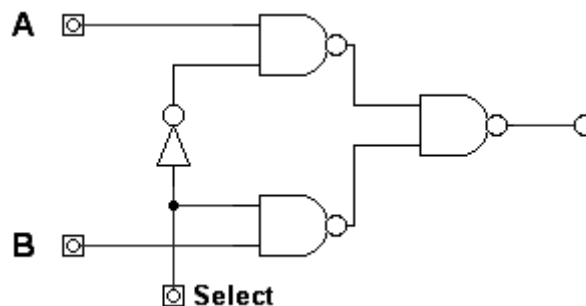
3.1.8 Gates with inverted inputs

All of the previously described logical functions can be performed by using different gates with the inputs inverted. For instance, if the inputs to an AND gate are inverted, it will perform the function of a NOR gate. Similarly, if the inputs to a NAND gate are inverted, it performs the function of an OR gate.

3.2 Multiplexers

A *multiplexer*, or *data selector*, is a circuit that selects any single input and transmits the data of that input to the output. An example of where a multiplexer might be used would be where there is a requirement to store time lapse images from a number of cameras onto a single storage medium. If two cameras tried to write to the medium simultaneously, the result would be garbage. Another example, particularly important in the IT world of today, is where several computers are trying to send data simultaneously across a single network cable. A simple 2-way multiplexer is shown in Figure 3.10. Build this circuit with Digital Works, and complete the truth table for it. Run the circuit and examine how it works.

Fig. 3.10



A *demultiplexer* performs the complementary function to a multiplexer. It has a single data input and a number of outputs. The outputs are selected by address, and if there are n address inputs this will support 2^n outputs.

A working example of an 8-to-1 multiplexer that can be used for converting an 8 bit parallel input to a single serial output will be made available to you.

We will discuss this circuit during the seminar. You should copy it to your H: drive, remove the 'read-only' attribute and run it using Digital Works, and attempt to understand what it is doing by looking at the logic history window when you run it. There is circuitry involved that you will not understand until we have covered Chapter 5 of this Lab Book. Concentrate on those aspects that you should be studying now, i.e. combinational logic circuits.

3.3 Encoders/Decoders

Encoders and *decoders* are examples of *code converters*, and in general terms they take an input of one type and give an output of another. For instance, a key press from a keyboard might be encoded as a 4 bit BCD value. A series of inputs might give a certain bit pattern that, when matched, causes a single output to go HIGH. This could be applied in practice to something such as a door security lock, where the lock is released upon receipt of the HIGH signal.

3.3.1 Encoders

In the context of this unit, an encoder is a digital circuit that converts from either decimal or hexadecimal into some other code. An encoder performs the opposite function to a decoder. A single input produces specific output data. The typical application of encoders is the keyboard, where a single key press must produce a unique binary code.

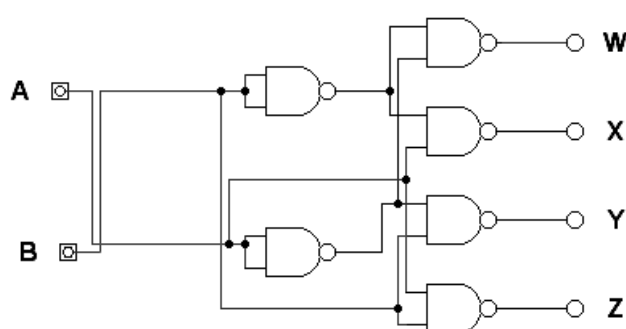
3.3.2 Decoders

A decoder is a circuit that detects a certain input, and gives a specific output to indicate that it has received that code. An example of a decoder that you might find useful is the BCD to 7 segment latch Decoder, found in Digital Works as a 74HC4511. When correctly wired this takes a binary input (the binary equivalent of the numbers 0-9) and for each one, the relevant outputs go HIGH to cause the illumination of the correct segments of the display to produce the familiar representations of the numbers 0-9.

Example 3.6

Figure 3.11 shows a circuit built from NAND gates. Draw up the truth table for all possible inputs of A and B. Use this to deduce the values of the outputs W, X, Y and Z. What does the circuit in Figure 3.11 do?

Fig. 3.11



Working back from the circuit:

$$\begin{aligned}
 W &= \overline{A \bullet B} & Y &= \overline{\overline{A} \bullet B} \\
 X &= \overline{A \bullet \overline{B}} & Z &= \overline{\overline{A} \bullet \overline{B}}
 \end{aligned}$$

giving the following truth table:

A	B	\overline{A}	\overline{B}	W	X	Y	Z
0	0	1	1	0	1	1	1
0	1	1	0	1	0	1	1
1	0	0	1	1	1	0	1
1	1	0	0	1	1	1	0

So, what exactly is this doing? If we consider that the first two columns in the truth table are the numbers from 0-3₁₀, and imagine that the outputs are lights numbered 0-3, then the light that is OFF will indicate the number that is selected. There is therefore in this circuit the possibility of an actual physical mapping between a binary number and some form of output.

3.4 Adders

A binary adder is a combinational circuit that is able to add together two binary numbers. The half-adder adds two inputs A and B to produce a *sum* and a *carry* but it is unable to take into account any carry from a previous stage. The full-adder also adds together two binary numbers, but can also take into account any previous carry. A circuit showing the gates of a four bit full-adder is shown as Figure 3.12, and the truth table for a one bit full-adder is given in Table 3.18.

Table 3.18

C_{in} (Carry-in)	B	A	S (Sum)	C_{out} (Carry-out)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

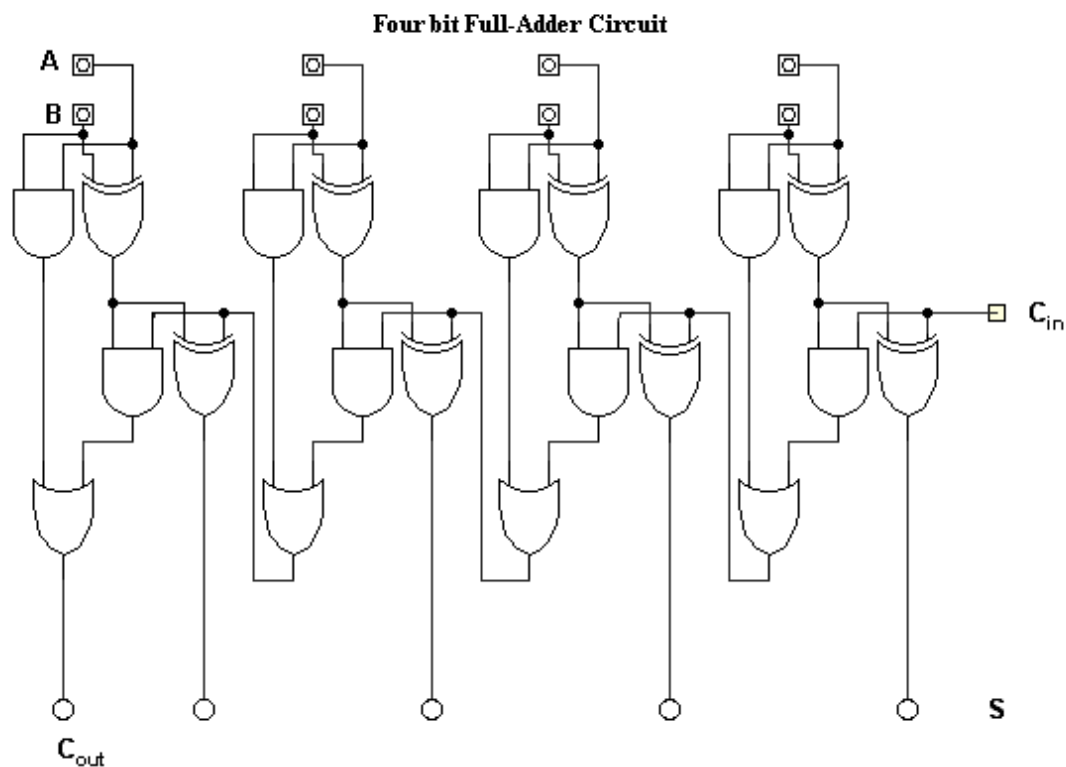
The Boolean equation describing the logical operation of the full-adder can be derived from the truth table. It is

$$\begin{aligned}
 S &= A \cdot \bar{B} \cdot \bar{C}_{in} + \bar{A} \cdot B \cdot \bar{C}_{in} + \bar{A} \cdot \bar{B} \cdot C_{in} + A \cdot B \cdot C_{in} \\
 &= (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C}_{in} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C_{in} \\
 &= (A \oplus B) \cdot \bar{C}_{in} + (\overline{A \oplus B}) \cdot C_{in} \\
 &= A \oplus B \oplus C_{in}
 \end{aligned}$$

$$\begin{aligned}
 C_{out} &= A \cdot B \cdot \bar{C}_{in} + A \cdot \bar{B} \cdot C_{in} + \bar{A} \cdot B \cdot C_{in} + A \cdot B \cdot C_{in} \\
 &= A \cdot B + (A \oplus B) \cdot C_{in}
 \end{aligned}$$



By connecting the carry-out of one four bit adder into the carry-in of another four bit adder an eight bit adder is produced, and this process can be repeated until an adder of the required size is produced. There are issues with this circuit, in that it can take some time for the carries to ripple through the circuit. In the real world, circuits would include fast look-ahead carry circuitry to overcome this problem.

Fig. 3.12

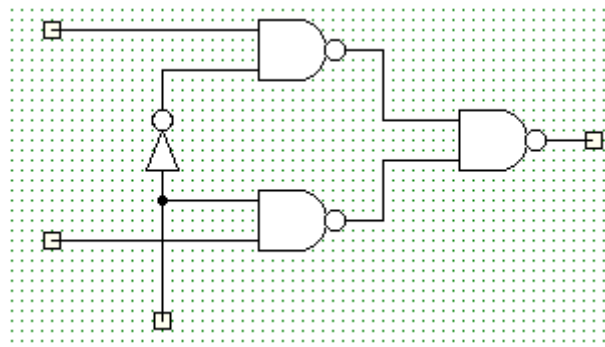


3.5 Macros in Digital Works

There was a brief introduction to macros in chapter 1, when we looked ‘inside’ some more complex components. Having seen the four bit full-adder circuit above, this is a good point to introducing you to the creation of your own macros. You will gain marks in your first assignment by the use of neat, well-ordered and well labelled macros.

Start by creating the simple multiplexer in Figure 3.10, but do not use the interactive inputs or the LED output. Use macro tags instead. The macro tag is shown  as with a context  pop-up of

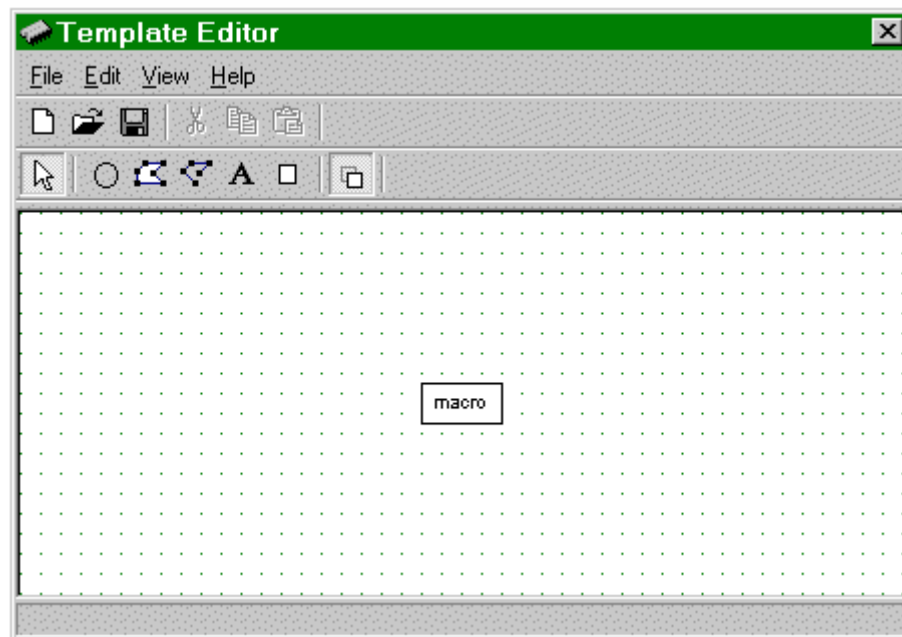
This will give you a circuit like this:




Now right-click one of the macro tags. This will bring up a context menu:



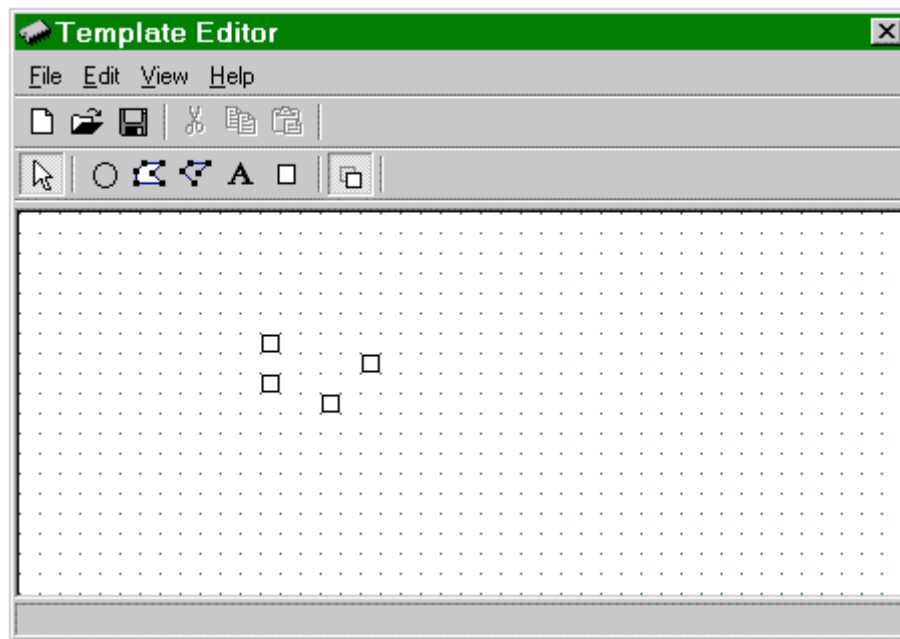
Select the 'Template Editor' option. This will bring up a new window containing a canvas like this (the actual contents may vary, according to whether you have used this feature before):



You may choose to manipulate the objects that are already there, or start with a clean blank canvas (the writer prefers to start with a new canvas). On the tools menu bar you will see the 'pin' icon  with a context pop-up that just says 'Pin'.

Place the relevant number of pins onto the canvas. You should take care at this stage to ensure accurate layout and correct numbers of pins. From experience it is very easy to get this stage of the process wrong. If this happens you will end up with a macro that is useless – mainly because the external view of the macro does not match either the internal structure and layout of components or it will not connect into surrounding external components correctly. Be aware of this because it is probable that macros created for your assignment may have thirty or forty pin connections.

For our simple circuit this will look like:



Now right click on the pin *that is to represent the macro tag that was clicked at the start of this procedure*. It is most important to ensure that you do these steps in a designed and methodical manner to avoid problems. This will again give a context menu:

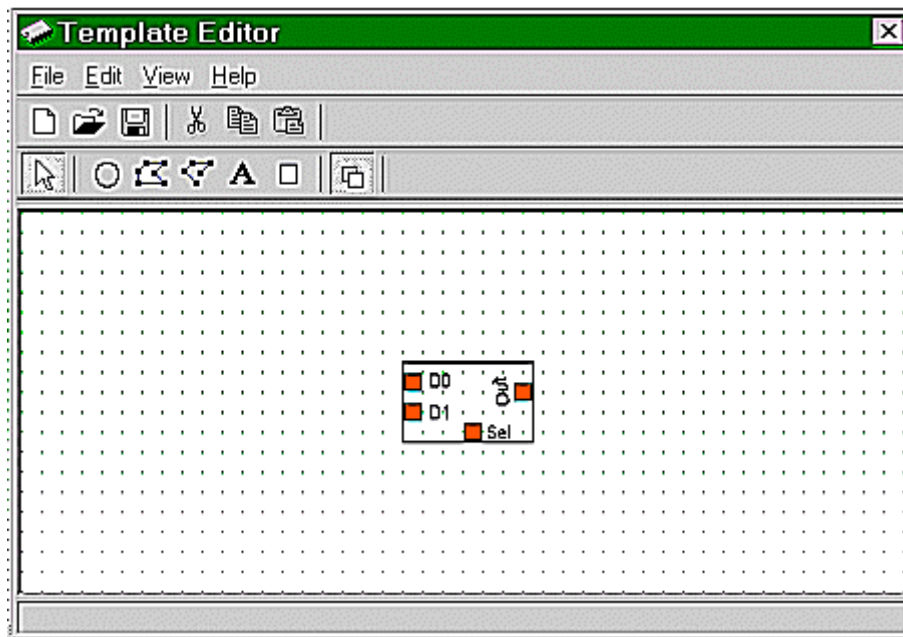


Select the 'Associate With Tag' item. Notice that the pin has now turned yellow, showing that an association has been made.

Next, close the template editor window and return to the Digital Works canvas. Select the next macro tab to associate, and follow the steps through again. Notice that when the template editor opens, already associated pins are now coloured red. Notice also that the macro tags become numbered on the Digital Works canvas as each one is associated. The numbering may be formatted using the right-click context menu and selecting the 'Tag Text Style' tool.

The final steps are firstly to neaten up and label the macro, and then to save it as a template file. Digital Works attempts to save templates into the default parts centre location. If you do not have write permission to this folder you will need to save the template into your own data area.

The final result will look like this:



You may now use this macro in exactly the same way as any of the supplied ones, by opening the location of the template and dragging a copy onto the main Digital Works canvas. It may be wired up in the normal manner and will behave exactly as the circuit that it contains.

Using this facility makes it possible to have numbers of components containing many gates/inputs/outputs/etc, all within a small area of the main canvas.

3.6 Fan-In and Fan-Out

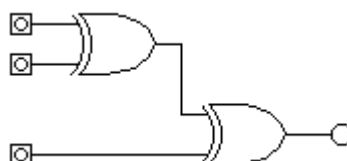
Fan-in and fan-out are terms used to indicate the number of inputs that are either fed into (*fan-in*) or fed from (*fan-out*) a single gate.

We can see fan-in quite simply by using Digital Works and changing the number of inputs that a gate has. Digital Works only allows a maximum fan-in of 4 – i.e. 4 inputs into a single gate. In practice, industry builds gates with a much greater fan-in than this.

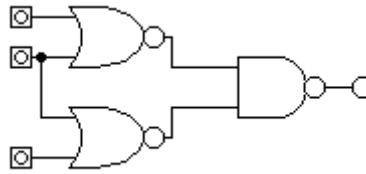
Fan-out is where the output of a gate has to drive a number of gates or sub-circuits. There is a physical limit upon fan-out because of the fact that the power of the output is divided amongst the subsequent inputs/sub-circuits. The response times of the 'downstream' gates increase as fan-out increases, and there is also a voltage drop-off to the succeeding devices, and a point is reached at which the circuit will no longer be reliable.

3.7 Exercises

1. Obtain the truth table for the following circuit:



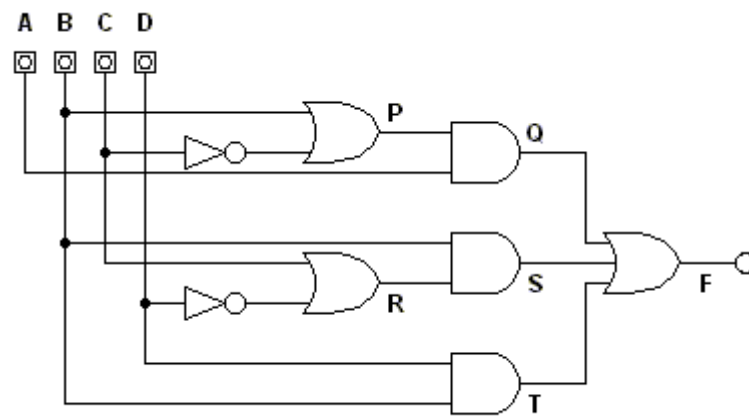
2. Obtain the truth table and the Boolean equation for the following circuit:



Suggest a replacement for this arrangement of gates.

3. A 2-input NOR gate has inputs \bar{A} and \bar{B}
 (a) use a truth table to determine the function performed
 (b) repeat the exercise for a 2-input NAND gate.
4. Inputs A and B are applied to an AND gate. The output from this gate is applied to one input of an OR gate, and input C is applied to the other input of the OR gate. What is the output of the OR gate?
5. Show how (a) an XOR gate and (b) an XNOR gate can be used as an inverter.
6. Inputs A and B are applied to a NOR gate. Inputs B and C are applied to a second NOR gate. The outputs of the two NOR gates are then applied to a NAND gate. Determine the Boolean equation of the output F of the circuit.
7. Draw the block diagram of a 3-to-8 line decoder and, with the aid of a truth table, explain its operation.
 (a) if the input signal is 101, which output line is selected?
 (b) if output line 7 is to be selected, what should be the input signal?
8. Draw logic diagrams using AND, OR, and NOT gates only, to implement the following Boolean expressions. Do not simplify these expressions, but draw the circuits directly.
 (a) $F = \bar{A} \bullet B + A \bullet \bar{B}$
 (b) $F = (A + B + C) \bullet (A \bullet B + A \bullet C)$
 (c) $F = (A + \bar{C}) \bullet (A + B \bullet \bar{D})$
 (d) $F = \bar{A} + \bar{C} \bullet A + B \bullet \bar{D}$
9. Tabulate the values of the variables P, Q, R, S, T and F for all possible combinations of inputs A, B, C and D in the circuit given in Figure 3.13. Your results should be in the form of the truth table:
- | A | B | C | D | $P = B + \bar{C}$ | $Q = P \bullet A$ | $R = C + \bar{D}$ | $S = B \bullet R$ | $T = B \bullet D$ | $F = Q + S + T$ |
|---|---|---|---|-------------------|-------------------|-------------------|-------------------|-------------------|-----------------|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . |
| 1 | 1 | 1 | 1 | . | . | . | . | . | . |
10. For the circuit in Figure 3.13 obtain a Boolean expression for the output F in terms of the inputs A, B, C and D.

Fig. 3.13



4 Karnaugh Map

The Karnaugh map is a simple, elegant, diagrammatic method for simplifying sum-of-product equations. If you use Boolean algebra, it is not easy to see if you have reached the simplest solution. With the Karnaugh map (K-map), it is easy to find the simplification, and address other, hardware specific issues. The function to be simplified is shown as a set of squares or *cells*, of which each cell maps to one term of the equation. There are 2^n cells in a map, where n is the number of variables in the equation to be simplified.

We will be dealing with K-maps of up to four variables, and K-maps are suitable for equations of up to five inputs. There are other, more complex, methods for simplifying equations with more than four inputs.

4.1 Sum-of-Product Equations

A sum-of-product equation involves analysing a truth table to identify all the terms where the truth table gives a TRUE, or logical 1, result.

4.2 2-Variable K-Maps

If we think of the equation

$$F = A\bar{B} + \bar{A}B$$

There are two inputs, or variables (A and B). We will therefore need a grid containing 2^2 cells = 4 cells. Label the rows and columns as shown in Table 4.1, such that every combination of A and B is represented.

Table 4.1

	A	
	\bar{A}	A
B	$\bar{A}\bar{B}$	$A\bar{B}$
B	$\bar{A}B$	AB

Considering the equation above, we now put '1's in the cells that are represented in the equation and '0's in the terms not represented, thus:

	A	
	\bar{A}	A
B	0	1
B	1	0

Now consider the equation:

$$F = AB + \bar{A}B + \bar{A}\bar{B}$$

Using the technique outlined above, enter the terms into the K-map:

	A	
	\bar{A}	A
B	1	0
B	1	1

To simplify an equation, adjacent squares containing a '1' are looped together, as shown on the right.

	A	
	\bar{A}	A
B	1	0
B	1	1

Looping adjacent squares in the map shown gives:

$$\begin{aligned} F &= \bar{A}(B + \bar{B}) + B(A + \bar{A}) \\ &= \bar{A} + B \end{aligned}$$

With a little practice the terms can be read, already simplified, directly from the K-map.

Example 4.1

Use a K-map to confirm the Boolean rule that:

$$\bar{A} + AB = \bar{A} + B$$

The mapping for the left hand side of the equation is shown opposite.

		A	
		\bar{A}	A
B	\bar{B}	1	0
	B	1	1

Looping the adjacent '1's gives two terms: \bar{A} (the vertical loop) or B (the horizontal loop). It is common practice to leave the '0' cells blank.

4.3 3-Variable K-Maps

The K-map is easily extended to deal with three variables (A, B and C). Using the 2^n rule, we now need 2^3 cells = 8. Table 4.2 shows the layout of the cells.

Table 4.2

		AB			
		$\bar{A}\bar{B}$	$A\bar{B}$	AB	$\bar{A}B$
C	\bar{C}	$\bar{A}\bar{B}\bar{C}$	$A\bar{B}\bar{C}$	$AB\bar{C}$	$\bar{A}B\bar{C}$
	C	$\bar{A}B\bar{C}$	$A\bar{B}C$	ABC	$\bar{A}BC$

Reiterating on how we get to the simplification:

- Extract the Sum-of-Products equation from the truth table.
- Map the terms of the Sum-of-Products equation into the K-map cells.
- Loop adjacent cells into groups of one, two, four or eight. The object is to include as many cells as possible in each loop, as this gives the greatest simplification. Overlapping of groups may occur, and any single terms not included in a group will have to be included in the final result, as all terms must be accounted for. It is important also to remember that the loops may 'wrap-around' to include terms on the opposite side of the table.
- Write down the simplified equation, by determining which variables are inside each loop.

Example 4.2

Use a K-map to simplify the expression

$$F = A(\overline{BC} + \overline{B}\overline{C}) + \overline{A}BC$$

(a) Expand the expression into its sum-of-products form:

$$F = ABC + \overline{A}B\overline{C} + \overline{A}BC$$

(b) Mapping gives:

C	AB			
	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
\overline{C}		1	1	
C		1		

(c) Looping the adjacent cells:

C	AB			
	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
\overline{C}		1	1	
C		1		

(d) The simplified expression is therefore

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

Example 4.3

Use a K-map to simplify

$$F = ABC + \overline{A}B\overline{C} + \overline{A}\overline{B}\overline{C}$$

The mapping is

C	AB			
	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
\overline{C}	1			1
C			1	

Notice the 'wrapped around' mapping!

From the looped cells

$$F = \overline{A}\overline{C} + ABC$$

4.4 4-Variable K-maps

When the equation contains 4 variables A,B,C and D we must use $2^4 = 16$ cells (or a 4 x 4 grid). The labelling of the grid is:

CD	AB			
	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}\overline{D}$	$\overline{A}\overline{B}\overline{C}\overline{D}$			
$\overline{C}D$	$\overline{A}\overline{B}C\overline{D}$			
$C\overline{D}$	$\overline{A}B\overline{C}\overline{D}$		$AB\overline{C}\overline{D}$	
CD	$\overline{A}B C\overline{D}$			

Example 4.4

Simplify the Boolean equation

$$F = ACD + AB\bar{C}\bar{D} + \bar{A}BD + \bar{A}\bar{B}\bar{C}D$$

The mapping of the equation is shown opposite.

The cells can be looped in two groups each containing four '1's. Hence the simplified equation is

$$F = AD + BD$$

CD \ AB	AB			
	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$				
$\bar{C}D$		1	1	1
CD		1	1	1
$C\bar{D}$				

Example 4.5

Use a K-map to simplify

$$F = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}D + \bar{A}BCD + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}\bar{C}D + ABCD$$

The mapping of the expression is shown opposite. The four '1's in the second column can be looped together to give the term $\bar{A}B$. The four corner cells can also be looped together to give $\bar{B}\bar{D}$. Hence, the simplified expression is:

$$F = \bar{A}B + \bar{B}\bar{D}$$

CD \ AB	AB			
	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$	1	1		1
$\bar{C}D$		1		
CD		1		
$C\bar{D}$	1	1		1

4.4.1 Inverse of a function

It is useful to know that it is possible to do the mapping using the '0's instead of the '1's, giving \bar{F} . Once the simplification has been completed, invert the result to obtain F.

4.4.2 Don't care/Not possible states

In some logic circuits it can occur that we either don't care what the result is, or it is not possible for certain circumstances to happen.

Take for example a system where there is an input A that is HIGH to indicate that (say) the pressure within a container is within a specified safety limit and there is also an input that goes HIGH when the pressure rises above the safety limit. These two inputs are mutually exclusive; that is they cannot both be HIGH and they cannot both be LOW.

Where this happens, we indicate this condition on our K-map with an 'X'. Now, when we do the simplification, we *include* in the loops any 'X's that assist in the simplification. It is *not* necessary to include all 'X's, only those that help us to make up groups of 2, 4, or 8.

Suppose the mapping of a function is

	AB			
	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}\overline{D}$	X	X	X	X
$\overline{C}D$		1	1	
CD		1	1	
$C\overline{D}$	1			1

Looping the '1' cells gives

$$F = BD + \overline{B}\overline{C}\overline{D}$$

If the 'don't care' cells are included then the equation simplifies to

$$F = BD + \overline{B}\overline{D}$$

Incidentally, if you find that in a four variable K-map all 16 cells contain either a '1' or an 'X', then the equation is TRUE and $F = 1$.

4.4.3 Further Simplification

There are some occasions when a K-map will produce a result that, whilst simplified, is not in its simplest form. It is advisable to check the final Boolean expression for this situation.

For instance, the straightforward result of a K-map simplification gives the result:

$$F = \overline{A}BD + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{C}\overline{D}$$

It is quite evident that this is not in its simplest form, and may be simplified further as:

$$F = \overline{A}(BD + \overline{B}\overline{C}\overline{D} + \overline{C}\overline{D})$$

4.5 Exercises

- Simplify the following using K-maps:
 - $\overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C} + ABC + \overline{A}BC$
 - $ABC + \overline{A}BC + \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}B\overline{C}$
- Simplify $A \oplus B \oplus C$
- Design a logic circuit whose output will go HIGH whenever the 4-bit binary input is odd.
- A voting system has four inputs A, B, C and D. If any three, or all four inputs are HIGH the output must be HIGH also. If the voting is split 2/2, then input D has the casting vote. If there is only one input that is HIGH, then the output must be LOW, and it must also be LOW when all four inputs are LOW.
- Design a logic circuit that inputs a BCD number and gives a HIGH output whenever the input number is odd and is less than 8.
- The equipment of a car includes an audible alarm that goes off if (a) either the headlights or the sidelights are ON AND the driver's door is open or (b) the ignition is switched ON and the driver's door is open. Draw up a truth table for the inputs and outputs. From this truth table form a sum-of-products equation. Use a K-map to simplify this equation and using Digital Works create a logic circuit for this function.
- A circuit has four inputs D, C, B and A encoded in 8421 natural binary form. The inputs in the range 0000 to 1011 represent the respective months from January to December. Inputs from 1100 to 1111 is undefined. The output of the circuit is HIGH if the month has 31 days, otherwise it should be LOW.

- (a) draw up a truth table for all the possible values of inputs. Remember to include the undefined or 'not possible' states.
- (b) obtain a sum of products expression for the function from the truth table
- (c) use a K-map to simplify the sum of products expression
- (d) construct a circuit to implement the simplified expression.

5 Sequential Logic/Flip-Flops

In some texts, the terms ‘latch’ and ‘flip-flop’ are used interchangeably, but strictly a latch refers to an ‘unclocked’ input whereas a flip-flop is ‘clocked’.

The fundamental difference between a *combinational* circuit and a *sequential* circuit is that the output of a combinational circuit depends *only* upon its present inputs. A sequential circuit, however, relies upon both its present and past inputs. The important thing that you need to know about a latch or flip-flop is that it ‘remembers’ what state it is in.

All latches/flip-flops have two outputs, normally labelled Q and \overline{Q} . The device cannot operate with both of these outputs in the same state; they will always be complementary.

We also introduce the concepts of *synchronous* and *asynchronous* circuits at this stage.

5.1 The R-S Latch

Using Digital Works make up the circuit shown in Figure 5.1. You will see that the inputs for the latch are labelled R and S. These stand for *reset* and *set*. The truth table for the R-S latch is given in Table 5.1. When you operate this circuit you will notice the following:

- When both R and S are LOW, the present output will remain unchanged
- If S goes HIGH whilst Q is LOW, Q will go HIGH
- If S goes HIGH whilst Q is HIGH the output will remain unchanged
- If R goes HIGH whilst Q is HIGH, Q will go LOW
- If R goes HIGH whilst Q is LOW, the output will remain unchanged
- If you attempt to set both R and S HIGH, an error will be displayed indicating a *race condition*.

Do not forget that \overline{Q} will always be the complement of Q.

Fig. 5.1

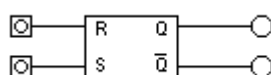


Table 5.1

R	S	Q	\overline{Q}	Comment
0	0	Q	\overline{Q}	Hold State
0	1	1	0	Set
1	0	0	1	Reset
1	1	X	X	Avoid condition

You can build an R-S flip-flop from either NOR gates (Figure 5.2) or NAND gates (Figure 5.3).

Fig 5.2

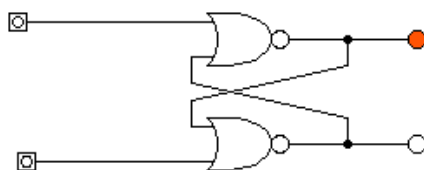
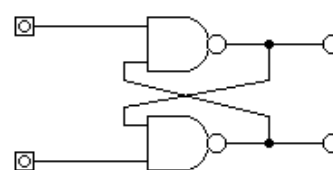


Fig 5.3



Effectively, an R-S flip-flop can be regarded as the simplest possible memory element – it ‘remembers’ a single binary digit (bit).

5.2 The D-Type Flip-Flop

The D-Type flip-flop overcomes some of the problems of the R-S flip-flop of having to ensure that both R and S cannot be HIGH at the same time. It has only one input, and internally to the flip-flop this input is complemented, thus ensuring that whatever state the input is in, its complement is always available to the other gate in the flip-flop.

Example 5.1

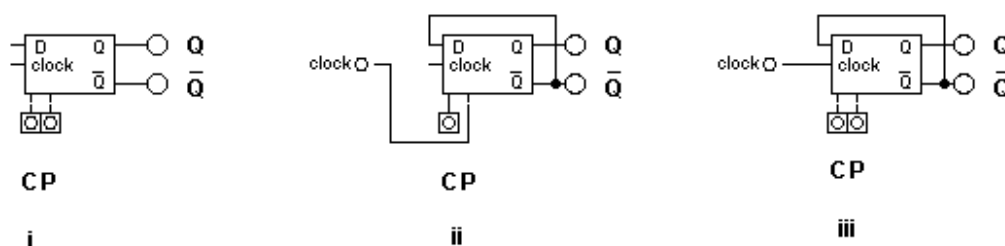
Create the circuit shown in Figure 5.4(i) using Digital Works. You will notice several inputs that the R-S flip-flop does not have. If you use the inputs that are marked ‘P’ and ‘C’ (‘Preset’ and ‘Clear’) you will notice that this flip-flop behaves almost as an R-S flip-flop. However, from the truth table in Table 5.2 you can see that there are two important differences. Firstly, when both P and R are HIGH, Q is HIGH and there is no ‘race condition’. Secondly, when R is HIGH, Q will take the same state as P, and \bar{Q} will be its complement.

Table 5.2

R	P	Q	\bar{Q}	Comment
0	0	Q	\bar{Q}	Hold State
0	1	1	0	Set
1	0	0	1	$Q=P$ and $\bar{Q} = \bar{P}$
1	1	1	0	$Q=P$ and $\bar{Q} = \bar{P}$

Now create the circuits shown in Figures 5.4(ii) and 5.4(iii). Run these circuits and explore the effect of the Reset input. What do you notice about the rate of the output? The circuit in 5.4(iii) is a *divide by 2* circuit; its output is half the frequency of its input.

Fig. 5.4



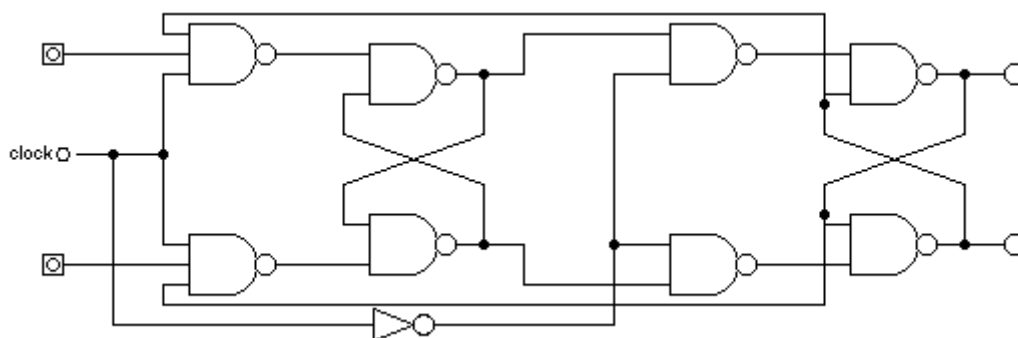
We have now introduced an important concept – that of the *synchronous* circuit. The important thing about a synchronous circuit is that it operates in response to a regular input, in the case of the circuits we have built with Digital Works this is the clock. Most of the communication of a computer with the outside world is *asynchronous*. For instance, when we type into the keyboard this is at comparatively irregular and indeterminate intervals. Most of the internal communication within the computer is synchronous, being governed by the internal clocks of the computer. At the time of writing 3.2GHz (Gigahertz) clock speed processors for PCs have made an appearance. The ‘Giga’ prefix indicates 10^9 or 1,000,000,000 in decimal numbers, or 2^{30} i.e. 1,073,741,824 in the binary system.

Now open a new canvas in Digital Works and click on 'Parts Centre'. Choose 'Macros' and import a 4-bit register. Using the context menu, open up the register. You will notice that this 4-bit register is made from D-Type flip-flops. We will cover various types of register in the next Chapter, but you should be aware that this macro will be of assistance for your first Systems Architecture Assignment.

5.3 The J-K Flip-Flop

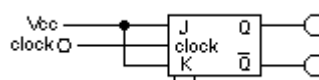
The third type of flip-flop is the J-K flip-flop (named after Jack Kilby of Texas Instruments). It has 5 inputs and the usual two outputs. The S input of the R-S latch becomes the J input of the J-K flip-flop, while the R input becomes the K. There are also a clock input, unconditional Preset and Clear inputs. The internal circuitry is such that, once an input has gone HIGH, it is disabled. This means that, externally, both J and K inputs can be held HIGH at the same time, overcoming the race-condition situation of the R-S latch. The basic logic circuit of the J-K flip-flop is given in Figure 5.5.

Fig. 5.5



One of the most important aspects of this behaviour is that the J-K flip-flop can be made to toggle, by holding both J and K HIGH. Create and run the circuit shown in Figure 5.6 to illustrate this behaviour.

Fig. 5.6



We will see why this is important when we come to study Section 6.5 – Counters.

5.4 Clocking Types

Looking back to Chapter 2, recall that Figure 2.5 shows the actual waveform of the voltage change in a two state device. This has an important bearing on the subject of clocking types that we will discuss.

There are three clocking modes:

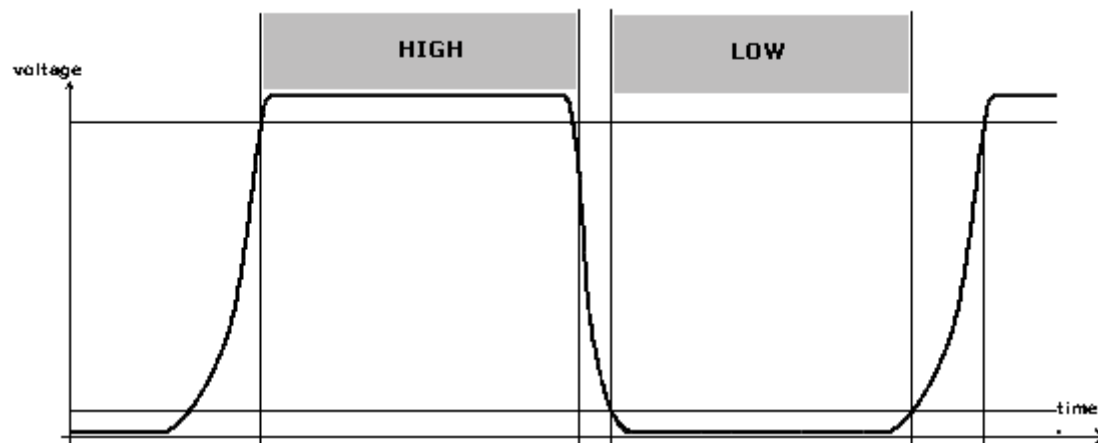
1. Level sensitive
2. Rising edge/ Falling edge
3. Master/Slave

5.4.1 Level sensitive

In a level sensitive device, any change to the input that requires a change to the output can cause that change at any time whilst the device is HIGH (or LOW if the circuit is active LOW, of course). This clocking mode has inherent problems associated with it as regards any circuit that relies on feedback.

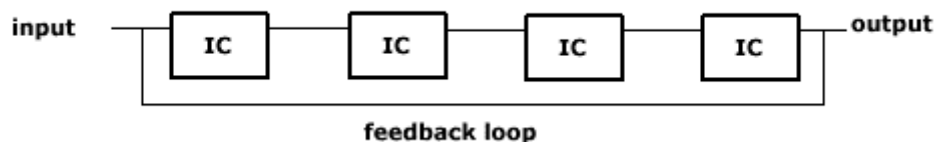
Firstly, look at the timing diagram Figure 5.7. The time span during which the input/output may be affected is shown by the grey box marked 'HIGH'.

Fig. 5.7



Now imagine a notional circuit that involves feedback, as depicted in Figure 5.8.

Fig. 5.8



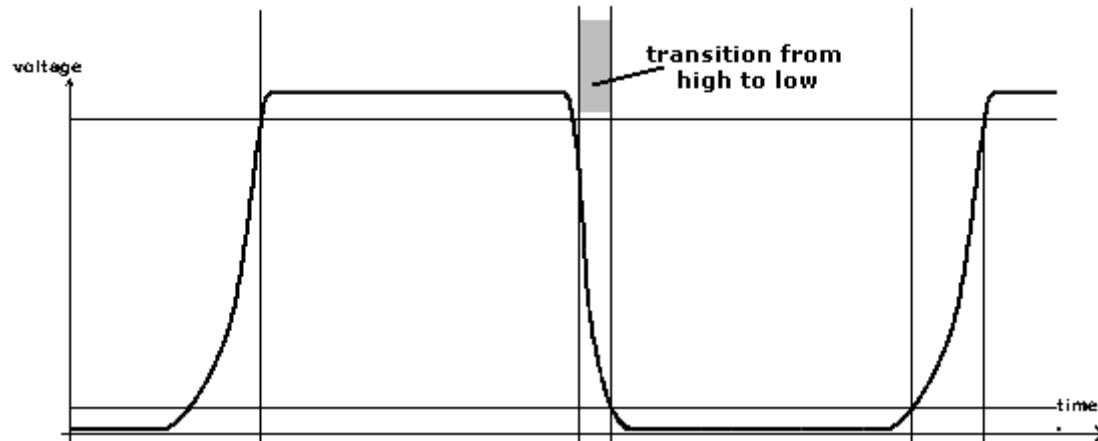
What will happen to the output of this circuit if, *during the time that the input clock is HIGH*, the feedback loop causes the input to change? The change will cascade through the system, and back to the input again, causing another change. This can be pictured as 'a dog chasing its tail'. The situation is even worse than that, because imagine now that one of the intervening ICs changes state as a result. The feedback loop, and the whole circuit, may now be at a totally different than expected.

Thus in a circuit of this type, ambiguous and unexpected results can occur from a level sensitive input device.

5.4.2 Rising edge/Falling edge

If we look at Figure 5.9, this illustrates the principle of a falling edge triggered device.

Fig. 5.9



As you can see from the area indicated in grey, the transition time from HIGH to LOW is very short in comparison to the total clock cycle. The diagram is an illustration of the time scale of these events; in reality the manufacturers will state a voltage at which the transition takes place, but they will also indicate an actual time for the transition. This time depends upon voltage, which itself can depend upon fan-out. Typical times could be of the order of <1 ns at 5 volts, to 6 or 7 ns at 1.75 volts.

Going back to our notional circuit in Figure 5.8, we can see that the chances of feedback affecting the input subsequently are virtually eliminated.

A rising edge triggered device operates upon exactly the same principle as the falling edge device, except that the change takes place during the transition from LOW to HIGH.

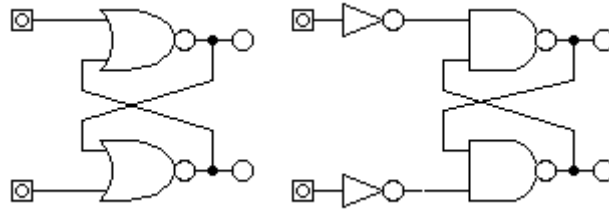
5.4.3 Master/Slave Flip-Flop

The principle of the master/slave flip-flop is that it is essentially two R-S flip-flops arranged in such a way that the master is clocked upon the rising edge of the clock cycle, and the logical state resulting from the input is then passed to the slave on the falling edge of the clock cycle. The output is therefor guaranteed to be stable when the circuit is 'clocked'.

5.5 Exercises

- What are the stable states of a flip-flop?
 - What is the difference between a latch and a flip-flop?
 - Outline the difference between a master-slave and an edge triggered J-K flip-flop.
- What is a sequential circuit, and how does it differ from a combinational circuit?
- Explain why it is necessary to use flip-flops in sequential circuits as opposed to latches?
- What are the three basic flip-flop clocking modes, and why is it necessary to have so many?

5. Demonstrate that the flip-flops shown below are equivalent. Are they exactly equivalent?



6. Many flip-flops have unconditional preset and clear inputs. What do these inputs do, and why are they required in sequential circuits?
7. Write down the truth tables for the D-type and J-K flip-flops. From this, determine how a J-K flip-flop can be made to behave like a D-type flip-flop.

6 Useful Circuits

6.1 Memory

Digital systems almost always include memory elements for the storage of information. In a von Neumann machine this will include both programs that are stored in memory when the program is run, and data. The data that is stored is essentially just bits and bytes, but it may represent many things; the letters that form the text of a word processing document, the results of internal calculations, the elements of a routing table or many other types of data.

Memory should be of sufficient capacity to satisfy the data and program storage requirements of the particular device. It should be fast enough in speed not to cause delay to the main processor, and it should be as cheap as possible and reliable.

Memory can be divided into categories according to their operating criteria. Firstly, memory may be either *volatile* or *non-volatile*. Volatile memory has the property that it requires to be connected to a power supply in order to retain information. This would include such items as the SIMM or DIMM memory modules found within PCs; these are normally built from capacitors and need refreshing several times a second in order to retain their contents. Switching off a PC for a number of seconds will totally clear the memory as the contents rapidly decay once power is removed. It also includes memory created from flip-flops, such as registers, although these maintain their state so long as they are connected to a power supply. They do not need to be refreshed.

Non-volatile memory includes such items as hard and floppy disk storage, CD-R, CD-RW, DVD and optical disks, *read only memory* (ROM), flash memory and certain very specialised items such as ferric core memory.

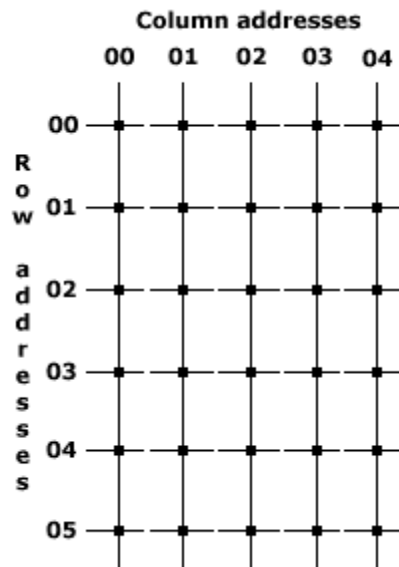
Semiconductor memories are either random access memory (RAM) or some form of ROM. ROM includes technologies such as PROM (*programmable read-only memory*), EPROM (*erasable programmable read-only memory*) and EEPROM (*electrically erasable programmable read-only memory*). Typically, the BIOS (*basic input/output system*) of a PC would be stored in ROM.

RAM is employed as a temporary store for running programs and data. The two basic types of RAM are static (*SRAM*) and dynamic (*DRAM*). SRAM is expensive, and is built from flip-flops. DRAM at the time of writing is extremely cheap, and as previously stated is capacitor based. There are a number of variations upon the theme of DRAM such as SDRAM which works upon the principle that the most likely data to be requested next will be upon the same 'row' of memory as the last request, so it goes ahead and reads it anyway.

The latest development is DDR RAM. DDR stands for Double Data Rate, and it works by retrieving data both on the rising and falling edges of the clock cycle, thus doubling the data rate that it can transfer.

The memory consists of a matrix of memory cells and circuitry that allow address selection and control, as in Figure 6.1. The important requirements are that i) any location in the memory can be found by address and ii) that the data at that address can be read (and in the case of RAM, written).

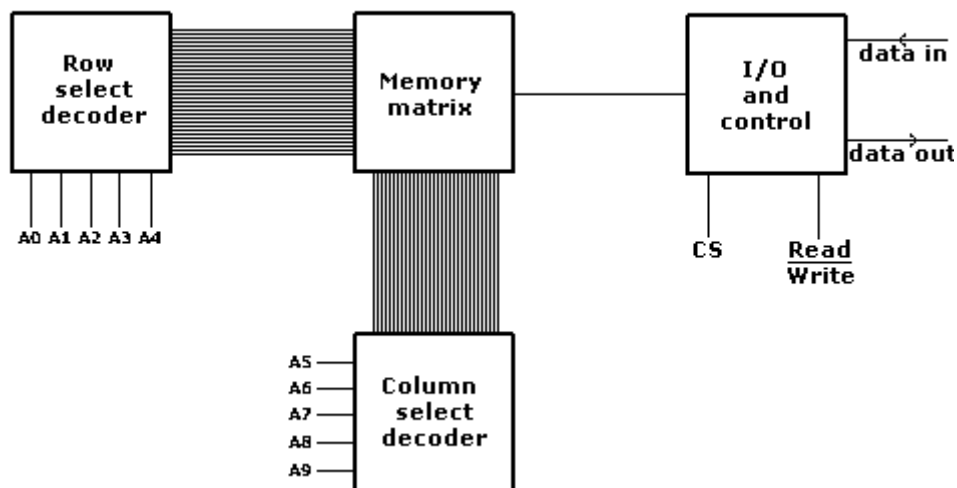
Fig. 6.1



In simple terms, when an address location is chosen, the circuitry of the module makes the logic state of the location available on the data out line.

Figure 6.2 shows the block diagram of a 1k x 1 bit memory. 1Kbit is 2^{10} or 1024 bits made of a matrix of $\sqrt{1024} = 32$ by 32 (2^5) bits. Thus the Row Address Select (RAS) and the Column Address Select (CAS) are 5 to 32 decoders.

Fig. 6.2



It is possible to store more than 1 bit of information at a location, and a typical modern 256MB DIMM fetches 72 bits from each location (64 bits of data and 8 bits for error checking). It is also possible to organize the internal circuitry such that the same lines may be used for both the column and row selects, by multiplexing. This reduces the number of address pins on this DIMM to only twelve, plus two 'Bank select' pins.

In DRAM, the read operation is destructive, that is, the charge on the capacitor is lost when read. This means that every read must be followed by a write to replace the data into its previous state.


ROM works in a very similar way to RAM, except that there is no 'write' operation. The logic of a ROM is programmed during manufacture and cannot be altered. ROMs made in this way are uneconomic to manufacture in small numbers, and a typical use would be the BIOS chip of a PC.

When smaller numbers of devices are to be made, the PROM is more useful. When manufactured the rows and columns are connected by fusible links. When the logic design indicates which locations are to store logic '1', the fuses are blown at those locations by the use of a large electrical current. Once programmed, a PROM cannot be reprogrammed. Typically, these devices would be used in control circuits for washing machines or ovens.

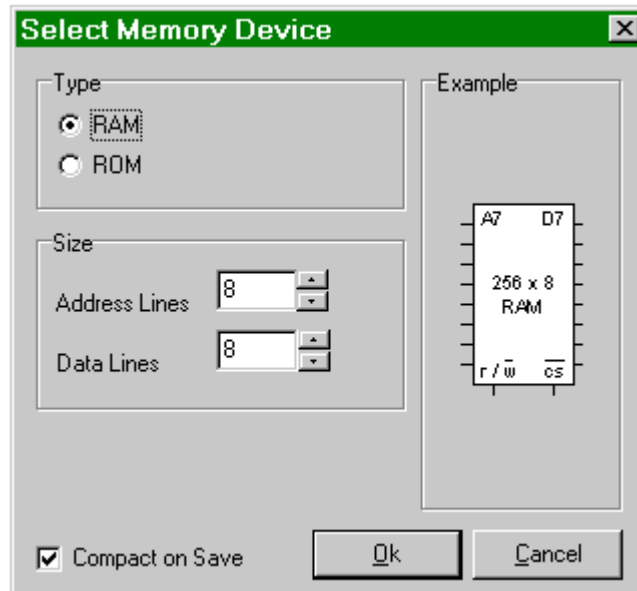
The EPROM may be reprogrammed many times. They work by storing an electrical charge that will hold for many years at the locations where logic '1' is required. They are erased by 20 minutes exposure to high-intensity ultra violet light. This removes all the electrical charges, and resets all locations to logic '0'. One disadvantage to the EPROM is that all locations must be rewritten when reprogramming takes place.

The EEPROM is also programmed by the application of a charge. It offers significant advantages in that it may be programmed *in situ*, the programming may be changed on a single location by the use of a reverse polarity charge and a single location may be programmed. Its disadvantage is that after a number of times of being programmed it loses its ability to hold the necessary charge.

6.2 Memory in Digital Works

Digital Works provides simulation of both ROM and RAM. The icon for this component looks like this  and shows a context pop-up of 'Memory Device'.

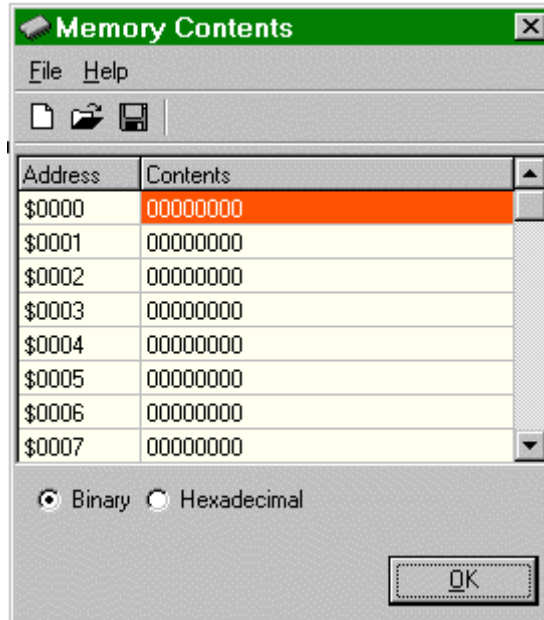
Using Digital Works, and with a fresh canvas, drop a memory device onto the canvas. This will open up a selection menu like this:



It can be seen that there is a choice of creating RAM or ROM, and there is a means of selecting both the number of address lines and the number of data lines. The maximum number of address lines is 14 and the maximum number of data lines 16. The minimum number of address lines is 4 and the minimum number of data lines is 1.

A simple binary calculation will reveal that this equates to a minimum size of memory of 16 bits and a maximum of 256 kilobits. As an exercise work out how the number of address lines and data lines relate to the amount of memory space available.

The memory in these devices is editable and may be saved to a memory map (.map file). Edit it by right clicking on the memory device and selecting the 'Edit Memory Contents' item. The memory editor menu looks like this:



6.3 Registers

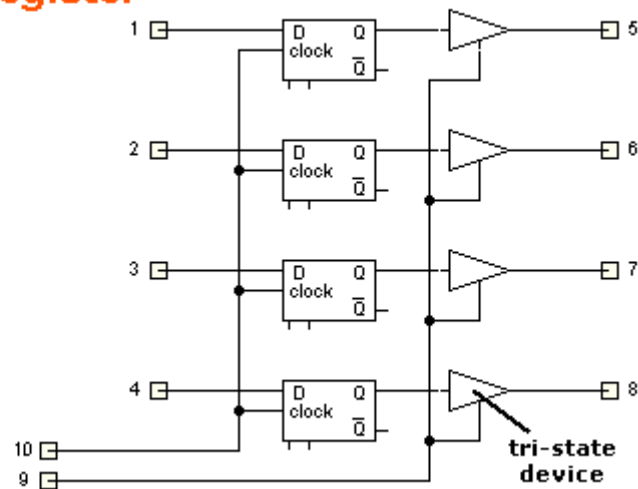
During term 2, you will be programming in assembler language, and you will come to understand registers well. Here, however, we will discuss the concept of registers; what they do and how they are made up.

A register is basically a collection of flip-flops that hold data. There are a number of registers within the CPU of the machine that you are using. These registers perform specific functions such as holding or manipulating data, holding instructions or holding memory addresses. Being situated within the CPU they are extremely fast and are at the heart of how the computer system works. Some registers are available to the low-level programmer, and are known as general purpose registers or accumulators; they may be regarded as being the in-built 'calculators' of the CPU. Other registers such as the Program Counter (which doesn't count!) are not available for the programmer to manipulate.

Using Digital Works, find and open up the 4 bit register macro shown in Figure 6.3.

With the exception of the tri-state device, there are simply four inputs, four D-type flip-flops, a clock and four outputs. The tri-state device is simply a means by which several devices may be connected to a common bus without interacting with one another. The device has the normal HIGH and LOW states, but has a third state of select or enable that allows the output to be switched into or out of the circuit. It would also be possible to connect the four 'CLR' inputs, so as to be able to clear the register, and the PRE inputs could be used to load up a certain bit pattern as required.

Fig. 6.3

Four Bit Register**6.4 Shift Registers**

A shift register consists of a number of flip-flops connected such that data may be stored, exactly as a normal register. A shift register can have its data moved, or *shifted*, either to the left or to the right when it is clocked. They are normally built from either D-type or J-K flip-flops connected in cascade. The number of flip-flops used is equal to the number of bits required to be stored. The flip-flops will also have the usual PRE and CLR inputs for presetting or clearing the register.

There are four ways in which the shift register may operate:

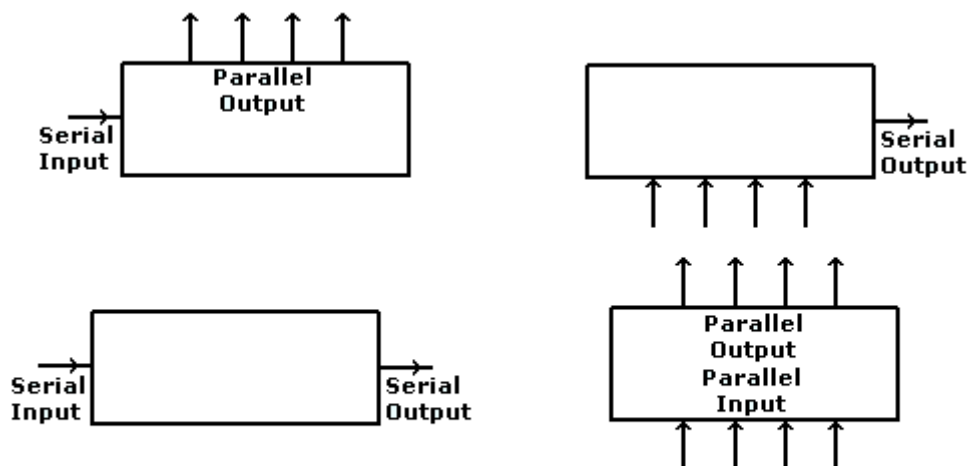
- Serial in/serial out
- Serial in/parallel out.
- Parallel in/serial out
- Parallel in/parallel out

Block diagrams are shown in Figure 6.4 for these operations.

The main uses of the shift register are:

- Temporary storage of data (as per the register)
- Serial to parallel or parallel to serial conversion
- Digital delay circuit
- Mathematical operations.

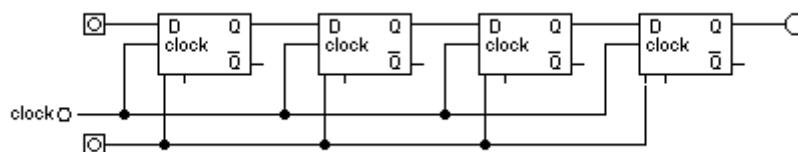
Fig. 6.4



6.4.1 Serial in/serial out

Using Digital Works create the circuit shown in Figure 6.5. This is a very simple serial in/serial out right-shift shift register.

Fig. 6.5



In order to understand what this circuit is doing, add both the interactive input D_{in} and the output LED to the logic history window. Now run the circuit by using the step facility to advance the circuit one clock cycle at a time. Enter various bit patterns by switching D_{in} between the clock cycles. Examine the logic history window and you will see that the output exactly follows the input, but with a delay of $3\frac{1}{2}$ clock cycles.

Note that once the data reaches the output, it is lost at the next clock cycle.

Now build the circuit shown in Figure 6.6. This is almost identical, except that J-K flip-flops are used, and the \bar{Q} output becomes the K input for the next flip-flop. As before, run this circuit and examine the logic history. In this case you will find that the output is exactly 4 clock cycles behind the input. Table 6.1 shows the logical state of each flip-flop at each clock cycle, and by following this from the top you can see that the entire bit pattern moves one flip-flop to the right at each clock cycle. The bit pattern at the input reappears as the bit pattern at the output.

Fig. 6.6

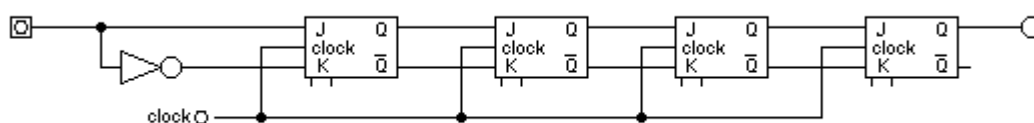
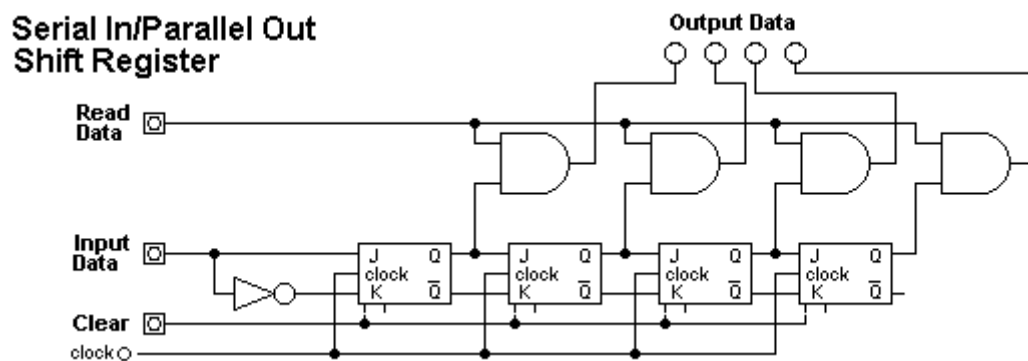


Table 6.1

	Flip-flop 1	Flip-flop 2	Flip-flop 3	Flip-flop 4	Output
Clock	0	0	0	0	0
Clock	1	0	0	0	0
Clock	1	1	0	0	0
Clock	0	1	1	0	0
Clock	1	0	1	1	0
Clock	0	1	0	1	1
Clock	0	0	1	0	1
Clock	0	0	0	1	0
Clock	0	0	0	0	1

6.4.2 Serial In/Parallel Out

The circuit of a serial in/parallel out shift register is very similar to that of the serial in/serial out shift register. The main difference is that the Q output of each flip-flop is accessible externally. Figure 6.7 shows a possible circuit for a serial in/parallel out shift register. Note the addition of the four AND gates that are connected to a 'read data' input such that the Q outputs can only be read when Read Data is HIGH. One full clock cycle is required for each flip-flop in order to load fully the serial input data. Only one clock cycle is required to read all the bits onto a data bus.

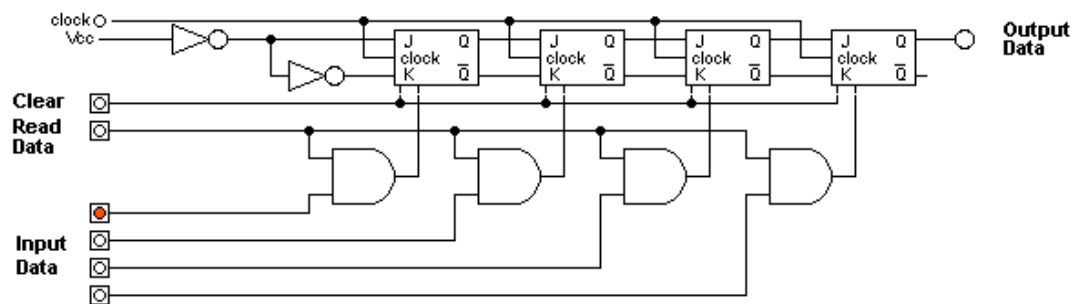
Fig. 6.7

6.4.3 Parallel In/Serial Out

In the parallel in/serial out shift register, shown in Figure 6.8, the operations of loading and reading data take place in exactly the opposite way to that of the serial in/parallel out register. The data is stored by first clearing any data that may already be held on the flip-flops by setting 'Clear' to HIGH for one clock cycle. When the 'Read Data' input is set HIGH for one clock cycle, this will transfer any data held at the data inputs to the respective flip-flop, causing it to be set. Subsequent clock cycles for each of the flip-flops will now cause that data to be transferred to the Output Data terminal.

Fig. 6.8

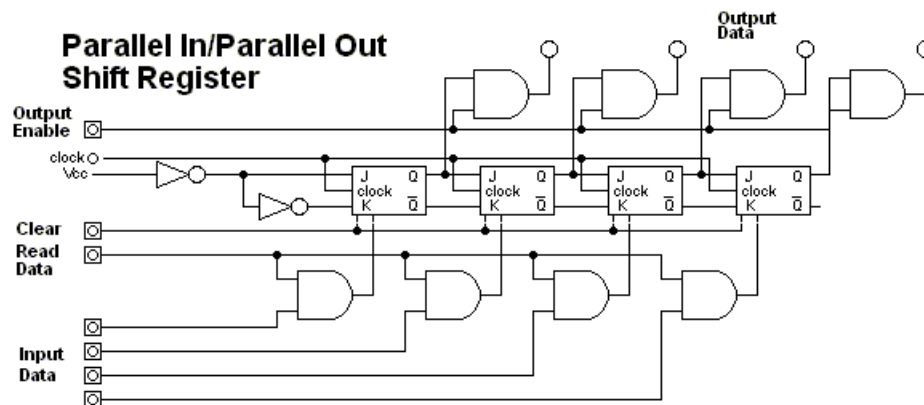
Parallel In/Serial Out Shift Register



6.4.4 Parallel In/Parallel Out

Figure 6.9 shows a parallel in/parallel out shift register. It can be seen that this is essentially a combination of the serial in/parallel out and parallel in/serial out registers, with sets of AND gates for both the input and output stages.

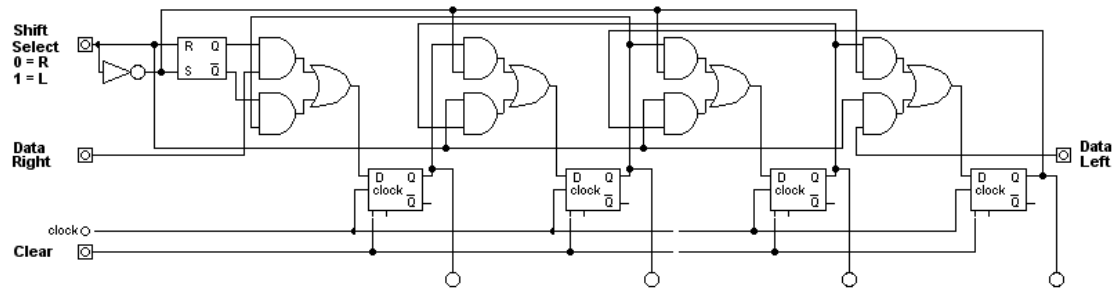
Fig. 6.9



6.4.5 Bi-directional Shift Registers

A *bi-directional shift register* is one whose data can be shifted either to the left or to the right, according to the state of another input. The circuit of such a shift register is shown in Figure 6.10. There are two data input terminals, one at each end of the register. The left-hand one is used for inputting data to be right-shifted, and vice versa. The direction of shifting is determined by the input labelled 'Shift Select'. When this input is HIGH the register will shift LEFT.

Fig. 6.10



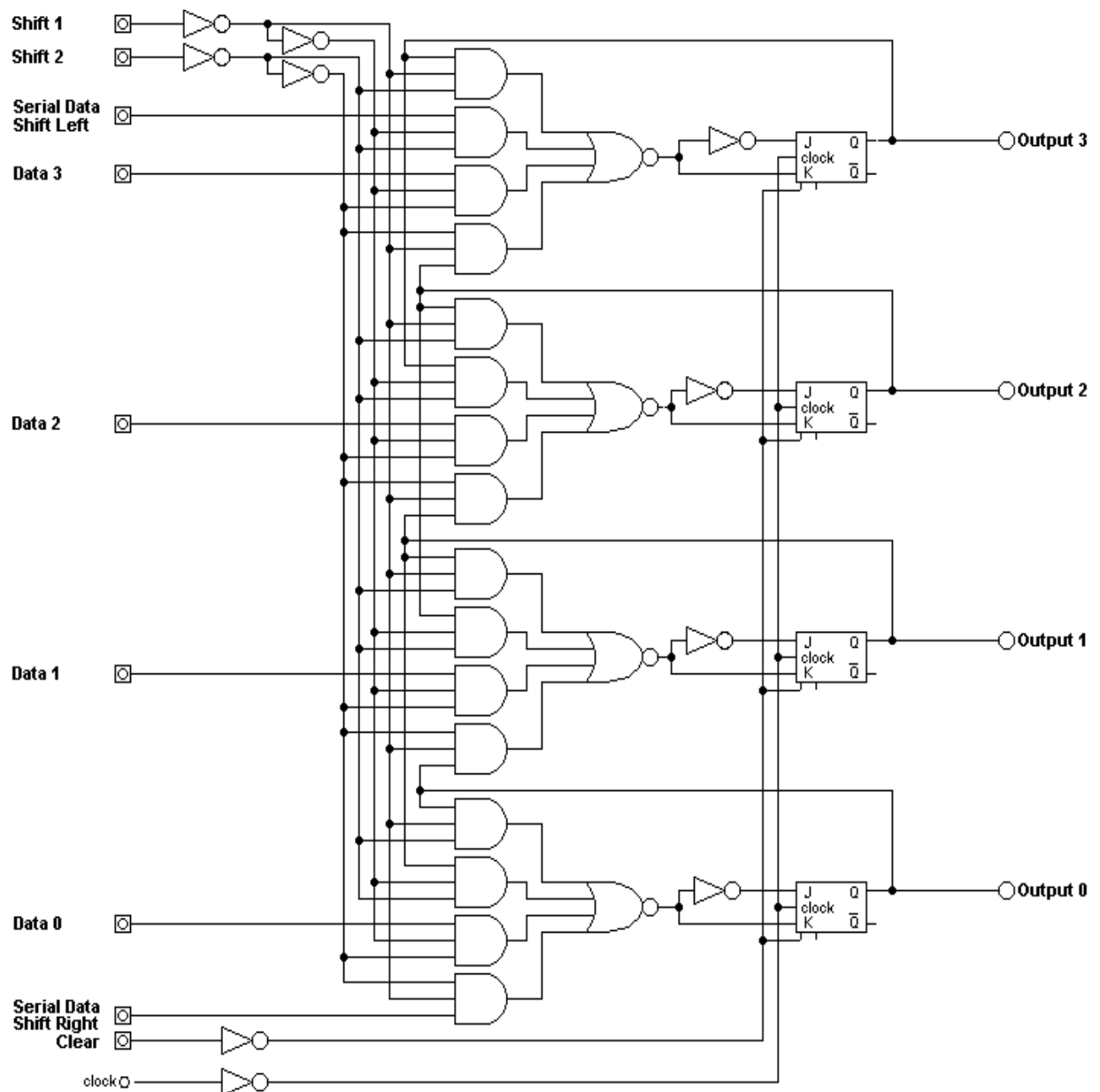
6.4.6 Universal Shift Registers

Figure 6.11 shows the logic circuit for a Universal shift register. This circuit is based on a Phillips Semiconductors 74F194 Universal shift register, and so is representative of actual circuits that are commercially available. This shift register is able, by the selection of the relevant inputs, to act as any one of the shift registers previously described. As you can see, this is a circuit of some complexity.

Its operation is dependent upon the status of the inputs marked Shift 1 and Shift 2, in the following manner:

- When S1 & S2 are both LOW, the register will simply maintain its present state, i.e. this is a 'do nothing' mode
- When S1 is HIGH and S2 is LOW, the register will take the data that is present at the Serial Data Shift Left input, and at each clock cycle will shift this data left
- When S1 is LOW and S2 is HIGH, the register will take the data that is present at the Serial Data Shift Right input, and at each clock cycle will shift this data right
- When both S1 and S2 are HIGH, the shift register will take a parallel load from the four data inputs Data 0 to Data 3.

Fig. 6.11



6.5 Counters

The counters that we are considering are digital circuits that can count the number of clock cycles applied to their inputs. The count may be output in a varying number of ways, such as a straightforward binary representation of the number of cycles counted, or a BCD output, or a unique code representation of each state of the count cycle. It may also be arranged that there is only one output that goes HIGH to indicate that a certain number of cycles have been counted. The count may be in either direction; where the count increases, the counters are referred to as *up-counters* and where the count decreases, *down-counters*. Counters are referred to as being *modulo- n* , where n is the number of different states that they can count. For instance, if a counter can count from 0 – 15 then that is 16 possible states, and this would be referred to as a modulo-16 counter.

Essentially, a counter consists of a number of either J-K or D-type flip-flops connected together as a cascade. In operation, these may be divided into two categories: asynchronous

and synchronous. These categories will be dealt with in detail in the succeeding subsections; at this stage just understand that in a synchronous counter all stages are clocked simultaneously, whereas in an asynchronous counter each stage relies on the change of state of the previous stage.

There are many applications for counters, such as the measurement of time, actual counting of objects, frequency division and so on. The implementation of various types of counter, their adaptation for special purposes and the advantages and disadvantages of the various types are covered in this section.

6.5.1 Asynchronous Counters

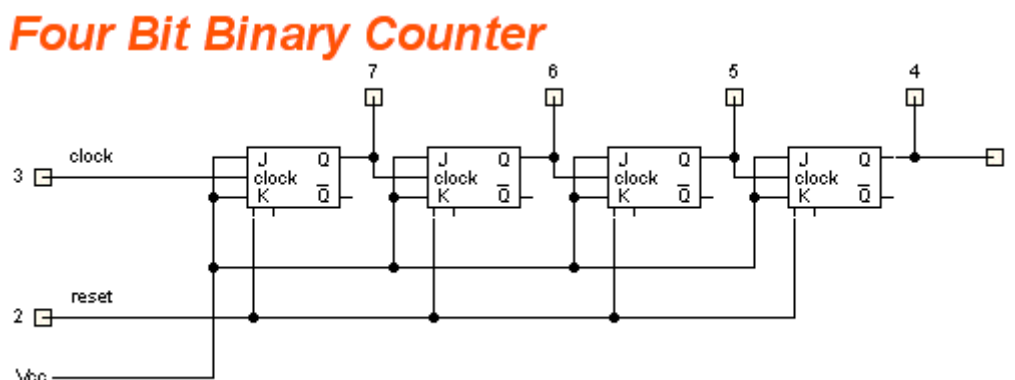
A single J-K flip-flop will act as a divide by two circuit, as we have previously seen. If the J and K terminals are both held HIGH and the circuit clocked the output will toggle at half the clock rate. If this output is now used to clock another J-K flip-flop (also with J and K held HIGH), this will toggle at half the rate of the previous flip-flop and one quarter of the clock rate.

By connecting together a series of J-K flip-flops in this way, n flip-flops will count 2^n states.

An example circuit is given as a macro in Digital Works. With a new canvas, import the macro titled '8 bit Asynchronous Counter'. Now attach a clock to the clock input, and LEDs to the 8 right hand outputs along the bottom of the macro. Run the circuit and you will see that the LEDs light up and extinguish in the exact order of a binary count, up to 1111111_2 . Confirm this by drawing up the truth table for the first (least significant) four bits, starting at 0000, 0001, etc., and compare this with the outputs of the four least significant LEDs of the circuit.

Use the context menu to open up the macro, then open one of the 4-bit counter macros inside. This circuit is shown below as Figure 6.12. An examination of the operation of the circuit reveals that the outputs of the flip-flops seem to ripple through the circuit; for this reason it is frequently known as a *ripple counter*.

Fig. 6.12



This also brings us to one of the shortcomings of this type of device. Consider what happens on the clock cycle after the counter has reached 1111111_2 . On the falling edge of the clock, the first flip-flop will return to a LOW state. The second flip-flop will then 'see' this as the falling edge of its clock, and it in turn will return to a LOW state, thus triggering the third flip-flop, and so on until the final flip-flop is reached. This action of all the flip-flops resetting takes time, and in some circuits this can cause timing problems. Also note that

counters of this type are *cyclic*; that is, having reached their highest count (11111111_2) they roll over to zero (00000000_2).

Example 6.1

Construct a counter that will count the series 0123456789012.....

The first thing to consider is how many bits we need in order to represent these numbers. 2^3 is only 8, and 2^4 is 16, so with 4 bits we can represent the numbers 0 – 15. Our problem is to construct a counter that will roll over back to zero once it has counted 9.

Consider this truth table:

Q_3	Q_2	Q_1	Q_0	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

From this truth table we can see that whilst the number represented is in the range 0 – 9, we do not want the count to roll over.

The point at which we need to make the counter go back to zero is immediately after the 9 has been counted. Recall that J-K flip-flops have an unconditional clear.

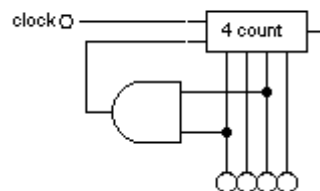
We must therefore issue a clear as soon as the count reaches 10. From our truth table we can see that this is represented by $Q_3 \bullet Q_1$.

This is a reasonably simple example, as we can see from the truth table that as the numbers increase from zero, there is no situation where $Q_3 \bullet Q_1$ is TRUE until the count is 10. In other situations it may be necessary to include also the NOT status of other outputs.

Using the 4-bit asynchronous counter macro supplied with Digital Works, this counter can be constructed as shown in Figure 6.13 (note – Figure is Digital Works V2). Build this circuit and verify that it counts the range 0 – 9. Notice that the Q_3 output is ANDed with the Q_1 output and fed to the Clear terminal.

Fig. 6.13

Modulo-10 Binary Counter



6.5.2 Synchronous Counters

An asynchronous counter is only useful for applications where the speed of operation is not important. We have seen that, where several stages of counting take place, the time taken for a clock pulse to ripple through the entire circuit may well be excessive. The operating time

can be shortened considerably and glitches avoided if all the flip-flops are subject to the same clock input. This is known as synchronous operation.

In a synchronous counter all the flip-flops change state simultaneously, being triggered by the clock. All flip-flops except the first must be prevented from changing state until it is their turn. Figure 6.14 shows how this is achieved for a modulo-8 synchronous counter with three J-K flip-flops. We have already seen that a J-K flip-flop will toggle if both J and K are held HIGH. Both J_A and K_A are held HIGH, so Q_A will toggle with each clock cycle. This Q_A output is fed to both J_B and K_B . The final stage C must only toggle when both Q_A and Q_B are HIGH, so these two outputs are fed to an AND gate, and the output from this gate is used to toggle stage C.

Fig. 6.14

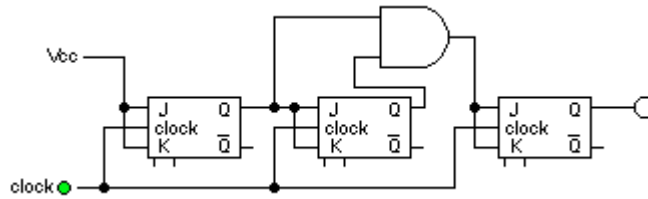


Figure 6.15 shows a modulo-16 counter. Initially the counter is reset so that $Q_A = Q_B = Q_C = Q_D = 0$.

Flip-flop A will toggle on the trailing edge of the first clock cycle so Q_A becomes 1.

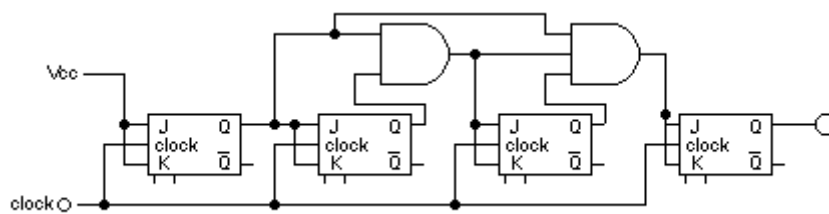
Flip-flop B now has $J_B = K_B = 1$, so it will toggle after one more clock cycle, when Q_A becomes 0. The left hand AND gate now has one input HIGH and one LOW, so flip-flop C has J_C and K_C both LOW so it will not toggle at the end of the next clock cycle.

At the third clock cycle flip-flop A now toggles again, so now $Q_A = Q_B = 1$.

Flip-flop C now has $J_C = K_C = 1$, and so it will toggle at the next clock cycle, as will flip-flops A and B, so now only $Q_C = 1$.

The operation continues in this way, such that all three inputs to the right hand AND gate will be HIGH after the seventh clock cycle. Flip-flop D will therefore toggle at the 8th clock cycle, and the previous sequence will then repeat; flip-flop D will obviously toggle again at the 16th clock cycle, so giving the modulo-16 count.

Fig. 6.15



By applying this principle, counters of any desired modulo base can be designed.

Example 6.2

Design a modulo 6 synchronous counter.

5. Using the 4-bit synchronous counter supplied with Digital Works as a starting point, amend the circuit so that it counts only in the range 1 2 ... 7 1 ...
6. Describe how a shift register may be employed to receive data with the most significant bit first, and then retransmit the data with the least significant bit first.
7. Briefly state the differences between
 - (a) SRAM, DRAM, EDO DRAM, BEDODRAM, VRAM, NVRAM and FRAM.
 - (b) ROM, PROM, EPROM, EEPROM and flash ROM.

7 Binary Arithmetic

In this chapter some more advanced topics on number representation are introduced. The problems associated with the architecture of having to represent both positive and negative numbers within a computer are discussed, together with the successive solutions to those problems.

7.1 Sign and Magnitude Representation

When computers were first used to perform binary arithmetic a fundamental problem had to be addressed. How to represent positive and negative numbers?

The initial solution to this was sign and magnitude representation. The most significant bit was sacrificed to indicate the sign of the number, and it was chosen that if the most significant bit was a '0' the number was positive, so, obviously, if the most significant bit was a '1' the number was negative. Thus:

$$\begin{aligned} 10010101 &= 1 \text{ (sign bit) and } 0010101 \text{ (magnitude)} = -21 \\ 00110110 &= 0 \text{ (sign bit) and } 0110110 \text{ (magnitude)} = +54 \end{aligned}$$

Take as an example *adding* -21 to 54 . A plus and a minus is a minus, so to perform this operation first change the sign bit of -21 from negative to positive, and then do a binary subtraction from 54 , thus:

$$\begin{array}{r} 00110110 \quad (+54) \\ - 00010101 \quad -(+21) \\ \hline = 00100001 \quad (+33) \end{array}$$

So what if we want to *subtract* the -21 from 54 ?

Well, the normal mathematical rule of minus a minus is plus applies. So, we change the sign bit of the 21 to a 0 (positive) and add the numbers:

$$\begin{array}{r} 00010101 \quad (+21) \\ + 00110110 \quad (+54) \\ \hline = 01001011 \quad (+75) \end{array}$$

Problem

Given this solution to handling binary numbers:

$$\begin{aligned} \text{If } 00000000_2 &= 0 \\ \text{What does } 10000000_2 &\text{ equal?} \end{aligned}$$

Well, obviously minus 0 ! This means that using sign and magnitude representation we cannot make the computer represent 0 . It can only represent $+0$ or -0 .

7.2 Ones Complement

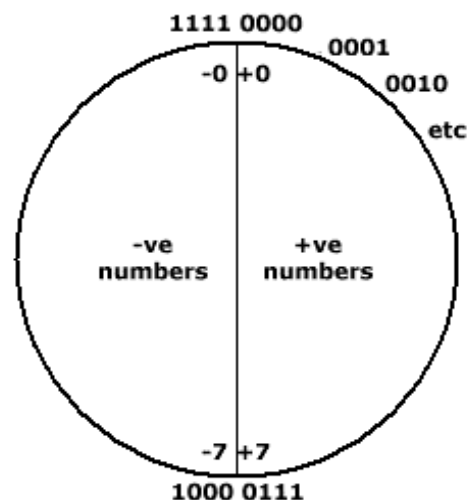
The next step is the ones complement method of representing negative numbers. Obtaining the ones complement is a simple step of 'complementing' the number. That is, all the '0's become '1's and all the '1's become '0's. Thus:

Dec	Bin	Dec	Bin
+0	0000	-0	1111
1	0001	-1	1110
2	0010	-2	1101
3	0011	-3	1100
4	0100	-4	1011
5	0101	-5	1010
6	0110	-6	1001
7	0111	-7	1000

Notice that the negative numbers are the exact complement of the respective positive number, and that all the negative numbers have a '1' as the most significant digit. We are still left with the situation of having two values for zero - +0 and -0.

There are two other particular things to note about this system. Firstly that the negative binary numbers decrease as the negative value increases, and secondly that if one considers the natural progression of the binary series, the decimal values suddenly 'wrap around' to negative numbers at the point where the most significant digit becomes a '1'. The numbers may be represented as a diagram in which the numbers form a circle, as in Figure 7.1.

Fig. 7.1



You should already have encountered this 'wrapping around' of the numbers during your C programming, when you exceed the range of an integer in 'C'.

7.3 Twos Complement

We have seen that one of the disadvantages of the systems we have looked at so far is that there are two values for zero ($00000000_2 = +0$ and $10000000_2 = -0$). Twos complement handles this situation in such a way that there is only one representation for zero. The twos complement of a number is simply the ones complement plus one.

Example 7.3

Find the twos complement of 27_{10} .

$$27_{10} = 011011_2$$

therefore the ones complement is 100100_2
 add 1 to this gives 100101_2 (Ans.)

An alternative method:

subtract this from the next higher power of 2

$$\begin{array}{r} 27_{10} = 011011_2 \\ 100000_2 \\ - \underline{011011_2} \\ = 100101_2 \text{ (Ans.)} \end{array}$$

and another alternative method

now complement all bits to the *left*
 of the least significant bit

$$\begin{array}{r} 27_{10} = 011011_2 \\ 100101_2 \text{ (Ans.)} \end{array}$$

A special case:

Find the twos complement of 00000000_2 (zero)

Ones complement is 11111111_2
 Twos complement is 11111111_2
 $+ \underline{00000001_2}$
 $= 100000000_2$

discard the overflow bit leaves 00000000_2
 Therefore $-0 = 0$

Hey presto! **Only one value for zero!**

7.4 Subtraction using Ones Complement and Twos Complement

We have already seen in the section on combinational logic circuits that an adder circuit is easily constructed, and is readily available as an IC.

By using ones complement or twos complement arithmetic, it is possible to do a subtraction by adding the ones or twos complement number. This saves the effort and complication of designing subtraction circuits.

Example 7.4

Using (a) ones complement and (b) twos complement, calculate $26 - 9$, and $9 - 26$.

$$26_{10} = 11010_2 \text{ and } 9_{10} = 01001_2$$

(a) The ones complement of 9_{10} is 10110_2 ,
 therefore:

$$\begin{array}{r} 11010_2 \\ + \underline{10110_2} \\ = 110000_2 \end{array}$$

there is a carry bit, so that we must remove
 the carry bit from the most significant digit

The ones complement of 26_{10} is 00101_2 ,
 therefore:

$$\begin{array}{r} 01001_2 \\ + \underline{00101_2} \\ = 01110_2 \end{array}$$

there is no carry bit so the result is negative.
 Now we complement 01110_2 :

position and add it to the least significant digit position, so:

$$= 10001_2 \text{ which is } -17_{10} \text{ (Ans.)}$$

$$\begin{array}{r} 10000_2 \\ + \quad \underline{1_2} \\ = 10001_2 \text{ which is } 17_{10} \text{ (Ans.)} \end{array}$$

- (b) The two's complement of 9_{10} is 10111_2 , therefore:

The two's complement of 26_{10} is 00110_2 , therefore:

$$\begin{array}{r} 11010_2 \\ + \quad \underline{10111_2} \\ = 110001_2 \end{array}$$

$$\begin{array}{r} 01001_2 \\ + \quad \underline{00110_2} \\ = 01111_2 \end{array}$$

there is a carry bit, but in this case it simply tells us that the answer is positive, so just discard it to get the answer, thus:

again, there is no carry bit, so the answer is negative. Complement 01111_2 and add 1:

$$= 10001_2 \text{ which is } 17_{10} \text{ (Ans.)}$$

$$\begin{array}{r} 10000_2 \\ + \quad \underline{1_2} \\ = 10001_2 \text{ which is } -17_{10} \text{ (Ans.)} \end{array}$$

7.5 Exercises

- Add 10110110_2 , 01110100_2 and 11000101_2
- Add 10001111_2 , 01011001_2 and 00111101_2
- Subtract
 - $73_{10} - 14_{10}$
 - $33_{10} - 17_{10}$
 - $56_{10} - 75_{10}$

using

- Direct binary subtraction
- Ones complement
- Two's complement

- Use two's complement arithmetic to perform
 - $127_{10} - 46_{10}$
 - $127_{10} - 128_{10}$
 - $1_{10} - 3_{10}$
- Use two's complement arithmetic to solve $58_{10} - 33_{10} + 14_{10} + (-45_{10})$
- Write down the two's complement of the following decimal numbers:

(a) -7	(b) -77	(c) -126	(d) -200
----------	-----------	------------	------------
- Evaluate using the two's complement method
 - $11011101_2 - 1101_2$
 - $10110111_2 - 100001_2$
 - $100000000000_2 - 101111_2$

8 Floating Point Numbers

Previous chapters of this book have dealt with various numerical and character representations that are used in current computer architectures. This Chapter will explain exactly how real numbers are represented at a binary level within the computer, and the uses and advantages of the floating point system. There are many different ways that real numbers can be represented, but we will concentrate on the representations as specified in IEEE Standard 754.

8.1 Scientific or Exponential Notation

Numbers in the denary (base 10) system may be represented quite arbitrarily in scientific or exponential notation. For example 17.2345 may equally be represented as:

$$17.2345 \times 10^0 \text{ or } 0.0000172345 \times 10^6 \text{ or } 172,345 \times 10^{-4}$$

All are equally valid scientific notations of the same number. They can also be quoted as 17.2345E0 (exponential notation).

Why use this notation at all? Well, with numbers in the range of say 10^6 to 10^{-6} there is often no reason at all. However, once we as human beings start dealing with either very large or very small numbers, ordinary notation becomes very error prone. As an example, imagine multiplying

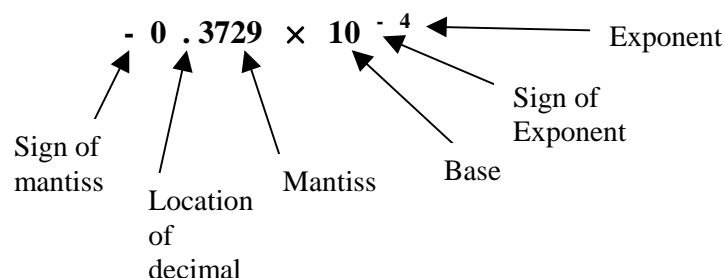
7,234,861,220,000,000,000,000,000 by
0.000000000000000000000000394756

This notation can be divided into a number of components, all of which are essential to maintain the accuracy of the information, but some of which may be implied.

These are:

1. The sign of the number (+ or -)
2. The magnitude of the number, also called the **mantissa**
3. The sign of the exponent
4. The magnitude of the exponent
5. The base of the exponent
6. The location of the decimal point

To take an example let us look at the number -0.00003729 . This can be represented in the following way:



8.4 Normalisation

The IEEE standard calls for the ‘normalisation’ of the floating point number. It also uses a rather neat trick to achieve an extra bit of precision by always implying that the number immediately to the left of the binary point is a ‘1’ bit.

Let us now look at normalisation in practice.

Example 9.1

Normalisation of floating point number 5.0 in IEEE 754

Binary value (Mantissa)	Exponent	IEEE Exponent	IEEE (Binary)
0101.00000000000000000000	2^0	127	01111111
010.10000000000000000000	2^1	128	10000000
01.01000000000000000000	2^2	129	10000001

This number is now normalised to IEEE 754 floating point format, and from the representation shown above we are able to complete the binary representation of 5.0 as:

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF
0 10000001 0100000000000000000000
```

which is seen to be a 32 bit binary number. Convert this into hexadecimal and it becomes 40A00000.

Just to prove that this is in fact how floating point operates, the following ‘C’ code gives the output shown in Figure 9.2, and checking the actual contents of memory gives the result in Figure 9.3.

```
#include <stdio.h>

int main(void)
{
    float i;    /*tut-tut, naming conventions!!*/
    i = 5.0;
    printf("Address %x\n", &i);
    printf("Number is %.2f\n", i);

    return 0;
}
```

Figure 9.2

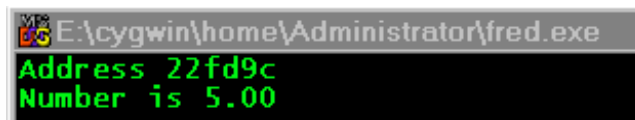


Figure 9.3

	0	4	8	C	ASCII
0x22fd9c	0x40a00000	0x0022ff30	0x61005b8e	0x00000001	...@0."...[.a....
0x22fdac	0x10021658	0x10020278	0x0022fde4	0x00000040	X...x.....".@...
0x22fdb0	0x10021684	0x0022fdd0	0x6d6f682f	0x64412f65"/home/Ad
0x22fdbc	0x696e696d	0x61727473	0x2f726f74	0x64657266	ministrator/fred
0x22fdd0	0x65786500	0x7ffdeb00	0x610705eb	0x610c41f4	.exe.....a.A.a
0x22fdec	0x00000000	0x0022fe0c	0x00000020	0x0000002c".
0x22fdf0	0x10021650	0x0022fe30	0x610a9d52	0x610c41ec	P...0."..R..a.A.a
0x22fe0c	0x00000000	0x0022fe40	0x00000001	0x00000030	...@.".....0...

As per example 9.1, IEEE
754 representation of 5.0

Example 9.2

Normalisation of floating point number -0.15625 in IEEE 754

Binary value (Mantissa)	Exponent	IEEE Exponent	IEEE (Binary)
0.001010000000000000000000	2^0	127	01111111
0.010100000000000000000000	2^{-1}	126	01111110
0.101000000000000000000000	2^{-2}	125	01111101
01.010000000000000000000000	2^{-3}	124	01111100

This number is now normalised to IEEE 754 floating point format, and from the representation shown above we are able to complete the binary representation of -0.15625 as:

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF
1 01111100 0100000000000000000000
```

Convert this into hexadecimal and it becomes BE200000. You will also notice that this example was specifically chosen because the fractional part of the number is identical to that in Example 9.1.

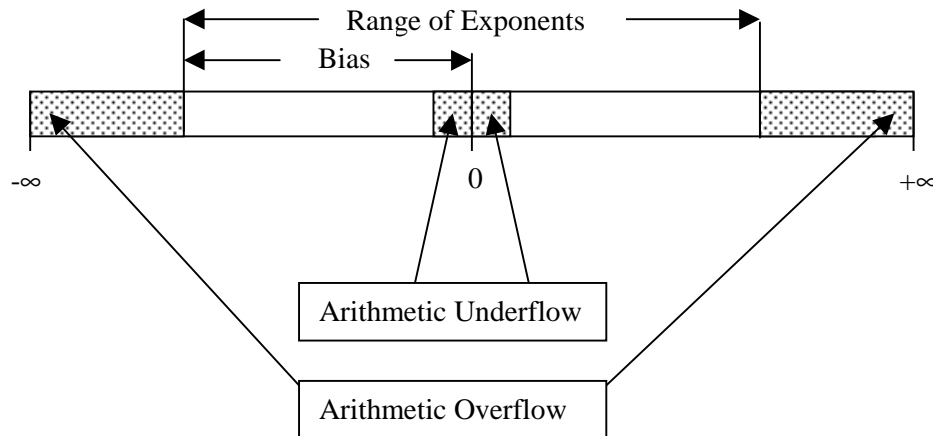
From these examples you can see that this representation allows a huge range of numbers to be represented. The 'C' library file float.h contains definitions of the minimum and maximum values of floats and doubles. These are $1.17549435 \times 10^{-38}$ minimum and $3.40282347 \times 10^{38}$ maximum for floats and $2.2250738585072014 \times 10^{-308}$ minimum to $1.7976931348623157 \times 10^{308}$ for doubles. To give these numbers some sort of context, the weight of a single neutron is $1.67492716 \times 10^{-24}$ grams – easily within the numeric range of a float, let alone a double.

8.5 Special Values

There are a number of special values that the IEEE 754 format defines. These are used to catch specific 'out-of-range' situations.

The range of representations can be visualised as a continuum of numbers between $+\infty$ and $-\infty$ as shown in Figure 9.4. The shaded areas represent numerical ranges that cannot be represented using the floating point system, i.e. numbers outside the ranges quoted in the previous section.

Fig. 9.4



8.5.1 Denormalised

If the exponent is all '0's, and the fraction is non-zero, this represents a *denormalised* number. Specifically, in this situation the implied leading '1' bit of the mantissa is no longer implied, so the fractional part has the straightforward value that the bits represent.

8.5.2 Zero

Because the IEEE scheme implies that there is a '1' bit in front of the binary fraction, it is not possible to represent zero directly. Instead, it is defined that when both the exponent and fractional parts of the number are all '0' bits, this represents zero. However, note that the sign bit is not mentioned, therefore it is possible to have positive and negative values of zero. Comparisons between $+0$ and -0 evaluate to equality.

8.5.3 Infinity

If the exponent is all '1's and the fraction is all '0's, this is defined as infinity. Again, the sign bit is not mentioned so there are positive and negative infinities ($+\infty$ and $-\infty$).

8.5.4 Not a Number (NaN)

Where an operation yields a result that does not represent a real number, this result is defined as being 'Not a Number', or NaN. These are represented by exponents of all '1' bits and non-zero fractions. These representations subdivide into 'Quiet' NaNs and 'Signalling' NaNs (QNaNs and SNaNs). A QNaN has the most significant bit of the fraction set and a SNaN has it clear. QNaNs indicate indeterminate operations and SNaNs indicate invalid operations.

There are very specific rules laid down when dealing with operations on these special numbers. For instance any operation on a NaN returns a NaN, $\text{Infinity} - \text{Infinity}$ is NaN, $\text{Infinity} \times \text{Infinity} = \text{Infinity}$, and so on.

8.6 IEEE Expressions

The IEEE notation can be expressed in terms of a simple expression:

$$X = -1^S \times 2^{E-B} \times 1.F$$

To summarise, a single precision floating point number, or a float consists of three parts:

- S** The sign bit, a 0 indicates a positive value, while a 1 indicates a negative number.
- E** The exponent, an 8-bit value, offset by the bias (B), to indicate how many places to move the binary point. For a single precision number the bias is 127. For a double precision number (a double) the exponent is an 11-bit value, and the bias is 1027.
- F** The Fractional part of the normalized mantissa. The mantissa is normalized into the form 1.F, thus there is no requirement to store the initial 1. In a single precision number the fractional part is 23 bit, while in a double precision number it is some 52-bits.

8.7 Exercises

1. The 'Italdoofouruz' floating point system uses a 12 bit representation, using IEEE 754 guidelines for representation of zero and NaN and with a bias of 7, of the format
SEEEEEFFFFFFFFF

Evaluate:

- a) The greatest negative number that can be represented
 - b) The smallest negative number that can be represented
 - c) The greatest positive number that can be represented
 - d) The smallest positive number that can be represented.
2. Convert 5.82754E-7 into a floating point number.
You may use:
 - a) IEEE 754 format, or
 - b) Any other recognized number format, or
 - c) A format of your own invention (in which case you should state the parameters that you have applied)

Your answer should be shown in hexadecimal format. You should give your answer in both big-endian and little-endian formats, and state what type of representation you have used.

3. Compile and run the two short 'C' programs on the next page. Do these programs perform in the way that you would expect them to? If they do not, can you give an explanation for the behaviour that you have noticed? Are there any implications for you as programmers?

```

#include <stdio.h>

int main(void)
{
    float i;    /*tut-tut, naming conventions!!*/

    for (i = 0; i <= 1.0; i += 0.1)
    {
        printf("Number is %.2f\n", i);
    }

    return 0;
}

=====

#include <stdio.h>

int main(void)
{
    float i;    /*tut-tut, naming conventions!!*/

    for (i = 0; i != 1.0; i += 0.1)
    {
        printf("Number is %.2f\n", i);
    }

    return 0;
}

```

4. For the brave, or foolhardy! Convert the number e into an IEEE 754 double precision floating point number, in hexadecimal format. Your answer should detail the derivation of the number e , to the relevant precision.

Appendix 1. Numbering Systems

1.1. Decimal System

The originals of the decimal system are shrouded in history. Some authorities claim that it originated in India around 5000 B.C., whereas others quote the Shang dynasty in China of about 1300 B.C.

The decimal system is sufficiently familiar not to need a detailed description here, other than a reminder of exactly *what* our decimal numbers represent, which will be useful in section 1.5 – Converting between Number Systems. Remember that 4251_{10} is:

$$(10^0 \times 1) + (10^1 \times 5) + (10^2 \times 2) + (10^3 \times 4)$$

1.2. Hexadecimal System

Most of the microprocessors that we come into contact with today deal with bits in groups of 4, 8, 16, 32 or 64. For this reason, the *hexadecimal*, or base 16, system is extremely useful.

One of the main reasons for this usefulness is that it is much easier for humans to read and understand hexadecimal numbers than binary. For instance, 12345_{16} is 10010001101000101_2 in binary. Note that each hexadecimal digit represents a corresponding four binary digits reading from the right.

This numbering system uses the digits 0–9 and the letters A–F to represent the range 0–15. There are also important implications regarding the direct mapping of binary to hexadecimal numbers, and the fact that most compilers recognize hexadecimal numbers, and many debugging tools report such things as memory addresses and register contents in hexadecimal notation.

For this reason you should be thoroughly familiar with this system, and with conversions both to and from the binary and decimal systems that are covered in detail in section 1.5 below.

1.3. Octal System

The *octal* system uses the base 8, so, as with the hexadecimal system, this creates a convenient method of grouping binary numbers. The octal system maps well onto the concept of the *byte* (8 bits). A group of three binary digits can be represented as a single octal digit.

As an aside, computing students should be aware that most compilers will ‘see’ integer type numbers with leading zeros as octal numbers (e.g. $x = 010$ is assigning x the value of octal 10, which is decimal 8). This can lead to inadvertent and hard to trace errors in code.

1.4. Binary Coded Decimal

In the binary coded decimal (BCD) system numbers are stored as the binary equivalent of the actual decimal digit. There are some advantages to doing this but there are also disadvantages. 4 bits of data are required for each digit as show in the following table, where X indicates binary patterns that are not used.

Dec	Bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
X	1010
X	1011
X	1100
X	1101
X	1110
X	1111

Since 4 bits of data can represent numbers from 0–15, and the BCD system only represents from 0–9, we can see that it is somewhat wasteful. A ten digit BCD number obviously requires 40 bits of data, whereas using a straight binary number would only require 34 bits; 2^{34} being 17179869184 – eleven digits.

Mathematics using BCD can be cumbersome, and many machines using BCD codes actually convert the operands to straight binary, perform the operation, then convert back to BCD.

The following ‘simple’ example shows what is involved in BCD calculations.

Take the simple addition $8 + 7$:

$$\begin{array}{r}
 \text{Decimal} \qquad \qquad \text{BCD} \\
 \begin{array}{r}
 8 \\
 + 7 \\
 \hline
 15
 \end{array}
 \qquad
 \begin{array}{r}
 1000 \\
 + 0111 \\
 \hline
 1111
 \end{array}
 \end{array}$$

which exceeds 9, so can't be a BCD number. We therefore have to add 6, which is the number of BCD codes that are not used

$$\begin{array}{r}
 15 \\
 + 6 \\
 \hline
 21
 \end{array}
 \qquad
 \begin{array}{r}
 1111 \\
 + 0110 \\
 \hline
 0001\ 0101
 \end{array}
 = \text{BCD } 15$$

This BCD representation uses the ‘normal’ 8421 bit weighting. That is, the leftmost or most significant bit represents the binary number 1000 or 8_{10} , the next most significant bit represents 100 or 4_{10} , the next 10 or 2_{10} and the least significant bit represents 1.

You should be aware that there are other BCD representations such as 4221 BCD, where the bits are weighted 4, 2, 2 and 1 respectively.

You may like to read IBM's Decimal Arithmetic FAQ[†] as this gives an interesting insight into the hardware and software issues involved in the use of BCD.

1.5. Converting between Number Systems

1.5.1. Decimal to Binary

There are two algorithms for converting from decimal to binary. The first uses powers of 2 and a subtraction method. The second uses remainders and division by two. These two methods are demonstrated by the examples below.

Example A1.1

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

[†] <http://rexh.hursley.ibm.com/decimal/decifaq.html>

Obtain the binary equivalent of 478_{10} :

478	–	256	=	222	(1 x 256)
222	–	128	=	94	(1 x 128)
94	–	64	=	30	(1 x 64)
30	–	32	won't go	(0 x 32)	
30	–	16	=	14	(1 x 16)
14	–	8	=	6	(1 x 8)
6	–	4	=	2	(1 x 4)
2	–	2	=	0	(1 x 2)
nothing left so:				0	(0 x 1)

Reading from the *top* we have 111011110_2 which is the binary equivalent of 478_{10} .

Example A1.2

Obtain the binary equivalent of 478_{10} (method 2):

$478 \div 2 =$	239	remainder	0
$239 \div 2 =$	119	remainder	1
$119 \div 2 =$	59	remainder	1
$59 \div 2 =$	29	remainder	1
$29 \div 2 =$	14	remainder	1
$14 \div 2 =$	7	remainder	0
$7 \div 2 =$	3	remainder	1
$3 \div 2 =$	1	remainder	1
		which leaves	1

Reading from the *bottom* we have 111011110_2 which is the binary equivalent of 478_{10} .

Either of these methods is equally acceptable, but be aware that when dealing with a large decimal number you will need to draw up a quite extended table of powers of 2!

1.5.2.Binary to Decimal

There are again two algorithms for this conversion. One of these algorithms is much simpler than the other, and is recommended.

Example A1.3

Obtain the decimal equivalent of 1011101110101101_2 :

Working from the right hand side (least significant digit) we have

$1 \times 2^0 =$	1	+
$0 \times 2^1 =$	0	+
$1 \times 2^2 =$	4	+
$1 \times 2^3 =$	8	+
$0 \times 2^4 =$	0	+
$1 \times 2^5 =$	32	+
$0 \times 2^6 =$	0	+
$1 \times 2^7 =$	128	+
$1 \times 2^8 =$	256	+

$$\begin{array}{rclcl}
 1 \times 2^9 & = & 512 & + & \\
 0 \times 2^{10} & = & 0 & + & \\
 1 \times 2^{11} & = & 2048 & + & \\
 1 \times 2^{12} & = & 4096 & + & \\
 1 \times 2^{13} & = & 8192 & + & \\
 0 \times 2^{14} & = & 0 & + & \\
 1 \times 2^{15} & = & \underline{32768} & & \\
 & & = 48045 & &
 \end{array}$$

Example A1.4

Obtain the decimal equivalent of 1011101110101101_2 (method 2) – the **bold** numbers indicate the number that we are converting – starting with the *most* significant digit:

$$\begin{array}{rclcl}
 \mathbf{1} & & & & \\
 \times 2 & = & & & \\
 2 + \mathbf{0} & = & 2 & & \\
 \times 2 & = & & & \\
 4 + \mathbf{1} & = & 5 & & \\
 \times 2 & = & & & \\
 10 + \mathbf{1} & = & 11 & & \\
 \times 2 & = & & & \\
 22 + \mathbf{1} & = & 23 & & \\
 \times 2 & = & & & \\
 46 + \mathbf{0} & = & 46 & & \\
 \times 2 & = & & & \\
 92 + \mathbf{1} & = & 93 & & \\
 \times 2 & = & & & \\
 186 + \mathbf{1} & = & 187 & & \\
 \times 2 & = & & & \\
 374 + \mathbf{1} & = & 375 & & \\
 \times 2 & = & & & \\
 750 + \mathbf{0} & = & 750 & & \\
 \times 2 & = & & & \\
 1500 + \mathbf{1} & = & 1501 & & \\
 \times 2 & = & & & \\
 3002 + \mathbf{0} & = & 3002 & & \\
 \times 2 & = & & & \\
 6004 + \mathbf{1} & = & 6005 & & \\
 \times 2 & = & & & \\
 12010 + \mathbf{1} & = & 12011 & & \\
 \times 2 & = & & & \\
 24022 + \mathbf{0} & = & 24022 & & \\
 \times 2 & = & & & \\
 48044 + \mathbf{1} & = & & & \\
 \text{Answer } \underline{48045} & & & &
 \end{array}$$

This method may seem awkward and difficult, but you should remember that this method may be used for conversions between any number bases. Conversions from binary obviously involve the maximum number of calculations, as the multiplication factor is only 2.

1.5.3. Decimal to Hexadecimal

Once again there are the two algorithms for performing this calculation, a division/remainder method and a product of powers method.

Example A1.5

Use the division/remainder method to obtain the hexadecimal equivalent of 42109_{10} .

Step 1	$42109 \div 16 =$	2631	Remainder	13	13_{10}	$= 0xD$
Step 2	$2631 \div 16 =$	164	Remainder	7	7_{10}	$= 0x7$
Step 3	$164 \div 16 =$	10	Remainder	4	4_{10}	$= 0x4$
Step 4	$10 \div 16 =$	0	Remainder	10	10_{10}	$= 0xA$

Reading from the *bottom* we have 0xA47D (answer).

Example A1.6

Use the product of powers method to obtain the hexadecimal equivalent of 42109_{10} .

First we need the powers of 16: $16^0 = 1$

$$\begin{aligned} 16^1 &= 16 \\ 16^2 &= 256 \\ 16^3 &= 4096 \\ 16^4 &= 65536 \end{aligned}$$

Divide 42109 by 65536 – won't go, so find out how many times 4096 will go into 42109

$$\begin{array}{rclcl} & 42109 & - & & \\ 4096 \times 10 & = & \underline{40960} & & 10_{10} = 0xA \\ & 1149 & - & & \\ 256 \times 4 & = & \underline{1024} & & 4_{10} = 0x4 \\ & 125 & - & & \\ 16 \times 7 & = & \underline{112} & & 7_{10} = 0x7 \\ \text{Remainder} & & \underline{13} & & 13_{10} = 0xD \end{array}$$

Reading from the *top* we have 0xA47D (answer)

Neither of these methods is 'better' than the other. Practice both methods, and use the one with which you are most comfortable.

1.5.4.Hexadecimal to Decimal

As before, there are two algorithms for this conversion. The first method is the sum of powers method, and the second the multiply and add method.

Example A1.7

Convert 0xBEAD to decimal using the sum of powers method.

Starting from the least significant digit

$$\begin{aligned} 0xBEAD &= (D \times 16^0) & + (A \times 16^1) & + (E \times 16^2) & + (B \times 16^3) \\ &= (D \times 1) & + (A \times 16) & + (E \times 256) & + (B \times 4096) \\ &= (13 \times 1) & + (10 \times 16) & + (14 \times 256) & + (11 \times 4096) \end{aligned}$$

$$\begin{aligned}
 &= 13 \quad + 160 \quad + 3584 \quad + 45056 \\
 &= 48813 \text{ (Ans.)}
 \end{aligned}$$

Example A1.8

Convert 0xBEAD to decimal using the multiply and add method.

Starting from the most significant digit

$$\begin{aligned}
 0xB (11_{10}) &= 11 \\
 11 \times 16 &= 176 \\
 176 + 0xE (14_{10}) &= 190 \\
 190 \times 16 &= 3040 \\
 3040 + 0xA (10_{10}) &= 3050 \\
 3050 \times 16 &= 48800 \\
 48800 + 0xD (13_{10}) &= 48813 \quad \text{(Ans.)}
 \end{aligned}$$

1.5.5.Hexadecimal to Binary

This conversion, and the following three are relatively straightforward, due to the direct mapping between the powers of 2 and the numbers represented.

Conversion from hexadecimal to binary is extremely simple:

Example A1.9

Convert 0xDEADBEEF to binary, not forgetting that the decimal digits 10–15 are represented by A–F:

All that is necessary is to convert each hex digit in turn, starting from either end, thus;

Hex	D	E	A	D	B	E	E	F
Bin	1101	1110	1010	1101	1011	1110	1110	1111

Answer 11011110101011011011111011101111₂.

1.5.6.Binary to Hexadecimal

The opposite conversion is equally easy. Simply divide the binary number into groups of four, starting at the least significant digit, and substitute the relevant Hex digit.

Example A1.10

Convert 11111110111011011011101010111110₂ to hexadecimal:

Method – divide into groups of four binary digits, starting from the right hand side, and convert each one, thus

Bin	1111	1110	1110	1101	1011	1010	1011	1110
Hex	F	E	E	D	B	A	B	E

Answer 0xFEEDBABE.

1.5.7.Decimal to Octal

Whilst the octal numbering system is much less popular in the computing field than it used to be, you should still be aware of how to do these conversions. You should also be aware of the fact that many compilers (especially 'C') regard any number with a leading zero as being specifically declared as an octal number. This can result in subtle and hard to detect programming errors!

The same algorithms may be applied to this conversion as for decimal to hexadecimal.

Example A1.11

Convert 42315_{10} to octal using the division/remainder method.

$$\begin{array}{rcl}
 42315 \div 8 = & 5289 & \text{Remainder } 3 \\
 5289 \div 8 = & 661 & \text{Remainder } 1 \\
 661 \div 8 = & 82 & \text{Remainder } 5 \\
 82 \div 8 = & 10 & \text{Remainder } 2 \\
 10 \div 8 = & 1 & \text{Remainder } 2 \\
 1 \div 8 = 0 & & \text{Remainder } 1
 \end{array}$$

Reading from the bottom we have: 122513_8 (Ans.)

Example A1.12

Convert 42315_{10} to octal using the product of powers method.

First obtain the powers of 8

$8^0 = 1$	$42315 \div$	$262144 = 0$	remainder	42315
$8^1 = 8$	$42315 \div$	$32768 = 1$	remainder	9547
$8^2 = 64$	$9547 \div$	$4096 = 2$	remainder	1355
$8^3 = 512$	$1355 \div$	$512 = 2$	remainder	331
$8^4 = 4096$	$331 \div$	$64 = 5$	remainder	11
$8^5 = 32768$	$11 \div$	$8 = 1$	remainder	3
$8^6 = 262144$	$3 \div$	$1 = 3$		

Reading from the top we have: 122513_8 (Ans.)

1.5.8.Octal to Decimal

Typically, one would only obtain the decimal equivalent of an octal number using the sum of powers method.

Example A1.14

Convert 122513_8 to decimal.

$$\begin{aligned}
 122513 &= (3 \times 8^0) + (1 \times 8^1) + (5 \times 8^2) + (2 \times 8^3) + (2 \times 8^4) + (1 \times 8^5) \\
 &= (3 \times 1) + (1 \times 8) + (5 \times 64) + (2 \times 512) + (2 \times 4096) + (1 \times 32768) \\
 &= 3 + 8 + 320 + 1024 + 8192 + 32768 \\
 &= 42315 \text{ (Ans.)}
 \end{aligned}$$

1.5.9.Octal to Binary

The conversion from octal to binary is again a special case, and very simple. Each octal digit can be directly converted to a 3 bit binary number.

Example A1.15

Convert 62701_8 to binary.

Octal	6	2	7	0	1
Bin	110	010	111	000	001

Answer 110010111000001_2

1.5.10. Binary to Octal

Similarly, the conversion from binary to octal is carried out directly in much the same way as the conversion to hexadecimal, but using groups of three binary digits starting at the least significant digit.

Example A1.16

Convert 1011010110111101010_2 to octal.

Bin	1	011	010	110	111	101	010
Octal	1	3	2	6	7	5	2

Answer 1326752_8 .

1.6. Binary and Hexadecimal Fractions

Binary and hexadecimal fractions follow the same mathematical rules as decimal arithmetic, even though their representations may at first appear a little strange.

1.6.1. Binary Fractions

If we recall from Chapter 2:

Table 2.1 Powers of 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

then consider the negative exponents in base 10.

Table 10.1 Negative Powers of 10

10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
0.0000001	0.000001	0.00001	0.0001	0.001	0.01	0.1

What these numbers represent is simply 1 divided by the positive result of the exponent. For example $10^2 = 100$, so $10^{-2} = 1 \div 100 = 0.01$.

In exactly the same way, the binary fractional numbers are represented, so

Table 10.2 Negative Powers of 2

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
$1 \div 2$	$1 \div 4$	$1 \div 8$	$1 \div 16$	$1 \div 32$	$1 \div 64$	$1 \div 128$

0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125
-----	------	-------	--------	---------	----------	-----------

The simplest way to calculate the binary fraction of a decimal number is by constantly multiplying the decimal number by 2 and recording, then discarding the digit to the left of the decimal point.

Example A1.17

Find the binary equivalent of 0.12.

0.12 x 2 = 0.24	Record and discard	0
.24 x 2 = 0.48	“	0
.48 x 2 = 0.96	“	0
.96 x 2 = 1.92	“	1
.92 x 2 = 1.84	“	1
.84 x 2 = 1.68	“	1
.68 x 2 = 1.36	“	1
.36 x 2 = 0.72	“	0
.72 x 2 = 1.44	“	1
.44 x 2 = 0.88	“	0
.88 x 2 = 1.76	“	1
.76 x 2 = 1.52	“	1
.52 x 2 = 1.04	“	1
.04 x 2 = 0.08	“	0
.08 x 2 = 0.16	“	0
.16 x 2 = 0.32	“	0
.32 x 2 = 0.64	“	0
.64 x 2 = 1.28	“	1
.28 x 2 = 0.56	“	0
.56 x 2 = 1.12	“	1

Notice that this last operand will leave us with 0.12, which is the figure we started with. We are therefore going to constantly repeat this whole series of calculations.

Therefore $0.12_{10} = 0.00011110101110000101_2$ recurring (Ans.)

Notice that for this simple decimal figure there is *no exact binary equivalent!* The solution given runs to 20 bits, and has produced a recurring solution. The important point to be recognized from this is that it is often impossible to produce an exact binary representation of a fractional decimal number. Also keep in mind that the ‘.’ between the leading 0 bit and the first fractional bit (in this case a ‘0’ bit) is the binary point, *not* a decimal point.

1.6.2.Hexadecimal Fractions

Hexadecimal fractions may be calculated by a similar method to the one shown above for binary, but using 16 as a multiplier, and not forgetting that 10–15 are represented as A–F.

Example A1.18

Find the hexadecimal equivalent of 0.12_{10} .

.12 x 16 =	1.92	Record and discard	1	= 0x1
.92 x 16 =	14.72	“	14	= 0xE
.72 x 16 =	11.52	“	11	= 0xB
.52 x 16 =	8.32	“	8	= 0x8
.32 x 16 =	5.12	“	5	= 0x5

Again, this result is recurring.

$$0.12_{10} = 0x1EB85 \text{ recurring (Ans.)}$$

As a final proof, take the result of the binary calculation, and apply the method for converting binary to hexadecimal, so:

Bin	0001	1110	1011	1000	0101
Hex	1	E	B	8	5

This shows that once a fraction has been found in either the binary or hexadecimal system, it may be simply converted to the other system.

1.7. Exercises

Note: You should be able to perform all these calculations using pen, paper and brain only! You should not use a calculator for any of them.

- Convert from decimal to octal
(a) 212 (b) 399 (c) 42 (d) 1000
- Convert from decimal to octal
(a) 212 (b) 377 (c) 42 (d) 1000
- Convert the following hexadecimal numbers into decimal
(a) AF (b) 2BC (c) BABE (d) D9E5
- Convert the following decimal numbers to hexadecimal
(a) 100 (b) 328 (c) 10000 (d) 73601
- For the decimal numbers 1024, 27, 1444 and 511
(a) Determine the highest power of 2
(b) Convert into binary

6. Convert into binary
(a) 26.23871 (b) $5 \frac{23}{64}$ (c) $17 \frac{3}{8}$
7. Convert into decimal
(a) 11101101.111 (b) 111.010101 (c) 10001.10001
(d) 101101 (e) 0.101101 (f) 11.010
8. Convert the following binary numbers to (a) BCD and (b) octal
(a) 11001 (b) 1010110 (c) 1010010011 (d) 110001.00101
9. Convert the following BCD numbers into hexadecimal
(a) 01111001 (c) 01100010000100000001
(b) 001101101000 (d) 000110001001010000110001
10. Convert the hexadecimal numbers 0xAB and 0x45 to
(a) Binary (b) Decimal (c) Octal
11. (a) Determine the smallest and largest hexadecimal numbers that can be represented in a 16 bit system.
(b) Calculate the number of different analogue values that may be represented.
11. Convert from binary into hexadecimal
(a) 110101011011 (b) 101100111000 (c) 101101100110111
12. Convert from hexadecimal into binary
(a) FED (b) 4AD (c) BC5