Theory and Design of Digital Computer Systems

Douglas Lewin

econd Edition

LOW-PRICED EDITION

E L B S

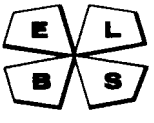
Douglas Lewin

Professor of Digital Processes, Brunel University

Theory and Design of Digital Computer Systems

Si quid novisti rectius istis,
Candidus imperti; si non, his utere mecum.
HORACE

'Now, brother, if a better system's thine,
Impart it frankly, or make use of mine.'



The English Language
Book Society *and* Nelson

To Will, my Father – a craftsman
who taught me engineering.

Thomas Nelson and Sons Ltd
Nelson House Mayfield Road
Walton-on-Thames Surrey KT12 5PL

116-D JTC Factory Building
Lorong 3 Geylang Square Singapore 14

PO Box 18123 Nairobi Kenya

Thomas Nelson (Hong Kong) Ltd
Watson Estate Block A 13 Floor
Watson Road Causeway Bay Hong Kong

Thomas Nelson (Nigeria) Ltd
8 Ilupeju Bypass PMB 21303 Ikeja Lagos

First published in Great Britain by Thomas Nelson and Sons Ltd, 1972
Second edition 1980

ELBS edition first published 1977
ELBS edition of second edition 1980

ISBN 0 17 771127 2
NCN 5905 42 0

Phototypeset by Tradespools Ltd,
Graphic House, South Parade, Frome, Somerset

Printed in Great Britain by A. Wheaton & Co. Ltd., Exeter

Theory and Design of Digital Computer Systems

Some other ELBS low-priced editions

Betts	SIGNAL PROCESSING, MODULATION AND NOISE	*Hodder & Stoughton*
Bryan	CONTROL SYSTEMS FOR TECHNICIANS	*Hodder & Stoughton*
Fox and Mayers	COMPUTING METHODS FOR SCIENTISTS AND ENGINEERS	*Oxford University Press*
Gosling	A FIRST COURSE IN APPLIED ELECTRONICS	*Macmillan*
Havill and Walton	ELEMENTS OF ELECTRONICS FOR PHYSICAL SCIENTISTS	*Macmillan*
Jeffrey	MATHEMATICS FOR ENGINEERS AND SCIENTISTS	*Nelson*
Lewin	LOGICAL DESIGN OF SWITCHING CIRCUITS	*Nelson*
Martin	NUMERICAL CONTROL OF MACHINE TOOLS	*Hodder & Stoughton*
Meadows	TECHNICIAN ELECTRONICS 2	*Cassell*
Ramshaw	POWER ELECTRONICS	*Chapman & Hall*
Schwarzenbach and Gill	SYSTEM MODELLING AND CONTROL	*Edward Arnold*
Scroggie	FOUNDATIONS OF WIRELESS AND ELECTRONICS	*Newnes–Butterworths*
Spain	VECTOR ANALYSIS	*Van Nostrand (U.K.)*
Turner	MODERN APPLIED MATHEMATICS	*Hodder & Stoughton*
Williams	NUMERICAL COMPUTATION	*Nelson*

Preface

Knowledge: A little light expels much darkness
Bahya ibn Paquda, Duties of the Heart.

Since this book was first published there have been major advances in LSI technology, in particular the developments in microprocessors and semiconductor memory. However, though the emergence of cheap and reliable microcomputers on a chip will certainly revolutionize computer systems architecture, the basic structure of the microcomputer itself still follows very closely the original principles of digital computer design. In fact in some respects it is rather like going back to the halcyon days of first generation computers!

Consequently, the fundamental principles described in the original edition of this book are still very relevant, and some aspects, such as machine-code programming, micro-programming and interfacing, have acquired even greater importance.

It is important to realize that the computer in its microcircuit form, has now become a system component which can be used to construct complex computer architectures. It is this aspect coupled with the availability of cheap semiconductor memory, that will change the face of computer engineering. The concepts of distributed and parallel processing, up to now barely being economically viable, will soon become standard practice. Moreover the structure of main frame computers as we know them today will drastically change, being replaced by dedicated microcomputer configurations performing both hardware and system software functions.

It is from this point of view that the second edition of the book has been written. The fundamental aspects of digital computers have been retained whilst considerable emphasis has been placed on parallel and distributed computing systems, including associative processing and the implementation of microcomputer based systems. Each chapter has been thoroughly revised and updated and the contents, where relevant, have been reorientated to take into consideration the effect of semiconductor technology. In particular important topics like stack processing, pipelining, distributed processing, interfacing, interrupt procedures and semiconductor storage systems have been completely revised and extended to include current thinking in these areas.

My sincere thanks must be recorded to all those readers of the first edition who spurred my enthusiasm to undertake this revision and to Molly Richardson, Betty Clark and Sue Lovett who unscrambled my manuscript into readable text.

Contents

1

The stored program principle

1.1 Introduction

There are three basic types of computer currently in use – the **analogue computer**, the **digital computer**, and a combination of both called the **hybrid computer**. In this book we shall be concerned primarily with the digital computer, but before we start it is worthwhile describing the characteristics of all three types.

The analogue computer represents the variables (and constants) in its calculations by physical quantities (usually voltage and time), hence the name 'analogue'. The slide rule is a very simple example, where 'length' is used to represent the actual values in a calculation. The accuracy of such calculations is of course limited by the accuracy with which we can measure the physical quantities involved. Usually the computing reference voltage is ± 10 V, and voltmeters, oscilloscopes and X–Y plotters are used to measure and record the values of the variables, generally to within an accuracy of 0·1% to 1%.

The solution to a mathematical or systems problem is obtained by setting up an analogue of the mathematical equations (or by simulating its transfer functions) using operational amplifier circuits functioning as adders, sign changers, integrators, and so on.[1] Thus each integration or addition, etc., in an equation is performed simultaneously by separate operational amplifiers working in parallel. Consequently the answer is in a continuous form; that is, the analogue computer produces a general solution to an equation which is normally displayed as a graph of voltage against time. The time required to produce a solution depends on the problem, but the computer can be suitably time-scaled (for example to allow for the response of the output equipment) to work either in **machine** or **real-time**. This has considerable advantages particularly in real-time problems when actual equipment can be included in a simulation. Another advantage is the rapport that exists between the designer and the machine, since the parameters of a problem may be easily changed by adjusting potentiometers and the results observed instantaneously.

The digital computer, on the other hand, represents its variables in a quantized or **digital** form. Thus numbers must be represented by using a discrete state or condition to represent each symbol (0–9 for decimal numbers). For example, in a decimal counting system (a car mileometer for example) gear wheels with ten teeth may be used to represent a decade, each cog corresponding to a symbol. A complete revolution

Table 1.1
Examples of the binary notation

Decimal radix of 10						*Binary radix of 2*										*Octal radix of 8*				
10^2	10^1	10^0	$\cdot$	10^{-1}	10^{-2}	10^{-3}	2^6	2^5	2^4	2^3	2^2	2^1	2^0	$\cdot$	2^{-1}	2^{-2}	2^{-3}	8^2	8^1	8^0	$\cdot$	8^{-1}	8^{-2}
100	10	1	$\cdot$	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	64	32	16	8	4	2	1	$\cdot$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	64	8	1	$\cdot$	$\frac{1}{8}$	$\frac{1}{64}$
		0	$\cdot$	1	2	5							0	$\cdot$	0	0	1			0	$\cdot$	1	0
		0	$\cdot$	2	5	0							0	$\cdot$	0	1	0			0	$\cdot$	2	0
		0	$\cdot$	5	0	0							0	$\cdot$	1	0	0			0	$\cdot$	4	0
		1	$\cdot$	0	0	0							1	$\cdot$	0	0	0			1	$\cdot$	0	0
		2	$\cdot$	0	0	0						1	0	$\cdot$	0	0	0			2	$\cdot$	0	0
		3	$\cdot$	0	0	0						1	1	$\cdot$	0	0	0			3	$\cdot$	0	0
		4	$\cdot$	0	0	0					1	0	0	$\cdot$	0	0	0			4	$\cdot$	0	0
		5	$\cdot$	0	0	0					1	0	1	$\cdot$	0	0	0			5	$\cdot$	0	0
		6	$\cdot$	0	0	0					1	1	0	$\cdot$	0	0	0			6	$\cdot$	0	0
		7	$\cdot$	0	0	0					1	1	1	$\cdot$	0	0	0			7	$\cdot$	0	0
		8	$\cdot$	0	0	0				1	0	0	0	$\cdot$	0	0	0		1	0	$\cdot$	0	0
		9	$\cdot$	0	0	0				1	0	0	1	$\cdot$	0	0	0		1	1	$\cdot$	0	0
	1	0	$\cdot$	0	0	0				1	0	1	0	$\cdot$	0	0	0		1	2	$\cdot$	0	0
1	0	0	$\cdot$	0	0	0	1	1	0	0	1	0	0	$\cdot$	0	0	0	1	4	4	$\cdot$	0	0

of a gear wheel causes the next gear wheel to enmesh so producing the effect of a carry. To perform the same task electronically we would need either a ten-state device, such as a decimal counter, or a specially constructed device using, in the simplest sense, ten on/off switches each connected to a lamp to represent one decade. As naturally occurring ten-state devices are very rare, and when specially made tend to be very expensive in components, it would appear obvious to use a number system with fewer symbols. Consequently in electronic digital computers the **binary system**, employing the two symbols 0 and 1 only, is used to represent numbers. This is a convenient and economic engineering solution since there are numerous examples of simple two-state devices (switches – on/off; transistors – conducting/cut off) which may be used to represent the symbols. Number systems may be defined mathematically in terms of the polynomial:

$$N = a_n q^n + a_{n-1} q^{n-1} + \cdots + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + \cdots + a_{-m} q^{-m}$$

where N is a positive real number, q a positive integer **radix**, and a represents the symbols. Thus, for example, the decimal number 27·5 may be expressed in terms of this polynomial as:

$$(27{\cdot}5)\ \text{decimal} \equiv 2 \times 10^1 + 7 \times 10^0 + 5 \times 10^{-1}$$

and

$$(27{\cdot}5)\ \text{binary} \equiv 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$$

Table 1.1 shows further examples of the binary system of notation; note that the binary (or radix) point occurs after the last positive index of the radix, that is, q^0. As a consequence of using the binary notation the accuracy of a calculation depends on the number of binary digits (bits) used to represent the variables. Using a 10-bit **word** we can expect an accuracy of 1 part in 10^3, the accuracy may be increased by using more bits, but one must be careful not to exceed the accuracy of the original data. It is pointless expressing input data, accurate to only 2%, to 1 part in 10^4!

We have said that the analogue machine works in a parallel mode producing an instantaneous solution. In contrast the digital machine works **sequentially**. Thus integrations which are performed autonomously in separate units in an analogue machine must be carried out one after the other in a digital computer. Furthermore the result is a numerical one, giving a particular solution to an equation rather than a general one. Thus to produce a general solution to an equation would require many iterations of the digital computing routine. Consequently in some problems, particularly real-time applications, the digital computer is too slow!

When speed and accuracy are required the hybrid computer is used; this incorporates some of the advantages of analogue and digital computers in one machine. In particular, iterative operations may be per-

formed, enabling a complete family of curves to be produced, by automatically changing the parameters of an equation using digital control techniques. Alternatively, analogue and digital computers may be used in conjunction, with the analogue machine providing overall control and calling upon the digital machine for function and delay generation, parameter changing, and so on. An alternative approach is to use a digital computer to simulate the operations normally executed by the analogue machine but to employ analogue sub-routines to perform integration or the solution of differential equations when high speed is required, as for example in real-time working. Another advantage of hybrid operation is that the preparation time required to program a problem (solutions are *patched* in an analogue computer by interconnecting individual sub-units) can in general be reduced. However, it is the author's contention that hybrid computer systems are a temporary means of overcoming the present limitations of digital computers; in the future when digital computers become faster (including fast storage systems) large analogue and hybrid computers may well become obsolete.

Digital computers are now being applied in all branches of technological and commercial endeavour. In particular, computers have been used to manufacture ice-cream, gas, and steel; to control road, rail, and air traffic; to set up newspaper type; to supervize stock control and insurance records; and to design engineering systems. Moreover the basic principle of digital computers, the **stored program concept**, has been employed in the design of many special purpose machines, such as telephone switching networks, and TV studio lighting.

More recently the development of cheap LSI microprocessors and microcomputer systems has revolutionized the design of digital systems. Computer techniques are now freely available to all engineers, who have employed them prodigiously in many different applications, particularly instrumentation and real-time control. Moreover, cheap programmable calculators, multi-function digital watches and even 'do it yourself' computer kits are now available for the public at large.

The object of this book is to explain the theory and design philosophy of digital computer systems. Not necessarily so as to enable people to design computers (very few will do this) but with the broader objectives of using computers or computer-like machines as modules in a large digital system. To achieve these objectives it is essential to know how to use the machines (the **software** design) as well as to understand their engineering and components (the **hardware** design).

1.2 Basic principles

The fundamental principles of organizing a digital computer were first set forth by Babbage[2] in the early part of the nineteenth century. As a result of building calculating machines to compute mathematical tables, Babbage conceived the idea of an Analytical Engine and, though it was never built, laid the foundation for modern automatic computing

machines. These ideas were later extended, though not changed in principle, by Von Neuman[3] who applied them to the design of an actual machine.

A digital computer consists of the following functional elements:

A **store** or memory for numbers (*operands*) and instructions.
An **arithmetic unit**, where arithmetical operations are performed on the operands.
The **control unit**, a device for controlling the logical operations of the machine, and causing them to take place in the correct sequence.
The **input/output unit**, used to transfer data into and out of the computer store.

Figure 1.1 shows a block schematic for a typical digital computer.

In general a numerical problem is solved, using a digital computer, by first breaking down the calculation into a number of discrete arithmetic operations (such as add, subtract) which are performed on the binary operands. These operations, together with any necessary organizing functions, such as input, output, register transfers, and so on, are then arranged to form a **program** of instructions for the computer. This program, suitably coded in a binary form, is written into the computer store using the input and control units. Instructions are read down from the store in sequence and obeyed, again under the action of the control unit, using the arithmetic unit as and when required. The store contains both program and data, plus 'working-space' and storage for results. The final operation is to output the results of the calculation via the output unit. Note the similarity between machine and manual computation. In the latter case a desk calculating machine could be regarded as the arithmetic unit, books of tables become the computing procedure and note-pads the store, and the control unit would of course be the human operator.

The philosophy of storing the program instructions as well as the operands, that is, the stored program concept, is the basic reason for the power of the digital computer. This is because, as we shall see later, the instructions can be treated as data and arithmetic operations may be performed on them in the normal way.

1.3 Instruction and number representation

In a digital computer the program instructions and operands (variables and constants) are stored together in the same storage unit. Each location in the store contains the same number of bits (called a **computer word**) and is allocated an absolute, uniquely identifiable address. The computer words, the length of which can vary from 8 to 32 bits for practical machines, are used to represent *both* the computer instructions and the data, see Figs 1.2 and 1.3. In the simplest case the **instruction word** would be divided up into an **order** and an **address** section plus some

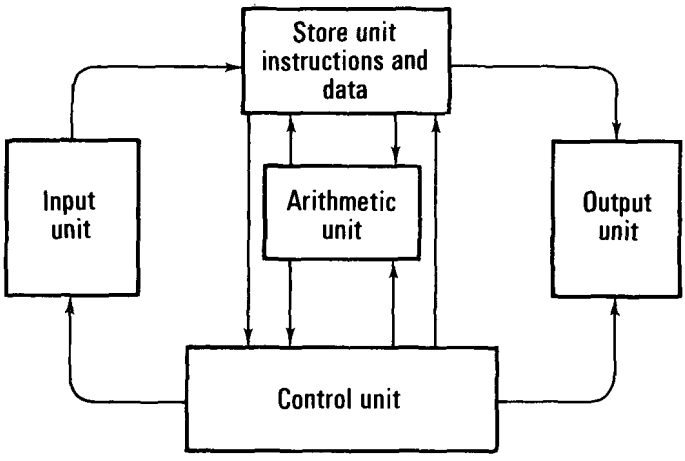
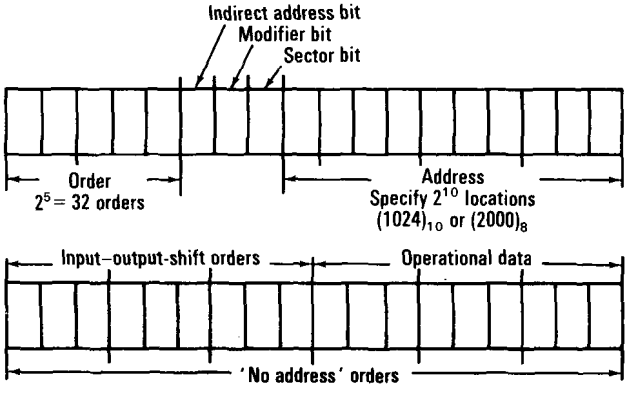


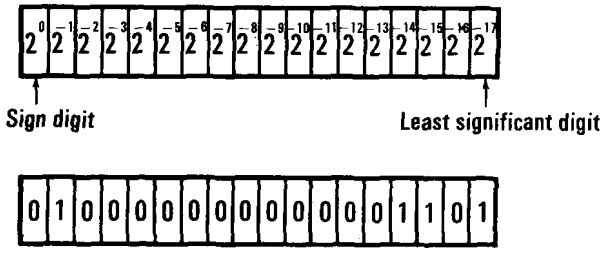
Figure 1.1 Computer block diagram.



(a) Address specified (SA) instruction.

(b) Input–output-shift (IOS) and zero-address instructions.

Figure 1.2 Instruction formats.



(i) Shift contents of accumulator 13 places left.
(ii) Integer 65549×2^{-17}.
(iii) Fraction 0·50009918.

Figure 1.3 Operand words.

control bits. The order part of the word has sufficient bits to allocate a unique binary or octal code to all the **machine-code orders**, likewise the range of the address section is such as to enable most of the store locations to be specified directly. Assuming an 18-bit wordlength, we can address 2^{10} $(=1024)$ store locations, and represent up to 2^5 $(=32)$ machine-code orders. The function of the three control bits, called the **indirect address**, **modifier**, and **sector** bits respectively, will be explained in the next chapter.

Numbers are generally represented in the computer using **fixed-point binary fractions**, such that $-1 \leqslant x < 1$, where x is the binary number; negative numbers normally take the **2's complement form** with the most significant digit indicating the sign (see Chapters 2 and 5). For representing instructions, however, the pure binary notation is too cumbersome to use. This is because of the large number of digits needed to represent a number and the necessity to convert from decimal to binary, and vice versa, when writing instructions or inspecting stored program in the computer. Consequently, the **octal** system of notation is used, which, since it is based on a radix of eight, is easily converted to binary and has the advantage of requiring fewer digits for number representation. For example:

$$(1467)_{10} \equiv (10110111011)_2 \equiv (2673)_8$$

The octal number 2673 may be represented in polynomial form as:

$$2 \times 8^3 + 6 \times 8^2 + 7 \times 8^1 + 3 \times 8^0 = (1467)_{10}$$

The conversion from octal to binary is easily accomplished by writing down the pure binary equivalent of each octal digit, for example:

$$\begin{array}{rl} & 2\ \ 4\ \ 6\ \ 6\ \ 7 \\ (24667)_8 = & (10\ 100\ 110\ 110\ 111)_2 \end{array}$$

The reverse process of converting from binary to octal follows directly from above.

A typical, but minimal, set of machine-code orders is shown in Table 1.2, the orders are divided into three groups consisting of:

(a) those orders which require the address of a store location to be specified, for example in the add, subtract orders, etc.;

(b) orders which do not require any form of address (ZA orders) and hence all 18 bits may be used to specify an order; examples of this type of order are the stop instruction, exchanging register contents, and incrementing registers;

(c) orders which do not require a store address but do need additional information to be specified (IOS orders), for example the shift orders which must include the number of places (n) to be shifted, and the input/output instructions.

Table 1.2
Typical machine-code order set

(a) Specified address instructions (SA)

Code in octal	*Mnemonic*	*Description of order*
00		Reserved for ZA orders
01		Spare
02		
03		
04		
05		
06		
07		
10		Reserved for IOS orders
11		Spare
12		
13		
14	ADDM	Add contents of address specified to modifier
15	FETM	Fetch contents of address specified to modifier
16	STRM	Store contents of modifier in address specified
17	LINK	Store contents of instruction register in address specified and jump to address specified +1
20	ADD	Add contents of address specified to accumulator
21	SUB	Subtract contents of address specified from accumulator
22	FET	Fetch contents of address specified to accumulator
23	STR	Store contents of accumulator in address specified
24	COL	AND accumulator with contents of address specified
25	OR	OR accumulator with contents of address specified
26		Spare
27		
30	JMP	Jump to address specified
31	JMPN	Jump to address specified if accumulator negative, otherwise take next instruction in sequence
32	JMPP	Jump to address specified if accumulator positive, otherwise take next instruction in sequence
33	JMPO	Jump to address specified if accumulator zero, otherwise take next instruction in sequence
34	JMPM	Jump to address specified if modifier zero, otherwise take next instruction in sequence
35	MULT	Multiply accumulator by contents of address specified, resulting 36-bit product contained in accumulator and X-register
36	DIV	Divide 36-bit dividend, contained in accumulator and X-register, by contents of address specified
37	EXCO	Perform exclusive OR function between accumulator and contents of address specified

(**b**) Zero address instructions (ZA)

Code in octal	*Mnemonic*	*Description of order*
000000	STOP	Stop the computer
000001	COML	Form 2's complement of accumulator
000002	NOOP	No operation
000003	ENI	Enable interrupts
000004	INI	Inhibit interrupts
000005	EXAM	Exchange contents of accumulator and modifier register
000006	INCM	Add +1 to modifier register
000007	EXAC	Exchange contents of accumulator and X-register
000010	EXIO	Exchange contents of accumulator and input/output register
000011	CLRA	Clear accumulator
000012 ⋮ 017777		Spare

(**c**) Input/output/shift instructions (IOS)

Code in octal	*Mnemonic*	*Description of order*
200	SLL	Logical shift left of the accumulator *n* places
201	SRL	Logical shift right of the accumulator *n* places
202	SLA	Arithmetic shift left of the accumulator *n* places
203	SRA	Arithmetic shift right of the accumulator *n* places
204	EAS	End-around shift of the accumulator *n* places
205	INA	Input word to accumulator
206	OTA	Output word from accumulator
207 ⋮ 217		Spare