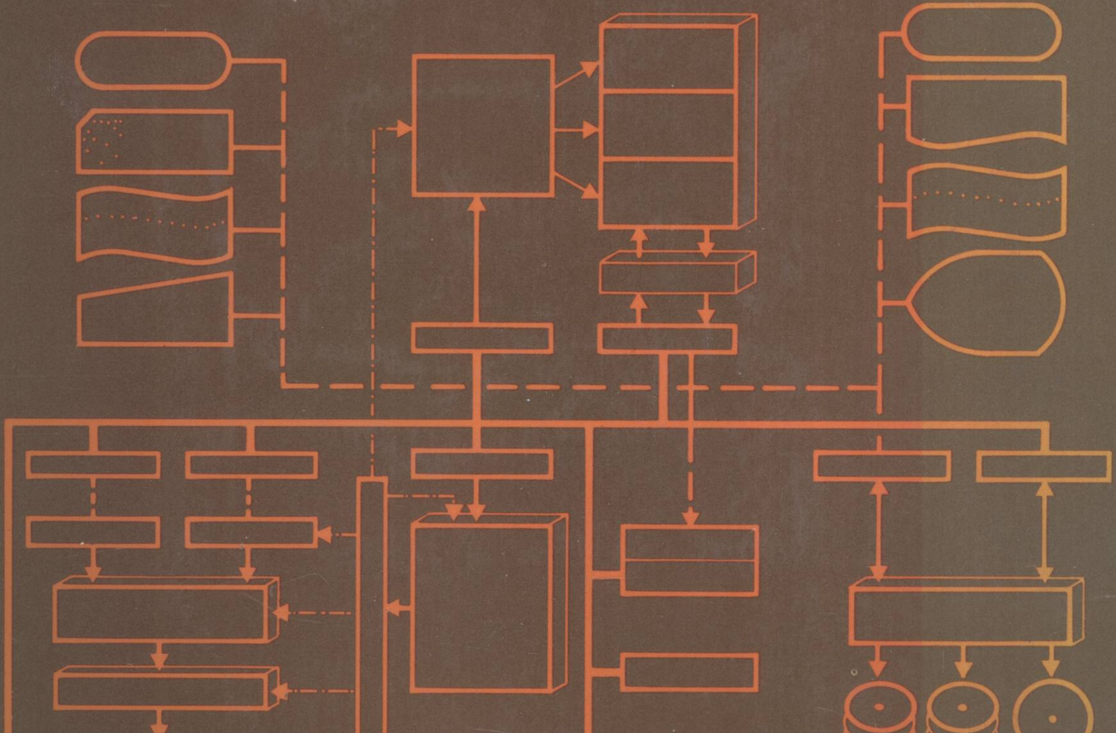
An Introduction to Computer Hardware

Martin Cripps



An Introduction to Computer Hardware

Martin Cripps

Department of Computing and Control
Imperial College of Science and Technology

Edward Arnold

First published in 1977 by
Edward Arnold (Publishers) Ltd.
25 Hill Street, London W1X 8LL

ISBN: 0 7131 2572 1

For Margot and Norman

Printed in Great Britain by the Pitman Press, Bath

Preface

Computing is one of the most important - some might say the most important - branches of science and technology. In reality it is a technology and a servant to all sciences and technologies, for it is useless to have a theory of computing unless there is something useful to compute and machinery on which to perform the computation.

This is an introductory book on the important topic of computer hardware. It is vital for all users and programmers of computers to appreciate the capabilities and restrictions of the machine which will execute their programs. The basic technique of modern computing is systems engineering, that is the software and hardware are engineered into a total system. A computing scientist or programmer requires a thorough understanding of computer hardware if he is to make the best use of it. The aim of this text is to present the material necessary for a good understanding of the design, construction and operation of computer hardware in a straightforward and logical fashion.

There are some excellent works on the detailed design and hardware of computers for those with a good technological or electronics background. However, students commence the study of computing at school, as undergraduates at university or sometimes as graduates in other subjects taking "conversion" courses. Their background frequently contains no more than O-level or A-level physics and little engineering or technology. This book was produced in the absence of any cheap, well-structured texts and was based on the notes produced for the undergraduate and postgraduate courses which I developed and presented at Imperial College. The book should also prove useful to anyone else not directly studying computing, who wishes to remove any gulf seperating him or her from the mystique of the "pretty coloured boxes with the flashing lights".

Computing, in modern terms, is thirty years old this year, although most of the ideas were foreshadowed by Charles Babbage and Ada Augusta, Countess of Lovelace, between 1833 and Babbage´s death in 1871. In 1833 Babbage conceived the idea of his "analytical engine", which was to include stored programs, a store, an arithmetic unit which he called the "mill" and input-output mechanisms. To be constructed using steel gears and to be powered by steam, it was a conception far beyond the technology of the time and was never finished. However, from Babbage's drawings and the detailed descriptions and suggestions for usage by his colleague the Countess of Lovelace, it is apparent that their ideas were both brilliant and correct. Their foresight even went as far as to point out that the analytical engine (like all subsequent computers) "has no pretensions whatever to originate anything".

A computer can do whatever we know how to order it to perform, and nothing further. This is a point which all should hold firmly in mind when discussing computers.

My approach is to divide the computer into logically separate parts and to study them individually before putting them together to create a complete machine. An introduction to the description of logic and design precedes the main body of text. A description of the technology used to implement the designs ends the book.

The assistance of Elsbeth Lindner who read the text and suggested valuable improvements is gratefully acknowledged. The comments, mostly polite and helpful, from students who have attended my courses have contributed to the accuracy and suitability of the text and are also much appreciated.

Martin Cripps,
London,
January 1977.

Contents

1 Basic Concepts

1.1 THE STORED PROGRAM MACHINE

The basic cycle of any computer is to *FETCH* instructions from a store and then to *EXECUTE* them. The instructions must be fetched in a predetermined sequence, so that they cause execution of the actions the user requires to solve his problem. They must contain sufficient information to determine these actions completely. Instructions need to contain some, or all, of the information in the form:

RESULT = *OPERAND {OPERATOR} OPERAND* *{NEXT INSTRUCTION}*
E.G. A=B+C, next instruction is at X.

No more information is ever required than these five "fields", though that information may have to be found by means of pointers to, or "addresses" in the store. It is usual for some of the fields to be implicit in the design of the machine, for instance, it is normal to assume that the next instruction is held in the store location immediately following that of the instruction being executed. Hence the next instruction field is implicit and only when a change to the sequential fetching of instructions is required is any information needed to tell the hardware (the electronics) to get the next instruction from a location other than the implied one.

The instruction to be performed, specified by the operator, sets the machine to add, multiply, compare, etc, the operands specified, and produce a result, which can then become an input operand for some later instruction. The most important feature of a computer is its ability to choose one of a number of different sequences of instructions, depending on results it has calculated previously.

To function in this way, a computer requires the following: a store to hold the instructions and any data or operands; a processing unit, often called an arithmetic and logic unit, to perform the operations such as adding or comparing; a control unit to ensure instructions and data are obtained from the store correctly, and that the proper sequences are maintained; and a set of input-output peripherals to communicate with the machine.

Computer hardware is essentially very simple. All the storage, control and processing elements can be constructed using a single electronic circuit containing only a few transistors, whereas a colour television requires many different circuits, using all kinds of components. With computers, the complexity comes not in the electronics or the circuits but in the way they are connected together. Fig.1.1 shows the connection of the five key elements described above and treated as "black boxes". This is the traditional way to view a computer. Each "box" can be looked at more or less independently and this will be done after a review of the basic circuits and the way they can be connected to give more complex units.

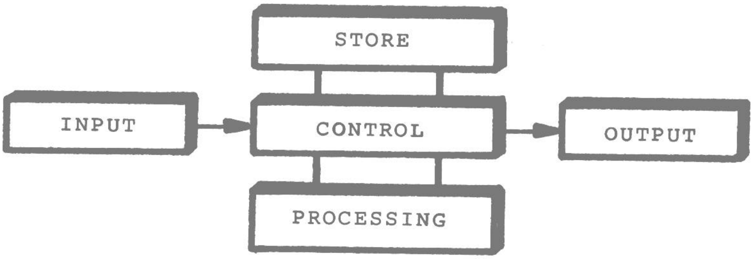


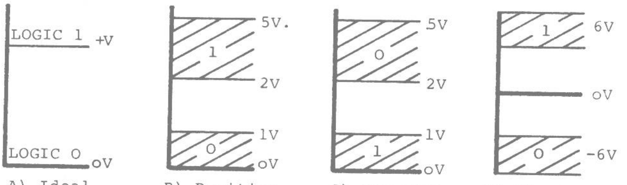
FIG. 1.1 *A SIMPLE COMPUTER STRUCTURE*

1.2 LOGIC CONVENTIONS

Signals which can take an infinite number of values or states are called analog signals and can be used, but as most computers use signals which can only have defined values or states, digital signals, these are used throughout this book. However, it should be borne in mind that both analog and hybrid computers are constructed as well as digital computers.

In most systems which process information in digital form, the signals are nominally two-valued or "binary" in nature. For example, information may be represented by a voltage or current which takes one of two values on a wire, or by a pulse of defined shape, which is either present or absent at a given time. In computers the two binary states are represented in different ways in different parts of the machine because of the physical nature of the devices which make up the machine. For a first discussion of computer logic it will be assumed that the states are defined by bands of voltage, with a forbidden region between them. Each band will be referred to by its nominal value, as in Fig.1.2

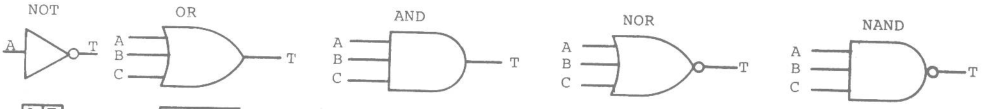
The more positive state has been arbitrarily chosen to represent the "1" or *TRUE* state: this is termed "positive" logic. The case where the "0" or *FALSE* state is the more positive and the true state more negative is called "negative" logic: the significance of this convention will become apparent further on. Also one of the levels has been chosen to be zero volts, which is obviously convenient as a switch being off gives zero volts and on, connected to a power supply, can give a positive voltage for the other state. If neither voltage is chosen to be zero volts, then the logic is called "bipolar". This is used for data communication, by the Post Office and others, as the "0" and "1" states are represented by voltages which are different from a failed state, such as a broken wire.



A) Ideal B) Positive C) Negative D) Bipolar

FIG. 1.2 LOGIC LEVELS AND NOMINAL VALUES

With the chosen definitions, switching arrangements could be devised, using relays or other types of switches, or using transistors in integrated circuits as in modern computers. It would obviously be inefficient to carry drawings of relays or transistor switch circuits through the whole design process, so logic "symbols" and "truth tables" are introduced to simplify drawing and to permit easy description of logic.



A	T
0	1
1	0

TRUTH TABLES

$T=A\neg$

A	B	C	T
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

$T=A\vee B\vee C$

A	B	C	T
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

$T=A\wedge B\wedge C$

A	B	C	T
0	0	0	1
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	0

$T=(A\vee B\vee C)\neg$

A	B	C	T
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$T=(A\wedge B\wedge C)\neg$

FIG. 1.3 SIMPLE LOGICAL ELEMENTS

The logical symbols, truth tables and Boolean expressions for all the simple logical elements are shown in Fig.1.3. The truth tables show the outputs which are produced for all possible different input combinations; the expressions are an algebraic description and are described later. The three basic elements are the inverter or NOT, the OR and the AND gates. The NOR and NAND gates, which are most common in computers because they happen to be easier to construct, are formed from OR and AND with an inverter following. As can be seen, an AND gate only produces a true output if *ALL* its inputs are true, so one input effectively acts as a gate to any others, shutting them off if it is false. An OR gate, on the other hand, acts as a union, combining inputs together, so that, if *ANY* input is true, the output is true.

The effect of positive and negative logic can now be seen. If the logic convention is changed, everything which represents "true" would represent "false" and vice versa. The physical circuit, which produced the NAND, would then produce a NOR, as can be shown by changing all 0's for 1's, and vice versa, in the truth tables. It is apparent that a single logic element, say a four input positive NAND, can also be made to perform as a NOR, by changing convention (to negative), and also as a NOT, by connecting all four inputs together (see the truth table). Even though a given design or logic diagram may call for a variety of gates, all the five basic ones can be "made" from just one. Storage elements can also be made from NAND and NOR gates.

1.3 BOOLEAN ALGEBRA, DEFINITIONS AND POSTULATES

Starting with the logical building blocks, complex logical functions could be designed by inspired guesswork, or experience, or a combination of both. This would not be very satisfactory, as there would be no checks on correctness of design, or on optimality in the design procedure. By using the logic of two-valued functions developed by the mathematician George Boole, we can produce formal design and optimisation techniques.

A Boolean variable "X" has two possible values, "0" and "1". These values are mutually exclusive. A Boolean function is determined when a relationship between two or more independant Boolean variables is given.

The following postulates (and notation) are adopted for addition, multiplication, inversion (complementation) and the logical functions in Boolean arithmetic and algebra. Each can occur in two forms, the second being the dual of the first. The dual of OR is AND and the dual of a variable is its complement.

INVERT	AND	OR	ADD	MULTIPLY
$\neg 0 = 1$	$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$0+0 = 0$	$0.0 = 0$
$0^{\urcorner} = 1$	$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$0+1 = 1$	$0.1 = 0$
$\neg 1 = 0$	$1 \wedge 0 = 0$	$1 \vee 0 = 1$	$1+0 = 1$	$1.0 = 0$
$1^{\urcorner} = 0$	$1 \wedge 1 = 1$	$1 \vee 1 = 1$	$1+1 = 0$*	$1.1 = 1$

*with a carry to the next bit.

1.4 ALGEBRAIC PROPERTIES AND BOOLEAN THEOREMS

Bearing in mind that Boolean variables can only take one of two possible values, the following algebraic properties of normal algebra also apply to Boolean algebra. Commutation implies that the order or sequence of variables has no effect on the value of an expression. Association implies that in sequences of only AND or of only OR functions, the placing of the parenthesis does not affect the result. The distributive property implies that an expression containing both AND and OR functions may be AND'ed out (multiplied out) with the AND taking precedence in a similar fashion to multiplication and addition in ordinary algebra.

COMMUTATIVE	$X_{\wedge}Y=Y_{\wedge}X$	$X_{\vee}Y=Y_{\vee}X$
ASSOCIATIVE	$X_{\wedge}(Y_{\wedge}Z)=(X_{\wedge}Y)_{\wedge}Z$	$X_{\vee}(Y_{\vee}Z)=(X_{\vee}Y)_{\vee}Z$
DISTRIBUTIVE	$X_{\vee}Y_{\wedge}Z=(X_{\vee}Y)_{\wedge}(X_{\vee}Z)$	$X_{\wedge}(Y_{\vee}Z)=(X_{\wedge}Y)_{\vee}(X_{\wedge}Z)$

The following provable theorems demonstrate the results of the logical functions on variables combined with a fixed "true" or "false", or with themselves or their complements. They can all be demonstrated by constructing the truth table, taking all possible combinations of true and false inputs and producing the output.

UNIT AND ZERO	$0_{\vee}X=X$	$1_{\vee}X=1$	$0_{\wedge}X=0$	$1_{\wedge}X=X$
IDEMPOTENCE	$X_{\vee}X=X$		$X_{\wedge}X=X$	
COMPLEMENTARITY	$X_{\vee}X^{\neg}=1$		$X_{\wedge}X^{\neg}=0$	
INVOLUTION	$(X^{\neg})^{\neg}=X$			

The absorption theorem shows how commonly-occurring patterns of variables are reduced to simpler forms by removing the redundancies occurring in the theorems above.

ABSORPTION	$X_{\vee}X_{\wedge}Y=X$	$X_{\wedge}X_{\vee}Y=X$
	$X_{\vee}X^{\neg}{}_{\wedge}Y=X_{\vee}Y$	$X_{\wedge}X^{\neg}{}_{\vee}Y=X_{\wedge}Y$

The exchange of the logical operators OR and AND is arranged by using De Morgan's theorem. It is easily shown to be the same as the convention change between positive and negative logic discussed earlier. By changing all true and false variables to their complements, the operator required is also changed to its *DUAL* (AND to OR, NAND to NOR and vice versa).

DE MORGAN	$\neg(X_{\vee}Y)=\neg X_{\wedge}\neg Y$	$\neg(X_{\wedge}Y)=\neg X_{\vee}\neg Y$

The absorption theorem and De Morgan's theorem are the basis for all minimisation techniques, the aim of which are to remove any redundancy from Boolean expressions. This is desirable, as each redundant AND or OR implies a redundant gate, hence some redundant components, and so more optimal solutions would result from the formal application of these theorems. Logical units of considerable complexity can be constructed using the basic logical functions and these complex units can then be used as "black boxes" for further design stages. To formalise the design techniques, all types of logic need to be considered. There are three types, namely combinational, synchronous sequential and asynchronous sequential, and they are diagrammatically represented in Fig.1.4. Combinational logic contains no storage, so the outputs at any time are only dependant on the inputs. There is no effect due to "history", as there is with sequential logic, which contains storage, so that the outputs depend on the inputs and on previous states which have been held. The difference between synchronous and asynchronous logic is that, whilst in the former a clock is used and changes are only acted on at predetermined times, so that stored states only alter at fixed times, in the latter changes can occur at any time and states may change at any time. Asynchronous sequential logic can give rise to "races" where two signals change and the order in which they change may not always be the same.

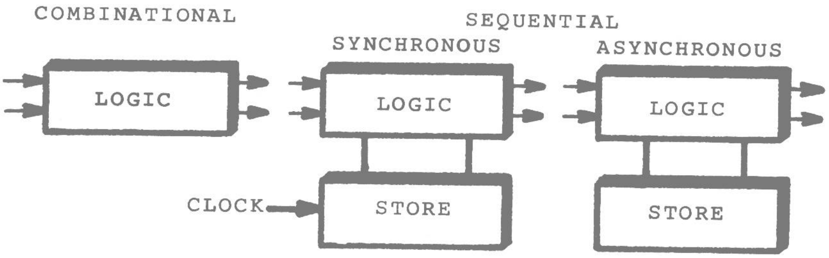


FIG. 1.4 LOGIC CIRCUIT FORMS

There are formal design techniques for all 3 types of logic, but the sequential logic design methods are complicated, seldom performed by hand and are well described in books on logic design. A simple example of one formal technique for combinational logic design is described to demonstrate the approach to such problems.

Starting from the truth table, various forms of Boolean expression can be produced, some of which are very useful, particularly the "canonical" or standard form.

If all elementary elements or their complements occur once only in each factor, then the expression is in canonical form. If the expression is in the form: $\{A\wedge B\wedge\neg C\}\vee\{\neg A\wedge B\wedge C\}\vee\{A\wedge\neg B\wedge C\}$, then it is in the *MINTERM* (or sum of products form), and if it is in the form: $\{A\vee\neg B\vee C\}\wedge\{\neg A\vee B\vee\neg C\}$, then it is in the *MAXTERM* (or product of sums form). Both these expressions are also canonical, whereas $\{A\wedge B\}\vee\{A\wedge\neg B\wedge C\}\vee\{\neg C\wedge D\}$ is a minterm form but not canonical.

1.5 KARNAUGH MAP DESIGN (K-MAP)

A K-map is the most common simple minimisation technique and is based on the layout of combinations of elementary variables such that only one elementary variable changes between adjacent squares on the map. This change corresponds to the $X \vee X^{\neg}=1$ function, so the variable may be removed without affecting the final function. As can be seen in Fig.1.5, the value of the squares 0,1,2 (equivalent to binary 000, 001, 010, etc.) is the value of a term of the canonical minterm form, hence its importance. This coding scheme which changes in one bit only is called the "Gray code".

A¬		A		
000 ABC	010	110	100 ABC	C¬
001	011	111	101	C
B¬	B		B¬	

A¬		A		
0000	0100	1100	1000	D¬
0001	0101	1101	1001	D
0011	0111	1111	1011	
0010	0110	1110	1010	D¬
B¬	B		B¬	

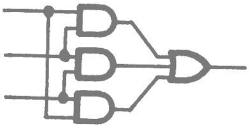
FIG. 1.5 THREE AND FOUR VARIABLE K MAPS

EXAMPLE: The truth table for a function is shown in Fig.1.6. The expression can be read off directly by an OR function between all the AND combinations, which are to provide a true (1) output. Alternatively, if there are fewer false indications than true required for output, then the method can be applied to them if an inversion is added, as the answer generated will be NOT what is required. The expression can be plotted onto a K-map directly and reduction is performed by combining adjacent squares into the fewest number of larger squares or rectangles. Each time two adjacent squares contain a 1, the function $X \vee X^{\neg}=1$ must occur, so the common variable is redundant at that point.

A	B	C	T
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$T = A^{\neg}{}_{\wedge}B_{\wedge}C_{\vee}A_{\wedge}B^{\neg}{}_{\wedge}C_{\vee}A_{\wedge}B_{\wedge}C^{\neg}{}_{\vee}A_{\wedge}B_{\wedge}C = A_{\wedge}B_{\vee}B_{\wedge}C_{\vee}C_{\wedge}A$ (reduced)

	A¬		A	
C¬	0	0	1	0
C	0	1	1	1
	B¬	B		B¬



Output is true if 2 of 3 inputs true

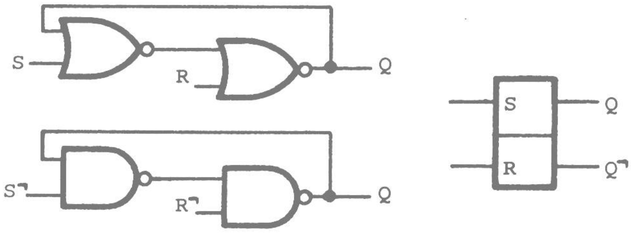
FIG. 1.6 EXAMPLE OF K MAP DESIGN

If a NAND only form is required, it can be obtained by applying De Morgan's theorem, as in the example, giving a NAND:NAND implementation instead of the NOT:AND:OR form. K-maps close on their edges, i.e. square 1000 is also adjacent to squares 0000, 1010. This technique can be used for up to 6 variables by hand; thereafter a computer is used. The better computer minimisation techniques are tabular and will handle any number of variables, the best known being the Quinne-McCluskey technique, which is well described in the literature. When plotting onto a K-map, it can be useful to use any outputs where you "do not care" whether the output is 1 or 0. They are plotted as # on the map and can be used, or not, at will.

So far only combinational logic has been described, where no previous states are relevant, but for a computer, storage will be required, as will circuits which can react depending on some past state or event.

1.6 BISTABLE CIRCUITS

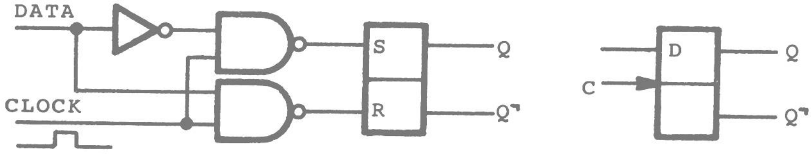
Bistable circuits can exist indefinitely in one of two stable, separate states. By connecting two NOR gates with feedback, as shown in fig.1.7, an element which depends on previous inputs as well as current inputs can be made. It will store one state or binary digit, often called a *BIT*, and is sometimes called a staticisor or a "flipflop". The Boolean expression for the circuit is: $Q=\neg\{R\vee\neg\{Q\vee S\}\}$ and the truth table shows how it reacts, depending on its previous state (P=previous state, Q=current state).



S	R	P	Q
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0?
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0?

FIG. 1.7 THE SET RESET {SR} BISTABLE

This "SR" bistable has one major drawback, which occurs when S=R=1. The final output Q, when S and R are both removed, is not determined. One can design so that this combination never occurs, but it is easier to have an element without this drawback. If only data is to be stored and the element is not required to be able to be set and reset independantly, then a "D-type" bistable is used. This has a single data input which is latched by a short clock pulse.



D	C	P	Q
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

FIG. 1.8 *THE D TYPE BISTABLE*

The D-type responds to a data input by retaining its value when the control line (or clock line) makes the transition from true to false. It responds on an "edge" and the circuit is shown with the truth table in Fig.1.8. This element is used to build data registers which hold many binary digits in parallel and hence a complete number.

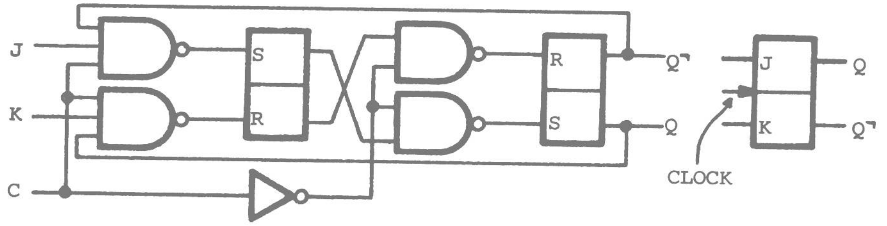


FIG. 1.9 *THE JK BISTABLE*

The "JK" or master-slave bistable is the one used most frequently when independent set and reset conditions are required. For instance, if a printer is to start on a given set of conditions and end on a different set, the JK of Fig.1.9 would be used to control it. The truth table for the JK is the same as for the RS except for the two undetermined states, which are arranged so that the output changes if the J and K inputs are both "1". Another useful property of the JK is that the output appears only when the clock is removed, so the JK acts as a delay of 1 clock pulse. It contains an RS within it, if one wants to use one in a design predominantly using JK's.

1.7 REGISTERS

A single state can be retained over time in a bistable, hence storing a group of bits in a parallel set of bistables, or register, as in Fig.1.10, will permit a number to be stored. Number representation is covered in more detail in Chapter 2, but as well as numbers, logical values, true or false, sometimes referred to as *FLAGS*, and characters will be needed. A single flag requires a single bit, but they are normally grouped together to give the same length *WORD* as for numbers. In most minicomputers the word length is 16 bits, as this is convenient for instructions, numbers and characters.

The normal number system used by people is the decimal system, using the digits 0-9 and columns for units, tens and hundreds. We are able to use other systems easily, for example the "duodecimal" (12) system for feet and inches, where columns represent twelves and "hundred and fourty-fours". Two systems are particularly important in computing because they work with powers of two, namely the "binary" and "hexadecimal" (16) systems.

Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	4	0100	4	8	1000	8	12	1100	C
1	0001	1	5	0101	5	9	1001	9	13	1101	D
2	0010	2	6	0110	6	10	1010	A	14	1110	E
3	0011	3	7	0111	7	11	1011	B	15	1111	F

From this it can be seen that each number represented by a sum of units, tens and hundreds in decimal has a similar binary pattern of units, twos, fours, eights, etc. and a hexadecimal coding of units, sixteens, and "two hundred and fifty-sixes". Conversion between the systems is easy, requiring only simple addition and subtraction; as an example, 677=1010100101=2A5. The binary and "hex" forms, containing sums of powers of two, can be stored easily in two-state stores. The example requires a ten bit register.

The standard code for characters is called the American standard code for information interchange or ASCII and uses seven bits to give 128 character codes, with an eighth bit available as a check bit. There are 32 control codes for such things as feeding lines, tabs and ringing a bell. The remaining 96 characters contain space and delete (all 1s) and 94 printable characters, which cover the decimal digits, the alphabet both upper and lower case, punctuation marks and special symbols.

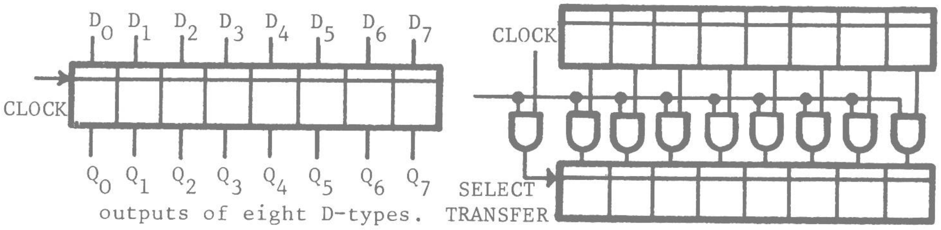


FIG. 1.10 A REGISTER AND REGISTER TRANSFER