

Preliminary Procedure

Remove all leads and modules from the CK353 deck. Insert the power plug into a main power supply outlet but do not switch on yet.

Discussion and Experimental Procedure

In the decimal number system there are ten digits, 0 to 9, the true significance of any digit depending on its position in the number: for example

Thousands	Hundreds	Tens	Units
2	4	3	1

really means

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

Remembering that $10^0 = 1$ we see that, counting the units column as No.0, the tens as No.1 and so on, the value of a digit is equal to that digit multiplied by ten (called the 'radix' of the system) raised to a power equal to the column number. In general a digit 'a' has a value A given by

$$A = a \times r^n$$

Where r = the radix (ten in this case)

and n = the column number.

Now imagine that we had only five digits available or in other words that the symbols 5, 6, 7, 8 and 9 did not exist. Our counting would have to be done with the only digits known to us, that is 0, 1, 2, 3 and 4 so that our number system would have a radix of five.

$$r = 5$$

and the values of 'a' would be restricted to

$$a = 0, 1, 2, 3 \text{ or } 4$$

This system would be called a 'quinary' system.

Then, for example, digit 2 in column No.3 has a value

$$2 \times 5^3 = (\text{in decimal equivalent}) 250$$

● Q1.1 What decimal value has a digit 3 in column No.2 in a quinary system?

In digital computer engineering it is not convenient, for many reasons, (see Practical Considerations) to employ devices to represent decimal or even quinary numbers as each device would have to be capable of taking up ten or five distinct states. Instead the number of states

is restricted to the smallest that will allow a change, that is two, and these are commonly denoted by digits 0 and 1.

This then is the 'binary' system whose radix is two.

● Q1.2 What decimal value has digit 1 in column No.5 of a binary system?

The first five columns in a binary system have decimal values $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$.

Thus the binary number 1 0 1 1 0 = decimal $1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 22_{10}$

The small suffix ten in this result is to indicate that the number is a decimal one. Sometimes confusion can arise otherwise, e.g.

101_{10} = one hundred and one in decimal
but $101_2 = 5_{10}$

Thus we should have written above

$$10110_2 = 22_{10}$$

You should have the answer 32 to Question 1.2. If not, read the last paragraph again and see where you went wrong.

Practical 1.1

Switch on the power to the CK353 deck. The power lamp on the deck glows red when power is on.

Now connect a lead from the output of toggle S1 to the input of indicator lamp L1 and toggle the switch.

● Q1.3 What state is the lamp in when the toggle is (a) up? (b) down?

In INTIKIT a lamp that is alight is taken to indicate a binary 1 state and one that is connected but not alight indicates a binary 0 state.

You should therefore find the following

Binary State	Lamp	Toggle
0	OFF	DOWN
1	ON	UP

As an exercise fill in the missing entries in the table on the following page.

Exercise 1.1

Decimal No.	Binary Equivalent
0	
1	0001
2	
3	1011
5	
15	
6	
10	
12	
9	
8	1101
14	0100
	0111

Ask your instructor if you have any problems with this exercise.

- Q1.4 How many binary columns are necessary to represent numbers up to 9?

Practical 1.2

The answer you should have is four. Connect leads from toggles S2, S3, S4 to lamps L2, L3, L4 respectively so that you now have four binary columns. Set up in increasing order all the decimal numbers from 0 to 15 on the toggles, observing the lamp indications as you go.

- Q1.5 How many different numbers can four binary digits represent?

Binary digit is usually abbreviated to 'bit'. A four-bit number can represent $2^4 = 16$ different numbers but the maximum number is $2^4 - 1 = 15$ because zero is included in the 16 possibilities.

Binary Decimal Code

Although computers use binary numbers internally, operators are more accustomed to interpreting decimal numbers. Thus when numbers have to be fed in via a keyboard or fed out to an electric typewriter or to decimal indicators it is convenient to use a system in which the individual decimal digits are separately represented by binary instead of as a whole. Such a system is called 'binary decimal' or 'binary-coded decimal'—BCD for short.

$26_{10} = 11010$ —'ordinary binary'

but $= \begin{matrix} 2 & 6 \\ \hline 0010 & 0110 \end{matrix}$ —BCD

Obviously, to be able to represent all digits up to 9 there must be four bits to each decimal column so two groups of four bits can represent all numbers from 0 to 99.

- Q1.6 How many bits are needed to represent 99 in ordinary (called 'pure') binary?

- Q1.7 Is pure binary more or less efficient than BCD in its usage of binary columns?

Practical 1.3

Connect four more toggles, S7, S8, S9 and S10 to their corresponding lamps so that you now have two groups of four bits.

Set up the BCD equivalent to the following decimal numbers: 0, 10, 25, 78, 99.

- Q1.8 In BCD is there any restriction on the bit patterns that may be used? If so, what is it?

Practical considerations & applications

In electronic digital computers using integrated circuits the binary states are represented by voltages whose range rarely exceeds 5 volts. Usually Binary 0 is represented by a voltage near zero and Binary 1 by one near +5V. There is always some variation in these voltages in practice and so the rule is made that any voltage between, say 0V and +0.4V will be called binary 0 and any between say +2.4V and +5V will be called binary 1. In these circumstances you can see that, even disregarding the problems of setting the nominal values of voltage to more than two levels it would be impossible to maintain them within non-overlapping bounds when the tolerances were accounted for. This is one reason why decimal number systems are not used.

Another is, as we shall see later on in Assignment 8 that binary arithmetic is very much simpler than decimal.

The BCD system is usually restricted to inputs and outputs but there have been several computers built which use it throughout for all arithmetic and other operations.

Further Reading

Discussion and Experimental Procedure

You saw in Assignment 1 that binary numbers comprise a collection of binary digits or bits, each of which can be either 0 or 1 and that these binary values can be represented in hardware by various means such as toggle switch positions and lamp states.

What we are now about to study are the operations that may be performed on one, two or more of these 'binary variables' which, although they were introduced to you as a means of representing binary numbers, actually have an independent existence and may be used to represent quite different things.

For instance 1 and 0 can represent any of the pairs of opposites shown in the following table.

'1'	'0'
TRUE	FALSE
IN	OUT
UP	DOWN
YES	NO
WET	DRY

In general binary variables can represent any pair of concepts in which the existence of one implies the non-existence of the other. Notice carefully the idea of 'opposite-ness' or 'NOT-ness'

TRUE is NOT FALSE
 UP is NOT DOWN
 1 is NOT 0
 0 is NOT 1
 etc., etc.

Practical 2.1

Plug in inverter (7404) module into any convenient position on the CK353 deck, aligning it lamps

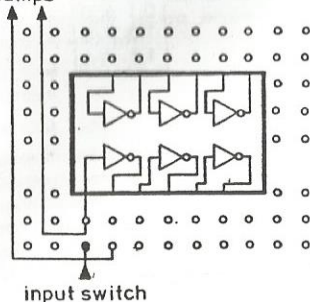


Fig. 2.1

vertically with the guide lines provided and connect it up to a switch output and a lamp input as shown in fig 2.1. Switch on the power and operate the toggle switch.

Observe the lamp indications in relation to the switch position and complete the table:-

INPUT	OUTPUT
0 (down)	
1 (up)	

The inverter is performing a NOT operation on the input and the simple table representing it is called a 'truth table'. The ideas of true and false have a particularly close association historically with the development of algebraic methods of dealing with binary variables, the earliest application being the study of formal logic. This is why we refer to computer circuits as logic elements and why we use the term 'truth table'.

The algebra that we shall use to describe the various operations we are going to study is called 'Boolean Algebra' after its originator and in it, just as in ordinary algebra, different variables are denoted by letters e.g A, B, C etc or X, Y, Z. The difference is that whereas in ordinary algebra A could be any value at all, in Boolean algebra it can only be either 0 or 1.

$$A = 0 \quad \text{or} \quad A = 1$$

The operation of NOT that we have just studied can be applied to a single variable like this:-

NOT A

This is called the 'complement' or 'negation' of A and it is usually symbolized by a bar over the variable and said— 'A bar'.

$$\text{NOT } A = \bar{A}$$

We could now write the truth table for NOT in a more general fashion thus

A	\bar{A}
0	1
1	0

● Q2.1 If \bar{A} is the same as NOT A, what do you think $\overline{\bar{A}}$ is?

Now it is time to turn attention upon operations that can be applied to more than one variable and we shall start, for simplicity, with two variables, let us call them A and B.

Practical 2.2

Switch off the power to CK353, remove the 7404 module and replace it with a 7400 module, connecting it as shown in fig 2.2.

The FEEDBACK INTIKIT

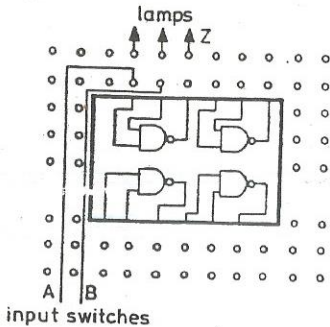


Fig. 2.2

A truth table gives the result of an operation for all possible values of the variables. We shall call the result Z. Switch on the power and successively set the input switches to the combinations shown in the table below, noting the binary value of Z (lamp on = 1, off = 0) that results.

A	B	Z
0	0	
0	1	
1	1	
1	0	

Truth table for NAND

You should have found that Z was 1 in all cases except where A and B were both 1. If we associate 1 with true and 0 with false, which is the usual convention, we can see that Z could be described as being true when it was not true that A and B were both 1.

Or, more simply

$$Z = \text{NOT } (A \text{ AND } B)$$

NOT-AND is usually abbreviated to NAND so we have

$$Z = A \text{ NAND } B$$

Even this abbreviated notation is not very convenient and so the notation $A \cdot B$ is adopted to mean A AND B.

This in turn gives us

$$Z = \text{NOT } A \cdot B = \overline{A \cdot B}$$

using the bar notation (say this 'AB bar'). When no ambiguity is possible the dot is usually dropped to give just AB.

Practical 2.3

On the INTIKIT plug in a 7404 module and connect Z to one of the inverters, taking its output to a fourth lamp. This will be \bar{Z} . Add

another column for \bar{Z} to the truth table you already have and complete it by setting A and B once again to all the possible values.

The operation you have now set up on INTIKIT is the AND operation because

$$\bar{Z} = \overline{AB} = AB = A \text{ AND } B$$

The AND operation is one of three fundamental operators in Boolean algebra, the other two being NOT, which we have studied already and OR, which we will meet soon.

You might ask why it is that AND was introduced in such a roundabout way via the NAND operation. The reason is that in integrated circuit logic NAND is the easiest and cheapest operation to achieve and it is therefore more widely used in practice as a logic element than any other.

The circuit symbols for NOT, AND and NAND are shown in fig 2.3.

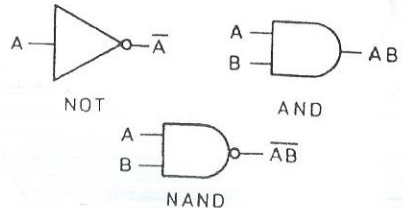


Fig. 2.3

Exercise 2.1

Draw the logic diagram for AND using a NAND followed by a NOT. Repeat for NAND using AND followed by NOT. Construct a truth table for AND.

The truth table for AND gives us some important basic identities that will later on assist in the manipulation of Boolean expressions.

0.0	=	0
0.1	=	0
1.0	=	0
1.1	=	1

Also

$$0.1 = 1.0 = 0$$

or in other words the order in which variables appear has no significance. More generally, AB and BA are the same thing. An expression like this is called a 'logical product' because of its similarity to arithmetic multiplication.

The third fundamental operation referred to earlier was that of OR. Applied to two variables A, B this says that the operation

$$Z = A \text{ OR } B$$

will have a true (1) result when either of A or B is true. The truth table is

A	B	Z
0	0	0
0	1	1
1	1	1
1	0	1

Truth table for OR

The operation is expressed symbolically as

$$Z = A + B$$

the choice of the plus sign suggesting, as is true, that it has similarity to ordinary arithmetic addition. It is called a 'logical sum' for this reason.

● Q2.2 How does the OR operation differ from ordinary addition?

To find out put another column alongside Z in the truth table for OR, head it 'A plus B' and fill in the correct arithmetic values for all cases.

The circuit symbol for OR is shown in fig 2.4.



OR

Fig 2.4

Take a look through the modules you have in the storage tray of INTIKIT. Do you see this symbol anywhere?

You probably will not since it is not a common function amongst digital integrated circuits. How then are we to perform this operation when we wish to?

To find the answer, carry out the following exercise.

Exercise 2.2

Complete the following truth table, filling in the columns from the left.

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

What do you notice about the contents of the last two columns? You should find that they are identical. Since all possible conditions are described in the table we can write

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

But the left-hand side of this equation is actually

$$\bar{A} \text{ NAND } \bar{B}$$

so that, by first negating both inputs and applying \bar{A} and \bar{B} to a NAND gate (gate is the usual term for a logic element), the OR operation is obtained. This is shown in fig 2.5

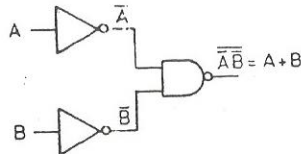
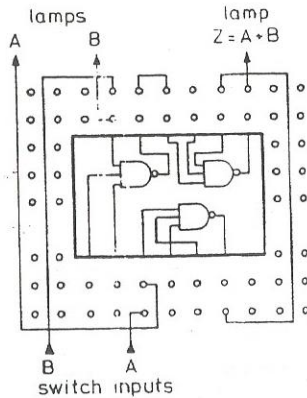


Fig 2.5

Practical 2.4

You will see that fig 2.5 shows a 7410 module and 3-input NAND gates have been used instead of inverters and a 2-input gate, as shown in the schematic. Before considering why this is acceptable, satisfy yourself that it does give the correct result by connecting up as shown and checking the output Z against the truth table for OR as you vary A and B.

It is a feature of the circuit used for the NAND gates that if an input is left unconnected it has the same effect as applying binary 1 to that input. So if a 3-input gate with inputs A, B, C has 2 inputs, B and C, unconnected

$$Z = \bar{A} \cdot B \cdot C = \bar{A} \cdot 1 \cdot 1$$

The FEEDBACK INTIKIT

But we found above that

$$0.1 = 0$$

and

$$1.1 = 1$$

Therefore $A.1 = A.1 = A$

so that $Z = \bar{A}$

Similarly if only input C is unconnected

$$Z = \overline{A.B.C} = \overline{A.B.1} = \overline{A.B}$$

Hence you can always leave unwanted inputs disconnected without affecting the results.

In the derivation of the OR operation above we saw that

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

This identity is a particular example of a very important general theorem called De Morgan's Theorem. This has a completely general form that we shall meet in the next Assignment but at present we are interested in the two forms as follows

$$\begin{aligned} A.B.C \text{ etc} &= \overline{\overline{A} + \overline{B} + \overline{C} \text{ etc.}} \\ \overline{A + B + C + \text{etc}} &= \overline{A}.\overline{B}.\overline{C}.\text{etc.} \end{aligned}$$

Examples:

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

$$\overline{A.B.C} = \overline{A} + \overline{B} + \overline{C} = \overline{A} + B + C$$

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

Exercise 2.3

Apply De Morgan's theorem to obtain alternative expressions for the following

- $\overline{A.B}$
- $\overline{A + B}$
- $A + \overline{B}$
- $\overline{A + B}$

The result of exercise (d) introduces the last operation of this Assignment. If $A + B$ means A OR B then $\overline{A + B}$ must be NOT (A OR B) or alternatively A NOR B. The circuit of fig 2.5 showed how OR was achieved and NOR is simply the complement or negation of OR.

Exercise 2.4

Draw the schematic for NOR using three inverters and one NAND gate.

Practical 2.5

Set this up on INTIKIT using the 7410 module. You already have connected plus one inverter from a 7404 module. Construct the truth table for NOR.

Exercise 2.5

From the truth table for OR write down the four basic identities in the form of equations e.g.

$$0 + 0 = 0 \text{ etc.}$$

Summary of this Assignment

You should now be familiar with negation or complementation and with the four operations AND, OR, NAND, NOR. These four are summarized in fig 2.6.

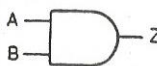



AND	OR	NAND	NOR
Z is true if A and B are true	Z is true if A or B is true	Z is false if A and B are true	Z is false if A or B is true
			
$Z = A.B$	$Z = A + B$	$Z = \overline{A.B}$	$Z = \overline{A + B}$
A B Z	A B Z	A B Z	A B Z
0 0 0	0 0 0	0 0 1	0 0 1
0 1 0	0 1 1	0 1 1	0 1 0
1 0 0	1 0 1	1 0 1	1 0 0
1 1 1	1 1 1	1 1 0	1 1 0

Fig. 2.6. The basic logical functions

You have also encountered the very important De Morgan's Theorem which relates operations to their complements.

Practical considerations & applications

As already mentioned the NAND gate is almost universally used in integrated circuit logic although all types of operation can be obtained, usually at greater cost. However, in designing logic systems, as we shall see later, the design tools and techniques are mostly better adapted to the use of the three fundamental operations of AND, OR, NOT so methods of converting the resulting expressions to the exclusive use of NAND have to be used.

Some formal methods do exist but in most practical cases these are heavy-handed and cumbersome and have little advantage over simple experience.

The fact that multi-input gates can be used for operations requiring fewer inputs than are available by leaving inputs disconnected means that when designing a large logic system it is easier to ensure that the number of modules (or packages) can be kept to a minimum. This is usually the most important economic factor nowadays. For example, suppose a circuit required three 2-input NAND gates and one NOT. This could be achieved using either

- (a) 1 7410 - triple 3-input NAND
plus 1 7404 - hex inverter
or
- (b) 1 7400 - quadruple 2-input NAND

Obviously (b) would be better.

Further Reading

Discussion and Experimental Procedure

You should now be aware of the fundamental Boolean operations of AND, OR, NOT and should understand the basic identities of logical sums and products:-

Logical Sums	Logical Products
$0 + 0 = 0$	$0.0 = 0$
$0 + 1 = 1$	$0.1 = 0$
$1 + 0 = 1$	$1.0 = 0$
$1 + 1 = 1$	$1.1 = 1$

Before we can go on to the design and construction of more elaborate logic circuits than have so far been met we have to learn some more theorems which will help in the manipulation of general expressions.

The method of establishing all Boolean theorems is to subject them to a truth table, setting all the variables in turn to both possible values so as to set up all possible conditions.

For example, what value has Z in the identity

$$Z = 0 + A?$$

Putting $A = 0$ and 1 in turn we find

$$\text{If } A = 0 \text{ } Z = 0 + 0 = 0$$

$$\text{If } A = 1 \text{ } Z = 0 + 1 = 1$$

Therefore $Z = A$ and $C + A = A$.

Exercise 3.1

Using the truth table method prove the following groups of theorems

(a) $1 + A = 1, 0.A = 0, 1.A = ?$

(b) $A + A = A, A.A = A$

(c) $A + \bar{A} = ?, A.\bar{A} = 0$

Look now at the following expression

$$Z = AB + AC$$

This differs from anything we have met so far because it contains both sum and product operations in the same expression.

The expression above is called a 'sum of products'. In ordinary algebra we know that we could take out A as a common factor to give

$$Z = A(B + C)$$

but can this be done in Boolean algebra?

To find out we will construct a truth table for the factored and un-factored forms and compare them for identity. This table will have eight rows because three variables are involved.

A	B	C	AB	AC	AB+AC	B+C	A(B+C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0
0	1	1	0	0	0	1	0
1	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1
1	1	0	1	0	1	1	1
1	1	1	1	1	1	1	1

Looking at the two columns headed AB+AC and A(B+C) we see that they are identical, which proves that factoring is allowable.

In other words

$$AB + AC = A(B+C)$$

This is all very well but how can an expression of this sort be set up using only NAND gates and inverters? Let's start by trying to realize the left hand side. We know how to form AB and AC separately because this is a simple AND operation. Fig 3.1 shows the circuits.

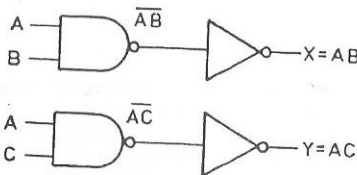


Fig. 3.1

Now to get $X + Y = AB + AC$ we need an OR operation as in fig 3.2.

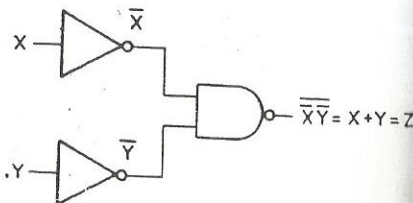


Fig. 3.2

Exercise 3.2

Put the two circuits of figs 3.1 and 3.2 together to get a logic circuit for $Z = AB + AC$. What do you notice about the inverters?

Practical 3.1

Connect up the circuit on INTIKIT and check out the truth table for all combinations of A, B, C. You should have had to use only one module 7400 to set up this function. If you seem to

The FEEDBACK INTIKIT

need more look again at the inverters. Ask your instructor if you still cannot see how to use one module only.

Leave the previous circuit connected and look now at the right hand side of the identity

$$AB + AC = A(B+C)$$

Exercise 3.3

Draw a logic circuit for $X = B+C$ and another for $Z = AX$; put the two together to get a circuit for $A(B+C)$.

Practical 3.2

Set up the result on INTIKIT, using the same input switches for A,B,C as you are already using but a different lamp to indicate the output. The two Z outputs should be identical for all settings of A,B,C.

- Q3.1 How many modules did you use in this exercise? Is this solution more or less economical than the first?

Now consider another expression

$$Z = A + BC$$

Unlike the previous case it is not evident from analogy with ordinary algebra that this can be factorized but in Boolean algebra it can, the result being

$$A + BC = (A+B)(A+C)$$

We can easily prove this by 'multiplying out' the right hand side and then applying the identities we learnt in Assignment 2

$$(A+B)(A+C) = AA + AC + BA + BC$$

But $AA = A$ so that

$$AA + AC + BA + BC = A + AC + BA + BC = A(1 + C + B)$$

Remembering that $1 + X = 1$

$$A(1 + C + B) = A.1 = A$$

Therefore $(A + B)(A + C) = A + BC$

Exercise 3.4

Draw logic circuits for both sides of this identity.

Practical 3.3

Set up your circuits for both sides as before and check that the outputs are equal for all settings of A,B,C. The left hand expression needs no more than one 7400 and one 7404. What does the right hand expression need and which is simpler?

The exercises and experiments you have just performed have demonstrated an important theorem of Boolean algebra called the 'distributive' theorem.

$$(A + B + \dots) (C + D + \dots) =$$

$$AC + AD + BC + BD + \dots$$

$$AB + CD + \dots + (A+C)(A+D)(B+C)(B+D) \dots$$

They also showed that the 'sum of products' form of an expression is generally the one most readily and cheaply realized by NAND logic. In fact you should take special note of the configuration shown in fig 3.3 which provides the general 'sum of products' function.

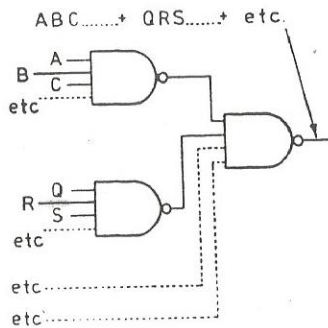
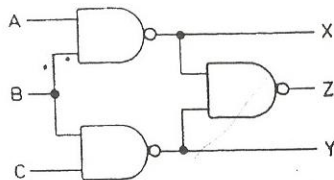


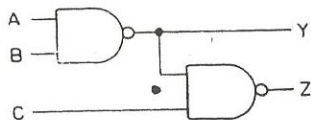
Fig. 3.3

Exercise 3.5

Write expressions for the outputs X, Y, Z in the circuits of fig 3.4 (a) and (b)



(a)



(b)

Fig 3.4

De Morgan's Theorem

In Assignment 2 we discussed De Morgan's theorem for negating a function but only as applied to simple sums like $(A+B)$ or products like $A \cdot B$. We need to know how to apply it to expressions such as

$$Z = AB + \bar{C}$$

which combine sums and products. You should recognise this expression as one of the solutions to Exercise 3.5 (b) above and you should have been able to write this down directly by application of the general sum of products configuration of fig 3.3.

Still looking at fig 3.4 (b) you should have also

$$Y = \overline{AB} = \bar{A} + \bar{B} \text{ by De Morgan}$$

$$\begin{aligned} \text{Now } Z &= Y \text{ NAND } C = \overline{AB} \text{ NAND } C \\ &= (\bar{A} + \bar{B}) C = A B + \bar{C} \text{ from above} \end{aligned}$$

● Q3.2 This identity is a solution for the complement of a combined sum and product expression. Can you see the simple rules which convert one to the other?

Practical 3.4

Set up logic circuits on INTIKIT for $(\bar{A} + \bar{B}) C$ and for $A B + \bar{C}$ and display their outputs on adjacent lamps. For all settings of A, B, C you should find the outputs to be complementary. Before drawing the circuit for $(\bar{A} + \bar{B}) C$ apply the distribution theorem to put it into sum of products form.

The simple rules for negation which you have probably recognised by now are

- (a) negate each variable individually
- (b) substitute a sum for every product
- (c) substitute a product for every sum

De Morgan's Theorem

Examples

- (a) $\overline{(\bar{A} + \bar{B})(\bar{C} + D)} = \bar{A} B + C \bar{D}$
- (b) $\overline{A + B(C + D \bar{E})} = \bar{A} (\bar{B} + \bar{C}(D + \bar{E}))$

Exercise 3.6

Write down sum of products expressions for the complements of

- (a) $(A + B)(C + D)(B + C)$
- (b) $(A + B)\bar{C} + (\bar{A} + \bar{B})\bar{C}$

simplifying the results as much as possible.

Exercise 3.7

Simplify the following

- (a) $A + AB$
- (b) $(A + B + C)(A + \bar{B})$
- (c) $(A + BC)(B + AC) + \bar{A} B \bar{C}$

Hint: The usual way to a simple result is to expand the expression using the distribution theorem and then look for terms that either cancel to 0 or have common factors.

In case you had difficulty with this exercise, here is a worked solution to (b)

$$\begin{aligned} (A + B + C)(A + \bar{B}) &= \\ A A + A\bar{B} + BA + B\bar{B} + CA + C\bar{B} &\text{— distribution} \\ &\text{theorem} \\ = A(1 + \bar{B} + B + C) + B\bar{B} + C\bar{B} \\ = A \cdot 1 + 0 + C\bar{B} \text{ (since } B\bar{B} = 0) \\ = A + C\bar{B} \end{aligned}$$

Here also is a solution to Exercise 3.6 (b)

$$\begin{aligned} (A + B)C + (\bar{A} + \bar{B})\bar{C} &= (\bar{A}\bar{B} + \bar{C}) + (AB + C) \\ &= \overline{AB + \bar{A}\bar{B}} + C + \bar{C} \\ &= \overline{A B + \bar{A} \bar{B}} + 1 \\ &\text{(since } C + \bar{C} = 1) \\ = \bar{1} = 0 \end{aligned}$$

Practical considerations & applications

The sum of products form for Boolean functions is certainly the most useful for NAND logic but if the number of variables is large or if the number of product terms is large, either could exceed the number of inputs available on a standard integrated circuit module.

Up to eight inputs are available on some types (e.g 7430) and in other inputs expanders can be connected to increase the number of available inputs.

Yet another solution if the large number of inputs is needed at the second gating level (that is, there are many individual product terms to be summed) is to use what is called a WIRED-OR connection.

The FEEDBACK INTIKIT

The WIRED-OR principle is shown in fig 3.5

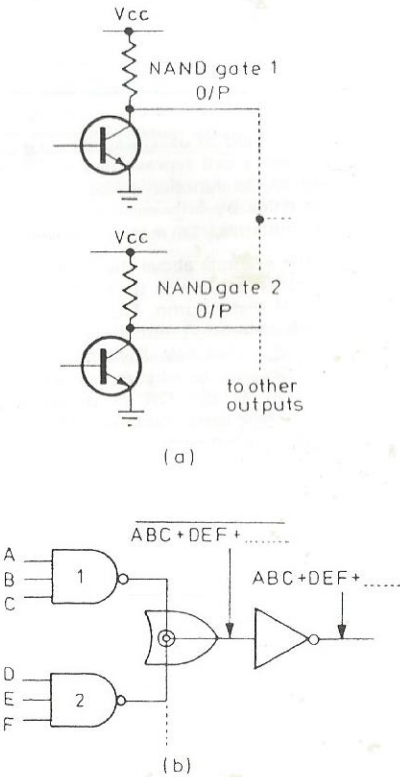


Fig. 3.5 Wired OR connection

When all inputs to a NAND gate are '1' its output is near ground since the output transistor conducts. The output is thus at '0'. If the outputs of two (or more) gates are simply connected together as in (a) the common output will be '0' when any one of the transistors conducts so the effect is to generate the function shown in (b), which also shows the symbol for a wired-OR connection. A simple negation then produces the required OR function.

A practical difficulty with the circuit of fig 3.5 (a) is that, if each gate has its own load resistor internally, all the load resistors are put in parallel by the wired-OR connection so that the current which any one transistor must conduct becomes very large. In practice such a current could not be passed and the output voltage would not go to near zero as desired. Therefore, for the wired-OR connection to be possible a

type of gate not fitted with its own load resistor must be used and this is usually called 'open-collector'. A single load resistor is then fitted externally. Usually up to about ten gates can be coupled in this way.

Apart from the various 'circuit' solutions just outlined it is of course possible to provide for any number of inputs by purely logical means. Fig 3.6 shows how eight product terms can be dealt with by two four-input NAND gates.

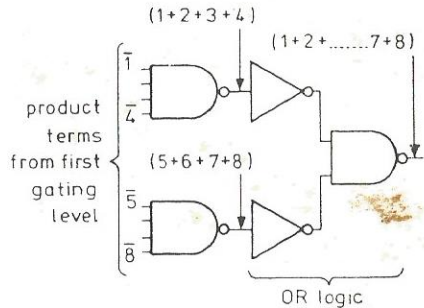


Fig. 3.6

To generate an eight-variable product term using two four-input NAND gates the logic of fig 3.7 could be used.

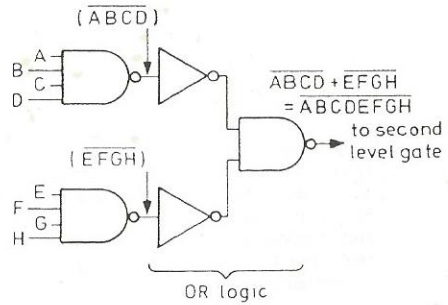


Fig. 3.7

Further Reading

Discussion and Experimental Procedure

If we look at a Boolean expression such as

$$Z = A + BC$$

one thing that might be noted about it is that neither term contains all of the three variables A, B, C. Can we rewrite it in a form such that all terms do contain all variables?

Yes, this is possible by the application of some basic theorems, but in reverse. For instance

$$\begin{aligned} BC &= BC.1 = BC(A + \bar{A}) \\ &= ABC + \bar{A}BC \end{aligned}$$

$$\begin{aligned} \text{Also } A &= A.1 = A(B + \bar{B}) \\ &= AB + A\bar{B} \\ &= AB(C + \bar{C}) + A\bar{B}(C + \bar{C}) \\ &= ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} \end{aligned}$$

Noting that the term ABC appears in both the above expressions and thus need not be repeated.

- Q4.1 Which basic theorem says this?

We can therefore write

$$Z = ABC + \bar{A}BC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C}$$

Now all terms contain all variables—in this form they are called 'minterms' and for three variables there can be up to $2^3 = 8$ of such minterms.

- Q4.2 How many minterms can there be for (a) 2, (b) 4, (c) 5 variables?

Since any Boolean expression can be expanded as was done above into a collection of minterms, a graphical representation of the individual terms should enable us to arrive at a unique diagram to represent any function we choose since any function is just the OR of a selection of minterms from amongst those available. The Karnaugh map is such a graphical representation and it contains one cell for each possible minterm. Fig 4.1 shows a map for the simplest case of two variables and a function.

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

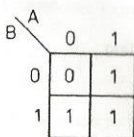


Fig. 4.1

The map has 4 cells as you would expect and the columns and rows are labelled with 0's and 1's to represent the possible states of each variable. Thus column A = 0 represents \bar{A} whilst row B = 1 represents B and the cell which is both in column A = 0 and in row B = 1 represents minterm $\bar{A}B$, and so on.

The method of using such a map is to enter a '1' into every cell representing a minterm which is part of the function to be mapped. We can do this either by fully expanding the function into its minterms, or more intuitively, as follows.

In the example above the term A will be represented by column A so 1's are entered in both cells of that column. The term $\bar{A}B$ is the cell in which column \bar{A} intersects row B so a '1' is entered in that too. The whole area of the map now covered by 1's represents the complete function as the OR of its separate product terms. Non-used minterm cells are either left vacant or filled with 0's.

Exercise 4.1

Expand $F = A + \bar{A}B$ into its minterms and verify that the same map is produced.

- Q4.3 Can you now suggest why the word 'minterm' is used?

In a Karnaugh map product terms are associated with the idea of intersections whilst the sum of product terms is associated with the idea of merging small areas into bigger ones.

So far so good but how can a Karnaugh map of a function help to simplify it?

The answer to this lies in the grouping of adjacent cells into pairs (or 4's or 8's in larger maps) so that each pair can be described by the simplest possible combination of variables (i.e. with as few variables as possible). Fig 4.2 shows the application of this idea to the map of fig 4.1.

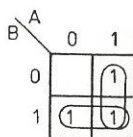


Fig 4.2

The FEEDBACK INTIKIT

The two cells linked horizontally represent B while those linked vertically represent A, the OR of these two representing the whole function.

$$\text{Thus } F = A + B$$

Of course, in this example, you could have taken this easy step from $(A + \bar{A}B)$ to $(A + B)$ without the aid of a map but in more involved examples the simplification would not always be so obvious.

Notice that the cell AB in the above example was used twice in drawing the loops but this is permissible because $AB = AB + AB$.

Exercise 4.2

In the map of fig 4.3 a function has been plotted. Write down the algebraic expression for this function. Can any simplification be possible in this case?

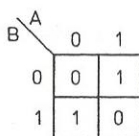


Fig. 4.3

Let us now study a map for three variables, containing eight cells. It is essential to adopt a methodical approach to the designation of rows and columns and to the mapping of functions—the three variable map of fig 4.4 illustrates both.

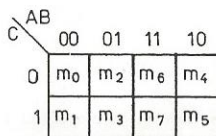


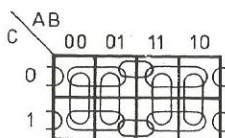
Fig. 4.4

The columns representing the various combinations of A, B are labelled so that only one variable at a time changes as we proceed from left to right. This ensures the greatest ease in the construction of loops when simplifying a function.

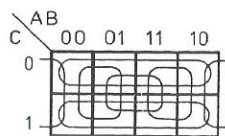
The labelling of separate cells in this map has been done in accordance with a natural binary progression as this provides an easy way of plotting a function when it is given in the form of a sum of minterms.

m_0 represents $ABC = 000$
 m_1 " " " = 001
 m_6 " " " = 110
 m_7 " " " = 111
 and so on.

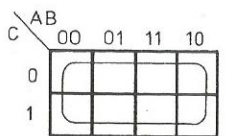
What kind of simplifying loops can be drawn on this map? Fig 4.5(a), (b) and (c) show all the possible loops of size 2, 4 and 8 which represent simple product terms.



(a)



(b)



(c)

Fig. 4.5

In the next diagram fig 4.6, some only of the above loops have been drawn and labelled with their algebraic expressions.

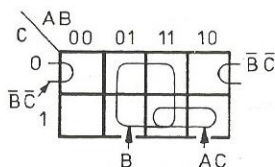


Fig. 4.6

Notice especially the loop $\bar{B}\bar{C}$, which is in two separate parts. For the purposes of simplification the map has to be regarded as being continuous from the right-hand edge to the left-hand edge.

Exercise 4.3

Redraw the maps of fig 4.5(a) and (b) on a larger scale and label all the loops in the manner of fig 4.6.

Q4.4 What is special about the loop of fig 4.5(c)?

Now let us apply the three-variable map to the simplification of some functions

(a) $F = A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C} + ABC + A\bar{B}\bar{C}$

This is a sum of minterms

$$F = m_5 + m_0 + m_7 + m_4$$

and the map, with loops added is as in fig 4.7.

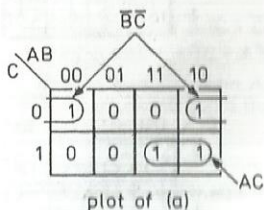


Fig. 4.7

The simplified function is $F = AC + \bar{A}\bar{C}$

(b) Take another function $F = A + BC + \bar{A}\bar{B}\bar{C}$, which can be mapped intuitively by filling 1's in all cells covered by A (last two columns on the right) then adding any extra covered by BC (just m_3) and finally adding minterm m_2 for $\bar{A}\bar{B}\bar{C}$. The result is fig 4.8.

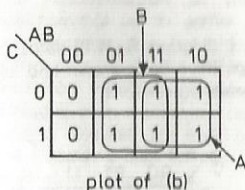


Fig. 4.8

The simplified function is $F = A + B$.

(c) $F = (A + B)(A + C)$

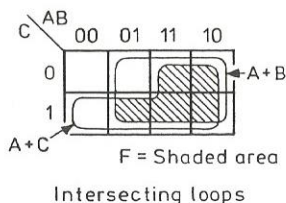
How are we to plot this since it is not in sum of products form? There are three ways—first by expanding it algebraically until it is a sum of products, not necessarily minterms.

$$F = A + AC + AB + BC$$

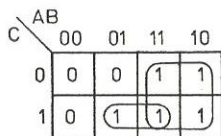
$$= A + BC$$

This method rather nullifies the point of the map as an aid to simplification because the algebra must be carried out anyway before plotting the map and in this case led directly to the simplest form. However in more complex examples it is a reasonable method.

The second way is to draw loops on the map for $(A+B)$ and $(A+C)$ separately and then fill in with 1's all cells that are included in the intersection (overlap) of the loops thus drawn. Finally the normal simplifying loops are drawn and the function written down. Fig 4.9 shows these two steps.



Intersecting loops



Simplifying loops

Fig. 4.9

The result of this is, as expected $F = A + B$.

The third method of plotting, which is sometimes useful is to apply De Morgan's Theorem to find the complement of the function and use the result, which will be in sum of products form, to plot the 0's into the map instead of the 1's.

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

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

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

Exercise 4.4

Plot the 0's from this expression and confirm that they occupy the part of the map not containing 1's in fig 4.9 above.

CD \ AB	00	01	11	10
	m ₀	m ₄	m ₁₂	m ₈
01	m ₁	m ₅	m ₁₃	m ₉
11	m ₃	m ₇	m ₁₅	m ₁₁
10	m ₂	m ₆	m ₁₄	m ₁₀

(a) 4 Variables

DE \ ABC	000	001	011	010	110	111	101	100
	m ₀	m ₄	m ₁₂	m ₈	m ₂₄	m ₂₈	m ₂₀	m ₁₆
01	m ₁	m ₅	m ₁₃	m ₉	m ₂₅	m ₂₉	m ₂₁	m ₁₇
11	m ₃	m ₇	m ₁₅	m ₁₁	m ₂₇	m ₃₁	m ₂₃	m ₁₉
10	m ₂	m ₆	m ₁₄	m ₁₀	m ₂₆	m ₃₀	m ₂₂	m ₁₈

(b) 5 Variables

Fig. 4.10

Maps for more than three variables

Fig 4.10 shows layouts for 4 and 5 variable maps with the minterm numbers inserted on the same principle explained above for the 3 variable map.

Satisfy yourself of the correctness of the minterm numbering in these two maps.

In 4 and 5 variable maps the number of possible loops of 2, 4, 8 etc minterms which can be drawn is quite large and a diagram showing them all would be confusing, but by now you have probably grasped the basic idea of adjacency between cells or groups of cells whose expressions differ in one variable only.

For example in fig 4.10(a) above the two minterms m₄ and m₁₂ are horizontally adjacent and the loop enclosing them represents the function.

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

The two terms differed only in A, which disappears from the description of the loop. Similarly minterms m₆ and m₁₄ are described by B C D. Now if the four minterms m₄, m₁₂, m₆, m₁₄ are enclosed by a loop the resulting expression for that loop is

$$\begin{aligned} &B \bar{C} \bar{D} + B C \bar{D} \\ &= B \bar{D} \end{aligned}$$

That is, C is not needed in the description. In this case the adjacency of the two loops of 2 is across the top and bottom boundaries, which must therefore be assumed adjacent.

Exercise 4.5

Trace through a similar argument in each case to show that the following groups of minterms form simplifying loops.

- (a) 4-variable (0, 2, 8, 10)
- (b) 4-variable (0, 1, 2, 3, 8, 9, 10, 11)
- (c) 5-variable (13, 15, 29, 31)
- (d) 5-variable (0, 2, 8, 10, 16, 18, 24, 26)

Redundant States

It frequently occurs in a practical application of logic design that one or both of the following conditions arise.

(a) Certain combinations of the input variables can never occur. An obvious case of this sort is in the set of binary variables representing the ten states of a decade counter, in which only those binary numbers from 0000 (zero) through 1001 (nine) are used. The other six combinations can never occur.

Such input states as cannot occur are often called 'can't happen' (or CH for short) conditions.

(b) Sometimes the output of a logic circuit will be immaterial for certain of the input combinations, or in other words it will not matter what output is produced by these combinations. Examples of this are more difficult to find but they do occur. One we shall meet with in Assignment 10 concerns the logic for control of a JK flip-flop. We often call the input states in such cases 'don't care' (DC for short) conditions.

So far as the design of combinational logic circuits is concerned CH and DC conditions can be treated as the same thing (this is not true for

sequential logic, but that need not concern us at this stage). They are both examples of redundancy and can often help towards a simpler solution than would otherwise be possible.

What we do is to enter X in the Karnaugh map for any input combination which is either DC or CH. Then, when looking for simplifying loops, we can choose to make any given X a '0' or a '1', whichever gives the simplest solution.

Here is an example:

A binary-decimal counter counts in natural binary code from zero to nine repeating. An output is required when the count is two, three, seven or eight but not otherwise.

The counter states from ten to fifteen will never occur so we have the following requirements for the various minterms

$$m_0, m_1, m_4, m_5, m_6, m_9 = 0$$

$$m_2, m_3, m_7, m_8 = 1$$

$$m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15} = X$$

These are mapped in fig 4.11.

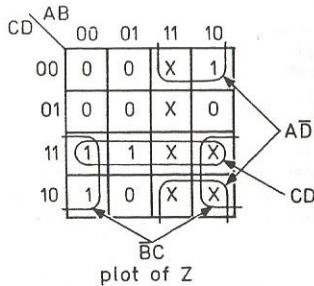


Fig. 4.11

The loops drawn in fig 4.11 make full use of the redundant (X) conditions to eliminate variables, giving

$$\text{Output } Z = A\bar{D} + \bar{B}C + CD$$

The logic diagram is in fig 4.12 and would need 1-7400 plus 1-7410 (triple 3-input NAND) using spare gates as inverters.

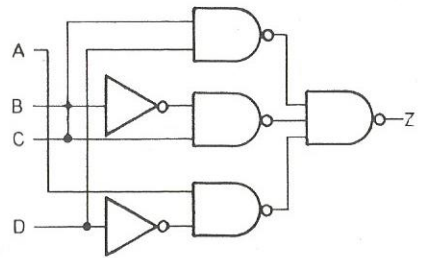


Fig. 4.12

Exercise 4.6

Write down the simplest expression for Z in the above example which could have been obtained without the use of X conditions. How many more logic modules would it have needed?

Exercise 4.7

A circuit with three inputs A, B, C must have a '1' output for minterms m_4 and m_7 . Minterms m_0 and m_6 are CH conditions. Map the function and write down the simplest expression for the output Z.

In the last exercise you should have found that there were two possible solutions, both apparently simple, namely

$$Z = \bar{A}B + \bar{B}C$$

and

$$Z = AB + A\bar{C}$$

If your solution was

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

you did not notice that you were including minterm m_4 twice unnecessarily. It frequently happens that alternative solutions exist for the same problem and in very complex cases it may even be difficult to ensure that you are not covering a minterm twice. There are formal tabular procedures for dealing with this difficulty but the problems you are likely to encounter will rarely need them. If you are especially interested your instructor will give you an idea of how to ensure that you only end up with minimal solutions.

Take another look at the two minimal solutions to exercise 4.7

$$Z = AB + \bar{B}C$$

$$Z = AB + A\bar{C}$$

Are they really equally simple?

To find out, let us draw their logic diagrams in fig 4.13.

The FEEDBACK INTIKIT

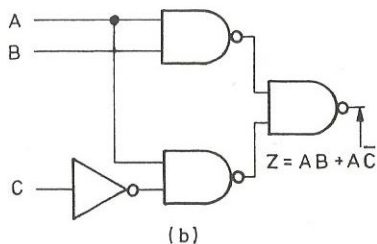
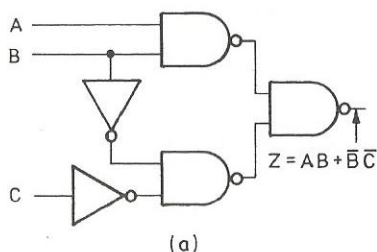


Fig. 4.13

The most important area of application you are likely to encounter is in the design of logic circuits for the excitation of inputs to flip-flops, particularly the JK type which you will meet in Assignment 6.

Further Reading

Solution (b) needs one only 7400 module but solution (a) needs one 7400 plus an extra inverter, assuming that only A, B, C are available as inputs.

This tells you that formal methods of logic simplification such as algebraic manipulation and Karnaugh Maps are only an aid to design. In the end you must use your technical judgment and commonsense to arrive at a final circuit which will cost the least.

Practical considerations & applications

Karnaugh maps for up to five or six variables offer a useful and, with practice in their application, a quick method of arriving at a first approximation to the cheapest solution to a logic design problem. After that experience and judgment must be applied to achieve the most practical configuration.

Above six variables the number of possible loops becomes very large and the advantage of a graphical representation is lost. Then tabular methods, possibly supported by computer processing, becomes necessary.

Probably the biggest single virtue of Karnaugh maps, even for relatively simple functions, is their ability to show clearly how redundant states (CH and DC conditions) can be used to best advantage.

LAB 2**The simple Latch and Clocked flip-flops and JK flip-flop.****OBJECTIVE**

To examine the basic unclocked and clocked flip-flops (bistables or memories); the Latch, the Clocked RS and the Clocked D types.

To study the Logical development of the JK flip-flop and its various forms.

QUANTITY	EQUIPMENT
1	CK853 with Leads
2	Mod. 7400
1	Mod. 7404
1	Mod. 7410
1	Mod. 7472
2	Mod. 7420

Discussion and Experimental Procedure

So far all the logic networks studied have been of a type such that their outputs are determined solely by their present inputs. These are called 'combinational logic' networks.

We now have to consider circuits whose outputs are determined partially or entirely by inputs that occurred in the past. In other words circuits having a memory. These are called 'sequential logic' networks.

Combinational networks have outputs determined by present inputs only
 Sequential networks have outputs determined to some extent by past inputs

Practical 1

Plug in a 7400 module and connect it up as shown in fig 5.1. Set input switch A initially to '0'. Switch on the power.

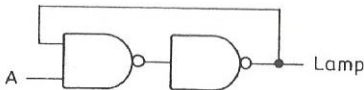


Fig. 5.1

● Q5.1 What output exists (a) when power is first switched on? (b) when input A is changed to '1'?

You should have found that the output was '0' in both cases—in other words it cannot be changed by operating the toggle. Why is this?

If you look back to Assignment 2 you will see that the truth table for NAND indicates that if either input is at zero the output will be a '1' regardless of the state of the other output. In general, for a multi-input NAND gate it needs only one of the inputs to be at '0' to give a '1' output.

In the circuit of fig 5.1, since A was at '0' when power was applied the output of the first NAND had to be '1' and that of the second NAND (an inverter) had to be '0'. But this output was 'fed back' to the second input of the first gate so that even if A now was changed to '1' this gate would still retain a '1' output.

We could say that the circuit 'remembered' that input A was initially at '0'. This 'memory' was achieved by the feedback link (in terms of the concept of feedback as applied to amplifiers etc

this is actually positive feedback since it reinforces an existing condition as opposed to negative feedback which opposes the effect of the input).

If operation of toggle A cannot alter the output to '1' how can it be done? If A were set to '1' then it would be sufficient if we could force the output to '1' even momentarily, since this '1' would feed back to the input and the first NAND gate output would become '0' since both its inputs would be at '1'. In turn this would hold the output at '1' and a stable set of conditions would again exist. Forcing the output momentarily to '1' can be achieved by temporarily applying a '0' to a second input of the second NAND gate, as in fig 5.2.

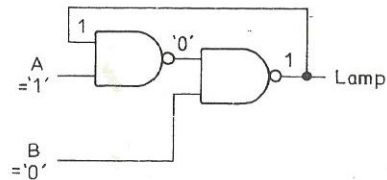


Fig. 5.2

Practical 5.2

Add input B to your circuit and observe the output when B is set to '0'.

Now set B to '1' and then alternately set A and B momentarily to 0. You will find that the output remembers the state set by the last input change, a '0' for a momentary '0' on A and a '1' for a momentary '0' on B.

The circuit of fig 5.2 is usually drawn in the symmetrical form shown in fig 5.3 and is called a simple 'latch', 'bistable', 'flip-flop' or 'one-bit memory', its outputs customarily being labelled Q as in the diagram.

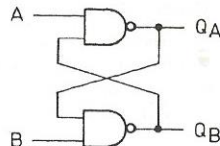


Fig. 5.3

The FEEDBACK INTIKIT

- Q5.2 What is the relationship between Q_A and Q_B in the latch circuit?

If you are not sure, connect a second lamp to Q_A (the one already connected is displaying Q_B) and observe Q_A and Q_B as A and B alternately are momentarily set to '0'.

A slightly different kind of truth table is needed to describe the behaviour of sequential circuits such as this latch because the output is not a unique function of the inputs—it depends what has gone before. What we do is to say 'If the output at the moment is such-and-such a state, what will it become when a certain input combination is applied?' For example, if Q_A is '1' when A and B are both '1', then if B goes to '0' Q_A will become '0' but if A goes to '0' Q_A will remain at '1'. In the table below Q_A is the initial state while Q'_A is the state after the inputs have been applied. There are eight rows because Q'_A depends upon three variables A, B and Q_A .

	A	B	Q_A	Q'_A
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	0	1	1	1
5	1	0	0	0
6	1	0	1	0
7	1	1	0	0
8	1	1	1	1

Truth table for fig 5.3

Let's look at this table more carefully to be sure we understand what it says.

Rows 1 and 2 say that whether Q_A starts at '0' or at '1', Q'_A will become 1 if A and B are both '0'.

Rows 3 and 4 say the same thing is true if A only is at '0'.

Rows 5 and 6 say that, whether Q_A was '0' or '1', Q'_A will be '1' if B only is at '0'.

Rows 7 and 8 say that, if A and B are both 1, Q'_A will be whatever Q_A was beforehand.

The answer to Question 5.2 above was that Q_A and Q_B are always complementary ($Q_A = 0$, $Q_B = 1$ and vice versa) under the input conditions specified. But if both A and B are set to '0' at the same time, both outputs become '1'. Try doing this on INTIKIT to verify this conclusion.

It is usual, but not essential, to ensure that A and B do not go simultaneously to '0' so that Q_A and Q_B can always be assumed complementary. Thus rows 2 and 1 in our table are not usually allowable. Apart from this you should have noticed that the behaviour of Q_A is the same for each row of the pairs 1-2, 3-4 and 5-6 whilst in rows 7 and 8 the condition that $Q'_A = Q_A$ is applicable.

This allows the truth table to be abbreviated to the following

A	B	Q'_A
0	0	1
0	1	1
1	0	0
1	1	Q_A

Abbreviated Truth Table for fig 5.3

Satisfy yourself that you fully understand the simple latch and its truth table before continuing, as it is fundamental to all forms of flip-flop that you will learn about subsequently.

The Clocked Set-Reset Flip-Flop

The simple latch circuit responded to changes in its input as soon as they occurred—a circuit like this is often called 'asynchronous' because its behaviour is not synchronised with any timing signals.

The circuit we shall study next is arranged so that changes to its inputs take effect only when a brief clock-pulse or timing signal is applied. This is achieved by the addition of two NAND gates to the circuit you have already constructed so as to control the input signals by a common clock signal.

Fig 5.4 shows the resulting circuit, in which the inputs are now called SET (S) and RESET (R) whilst the outputs are labelled Q and \bar{Q} on the assumption that they will always be complementary (that is simultaneous '0' inputs to both sides of the latch will not be allowed).

Practical 5.3

Switch the power off and reconnect the 7400 module in accordance with the layout of fig 5.4. Use the push-button at the left of INTIKIT for the clock signal input.

Set S and R initially to '0' and switch on power.

- Q5.3 Can you say with certainty what the initial state of Q will be on first switching on?

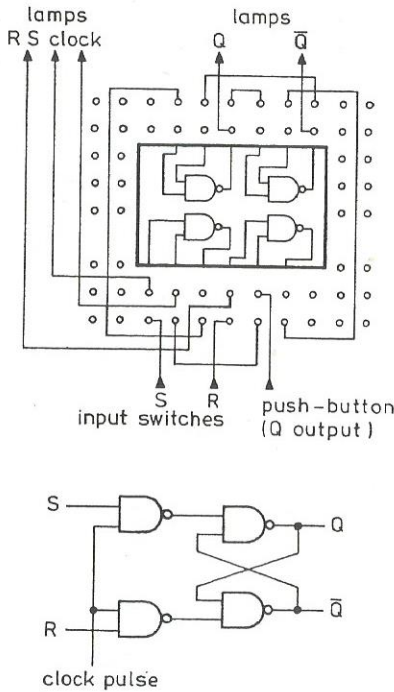


Fig. 5.4

Proceed now by setting R and S to the values shown in the following table, operating the push-button once after each change and observing the resulting value of Q before and after each step and filling in these values in the blank positions in the table. The sequence has been devised to ensure that every allowable progression is tested. Notice too that Q_n and Q_{n+1} are used to represent the values of Q before and after the clock signal; take particular note of the exact moment that any change in Q occurs.

	S	R	Q_n	Q_{n+1}
1	0	1		
2	1	0		
3	1	0		
4	0	0		
5	0	1		
6	0	1		
7	0	0		

Exercise 5.1

When you have completed this table study the results for the pairs of rows (4, 7) (5, 6) and

(2, 3) in that order and thus complete the following abbreviated table, similar to that for the simple latch given earlier.

S	R	Q_{n+1}
0	0	
0	1	
1	0	
1	1	?

We must take a closer look at the last line of this abbreviated table. In the simple latch we found that '0' could be applied to both inputs and resulted in both outputs going to 1; this was permissible in the sense that a definite output was produced but not generally, you were told, very useful. But in the clocked circuit, we want to know what state Q will be in at the end of a clock pulse. Now when the clock is applied, if R and S are both 1, the NAND gate outputs both go to '0' and both outputs go to '1', as we have seen. But what will happen as the clock signal ends (goes to '0')?

Both NAND gate outputs go to '1' but, so far as the following latch circuit is concerned, one of them must effectively reach '1' before the other, but which one is uncertain.

In other words it is a matter of chance whether the latch output Q ends up as a '0' or a '1'. That is why a query is shown in the table under Q_{n+1} . Try this on your circuit by setting R and S to '1' and operating the push-button repeatedly observing Q after each operation.

Let's think for a moment about the relative timing of the input changes and clock pulses that we used in testing the clocked SR flip-flop. You recall that you changed R and S first and then applied the clock signal, removing it again before making any further change to R and S. You should also have noticed that the latch outputs Q and \bar{Q} changed as the clock signal became '1', that is as the button was pressed. Releasing the button caused no further action.

This is represented in fig 5.5.

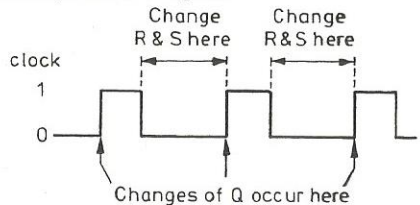


Fig. 5.5

The FEEDBACK INTIKIT

If changes to R and S were made during the '1' period of the clock then, of course the latch would respond to those changes and the output after the end of the clock would depend on what values of R and S existed just before the end (or trailing edge) of the clock pulse. Exactly how the circuit is used in a particular application depends on the circumstances but the timing shown in fig 5.5 is commonly used.

Exercise 5.2

On the circuit of fig 5.4 mark in the logic states at all inputs and outputs when the clock is at '1' and S and R = 01, 10, and 11 in turn.

The D type Flip-Flop

In the truth table for a clocked SR flip-flop, which you should have found to be thus:-

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

it can be seen that the second and third rows, taken by themselves, describe a circuit whose

output Q_{n+1} follows the input S when a clock pulse is applied. It can thus be regarded as a simple, clocked one bit store and is widely used for this purpose. In the two relevant rows S and R are complementary, so we need only insert an inverter after S to obtain R.

Fig 5.6 shows the logic and S has been replaced by D, the usual designation. In the patching diagram a NAND gate has been used instead of an inverter.

Practical 5.4

Construct the above circuit, either as shown or by adding an inverter to your existing circuit and confirm the action by altering D and operating the push-button alternately. You should obtain the following truth table.

D	Q_{n+1}
0	0
1	1

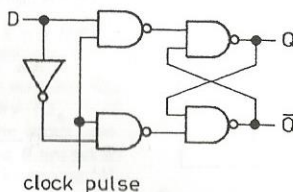
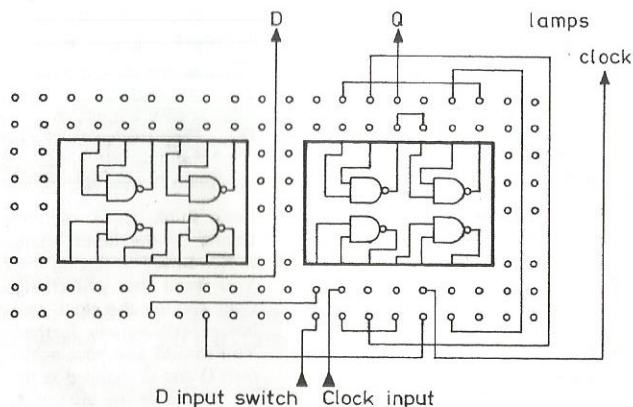


Fig. 5.6

Practical considerations & applications

Although the simple unlocked latch and the clocked SR form of it are often useful in circuits constructed in just the way they were in this Assignment, they do not usually find a place in the families of commercial integrated circuits. The reasons for this are first that the unlocked latch is so easily constructed from two NAND gates and thus not worth making in integrated form and second that the clocked SR form is restricted in its use by the uncertainty which arises when S and R are both '1'.

This uncertainty can be resolved by the addition of further logic as we shall see in Assignment 6 in which you will study the JK flip-flop, and thus once again it is not worth making the less versatile circuit in integrated form.

The D type flip-flop, however, removes the uncertainty about $S = R = 1$ by making S and R always complementary and this circuit does appear in the I.C families, although it is often referred to simply as a latch. Look up, for example the type 7475 in a maker's data book and you will find that it contains four identical latches each of which behaves exactly like the one you constructed in the Assignment. The particular point to note is that the Q output of each latch follows the D input all the time the clock is at '1' (i.e. high) but retains the last state of D prior to the clock going low again.

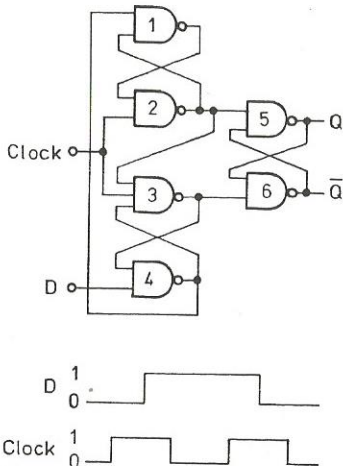


Fig. 5.7

There is another type of D flip-flop which you will see in the data books and also perhaps amongst your INTIKIT modules—this is type 7474 and is described as being 'edge-triggered' (two circuits per module).

This is something we have not previously met. The difference in behaviour is that Q takes up a state equal to input D during the rise of the clock from '0' to '1' but is independent of D whilst the clock is steady at '1' and during its fall to '0' at the end of the clock pulse. In practice this is achieved by controlling the simple latch by two further cross-connected latches in the way shown in fig 5.7, which is a slightly simplified version of a schematic taken from a maker's data book.

Exercise 5.3

As an additional exercise analyse this circuit by drawing a truth table for all gate outputs for the sequence of conditions represented by the waveform diagrams for D and CLOCK; to confirm that changes of Q occur only when the clock goes from '0' to '1'.

Practical 5.5

Construct the circuit on INTIKIT using one 7400 module for gates 1, 2, 5, 6 and one 7410 for gates 3, 4. Use the circuit to verify the truth table you have drawn.

The practical applications of D type flip-flops of both kinds are numerous but the outstanding one is as elements in multi-stage stores and shift registers which as you will see in Assignment 9 are fundamental to the majority of digital computers and digital systems generally.

A common application of a simple unlocked latch is to remove the effects of contact bounce from toggle switches and push-buttons as shown in fig 5.8.

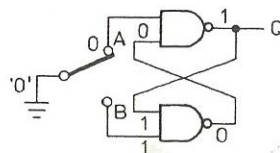


Fig. 5.8 Anti-bounce circuit

The FEEDBACK INTIKIT

When the switch is in position A the logic levels are as shown and $Q = 1$. As the switch is moved from A it becomes effectively '1' (open circuit) but this has no effect on Q , even if many bounces occur on breakaway. When the switch reaches B the first contact sets Q to 0 and again subsequent bounces can have no effect.

This anti-bounce circuit is used on the INTIKIT push-buttons to ensure clear single transitions for use in clocking counters, shift registers and other circuits, as you will see later.

Further Reading

Discussion and Experimental Procedure

In Assignment 5 we met several different kinds of clocked and unclocked flip-flops having different characteristics. However, all these types had one thing in common, which was that one of the possible combinations of the two inputs was not allowed because it led to uncertain behaviour.

In the SR types this input combination was simply barred from use and in the D types it was avoided by ensuring that the two inputs were always complementary.

It would be very convenient if it were not necessary to avoid this condition so that any input combination had a meaning, and it would also be useful if we could put this input condition to work to make the flip-flop do something that so far it has not been able to.

● Q6.1 What do you think this extra job could be?

Whether or not you can see the answer to this question let us find out by an experiment what a flip-flop of the JK type can do before going on to study its logical structure in more detail.

Practical 6.1

Plug in a 7472 module and connect it to four toggle switches and the Q output of the right-hand push-button as shown in fig 6.1.

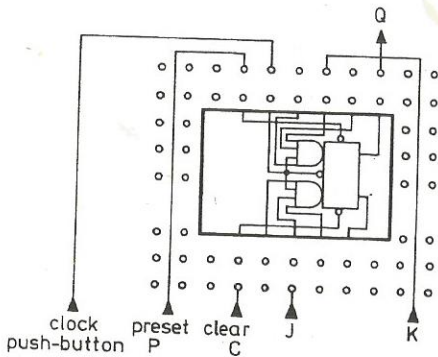


Fig. 6.1

For the moment connect both P and C to binary '1' while you concentrate on inputs J and K and the output Q. Do not worry about the details of the circuit symbol at this stage. Switch on the power and complete the truth table below by applying inputs in the sequence

shown and then operating the push-button once after each new setting. Record the state of Q before (Q_n) and after (Q_{n+1}) the clock pulse in each case.

STEP	J	K	Q_n	Q_{n+1}
1	1	0		
2	0	0		
3	1	1		
4	0	0		
5	1	1		
6	1	0		
7	0	1		
8	0	1		
9	1	0		

When you have completed this table and ignoring row 1, which is just a means of ensuring the correct starting state, collect together the results of the pairs of rows (2, 4), (7, 8), (6, 9) and (3, 5) in that order to complete the following abbreviated table in which the column for Q_n no longer appears.

	J	K	Q_{n+1}
(2,4)	0	0	
(7,8)	0	1	
(6,9)	1	0	
(3,5)	1	1	

Compare this with the table for a clocked SR flip-flop from assignment 5.

● Q6.2 What difference do you notice?

The answer, of course, is that the JK flip-flop allows inputs J and K to be '1' simultaneously and, when they are, the output always changes to the complement of its present state at every clock pulse. In other words we say that the flip-flop 'toggles'.

Practical 6.2

Now go back to the circuit and, by observing the output changes for different settings of J, K satisfy yourself that the above table really does describe the circuit behaviour. Also find the answer to the following questions.

● Q6.3 At which edge of the clock pulse does the output change occur (a) the '0' to '1' or leading edge or (b) the '1' to '0' or trailing edge?

● Q6.4 If you maintain the clock at '1' whilst altering J and K to some setting different from the one they had when the clock was changed to '1', does the output change too?

The FEEDBACK INTIKIT

● Q6.5 If you hold the clock at '1' and alter J and K as before and then restore the clock to '0' which setting of J, K does the circuit respond to:-

- (a) the setting which existed before the clock went to '1'
or (b) the setting which exists just before the clock returns to '0'?

Take care in obtaining the answers to these questions as they are quite important.

Practical 6.3

Before leaving this circuit carry out two more tests. First, set the output to '0' using J, K and the clock—then switch the P input briefly to '0' and back to '1' and then the C input briefly to '0' and back to '1'.

You should find that P sets the output to '1' and C sets it to '0' without any operation of the clock. These are asynchronous (unclocked) 'Preset' and 'Clear' inputs and are used as a simple way of setting an initial state into the flip-flop.

Finally set JK = 1, 1 and operate the clock repeatedly. Draw a waveform diagram showing the relationship between the Q output and the clock input.

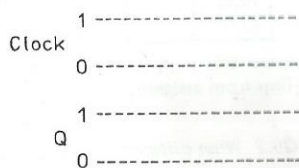


Fig. 6.2

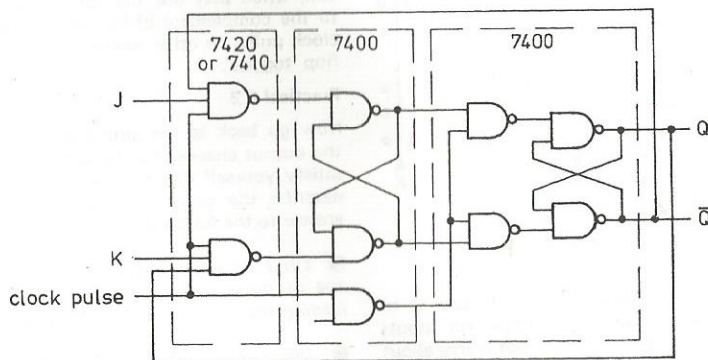


Fig. 6.3 Master-Slave JK

In this mode the flip-flop behaves as a 'divide-by-two' circuit or 'one-bit counter' since the output makes one full cycle from '0' to '1' and back for every two such cycles of the clock.

The Master-Slave JK

If you obtained the correct answers to questions 6.3, 6.4 and 6.5 you will have discovered that the output does not respond in any way to J and K until the trailing edge of the clock but then it always responds according to the last setting just prior to that edge. In this respect it is not like any of the flip-flops which came before. Just to remind you:

Clocked SR and its derived D type:- output changes at clock leading edge but if S and R then change output responds until clock trailing edge.

Edge-triggered D type:- output responds only at clock leading edge—subsequent change in D has no effect.

If you compare these descriptions carefully you will see that the JK we have just examined differs from, for example, the clocked SR in that it seems to be able to remember input changes after the clock is high without passing them to the output. It is in fact two flip-flops, referred to as the 'master' and the 'slave' and they are actually clocked SR types coupled together as shown in fig 6.3.

Notice about this circuit that

- The clock to the second SR flip-flop is inverted
- The inputs to the first SR flip-flop are effectively

$$S = J\bar{Q}$$

$$R = KQ$$

The action is as follows—when the clock is high the second or slave flip-flop is disabled by the inverted clock input but the first or master flip-flop is able to respond to its inputs as in the above expressions.

When the clock goes low again the master flip-flop is disabled but whatever contents it has are passed to the slave flip-flop and hence the output. To confirm that this circuit should behave in accordance with the truth table we obtained previously for the 7472 module let us complete the following table for Q^1 , the master flip-flop output, for various values of Q , J and K .

Q^1 of course becomes Q_{n+1} at the clock trailing edge whilst Q is the same as Q_n .

STEP	J	K	Q	S=J. \bar{Q}	R=K.Q	Q ¹
1	0	0	0	0	0	0
2	0	0	1	0	0	1
3	0	1	0	0	0	0
4	0	1	1	0	1	0
5	1	0	0	1	0	1
6	1	0	1	0	0	1
7	1	1	0	1	0	1
8	1	1	1	0	1	0

Some of the rows in this table need explanation. In rows 1 and 2, since S and R are both '0' the Master output Q^1 retains its previous state. But this must be the same state as that of the slave because the slave was made equal to the master

at the end of the previous clock pulse. Thus in both rows $Q^1 = Q$.

In rows 3 and 6 exactly the same argument applies.

If we now write Q_n instead of Q and Q_{n+1} instead of Q^1 and collect the rows in pairs (1, 2) (3, 4) (5, 6) (7, 8) as before we find exactly the same truth table as was obtained for the 7472. Check this for yourself.

Practical 6.4

Switch off the INTIKIT power, remove the 7472 module and all leads and replace with the modules shown in fig 6.3, connecting up appropriately. This circuit has no preset or clear inputs. Switch on and verify that the circuit behaviour conforms with the JK truth table and responds on the clock trailing edge. It is instructive to connect a second lamp to Q^1 , that is the upper output of the master flip-flop, so as to observe the master-slave action more clearly.

The Edge-Triggered JK

Just as the D type flip-flop studied in Assignment 5 could be designed to operate on the leading or trailing edge of its clock pulse so it is with the JK flip-flop.

The master-slave type we have just investigated responded at its output when the clock went low, having remembered the values of J, K existing when the clock went high. It is quite possible to design a circuit in which the output responds to J and K as the clock goes high and thereafter ignores any changes in J, K until the

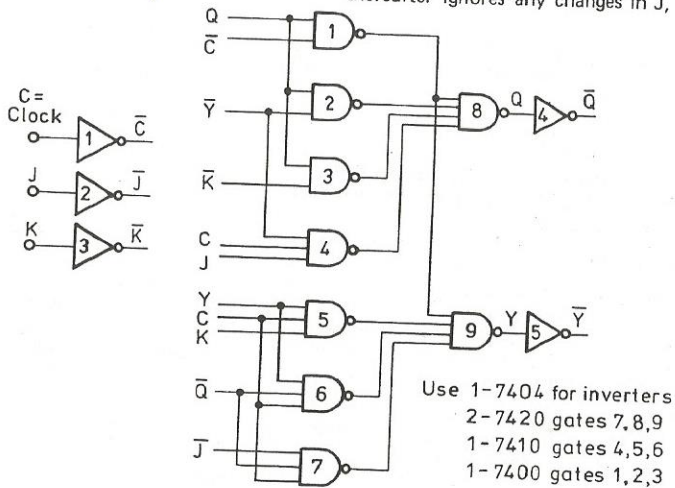


Fig. 6.4 A logical edge-triggered JK

The FEEDBACK INTIKIT

clock goes high the next time. This is usually called an 'edge-triggered JK' by the I.C manufacturers.

In a manufacturer's data book you should find for example, type 7470, although this type does not appear in a basic INTIKIT. The logic of this particular realisation of an edge-triggered JK is a little unusual and there is little to be gained by attempting to construct it on INTIKIT because it employs special circuit techniques to give a reasonably economic solution.

That such a circuit can be constructed with ordinary logic may be verified by any student interested enough to assemble the circuit of fig 6.4 and check its operation against the usual JK truth table. The particular feature to ensure is that output changes occur as the clock goes high and that whilst it remains high and after it goes low again, changes to J and K have no effect.

Practical 6.5

Construct the circuit of fig 6.4 carefully and methodically to avoid errors. Use lamps to indicate J, K and Q and use the right hand push-button for C. Do not worry about the derivation of this circuit at present. The method used in its design is beyond the scope of these assignments.

In all the circuits of JK flip-flops you have set up in this assignment you have been instructed to use a push-button for the clock, thus giving a single, bounce-free transition as described in assignment 5 (fig 5.8).

● *Q6.6 Why do you think it is so essential for the JK when it was not for the other types of flip-flop?*

The JK as a D-type

It should be obvious that by making $K = \bar{J}$ and putting $J = D$, a JK flip-flop will behave exactly like a D type.

Practical 6.6

Construct a circuit using one 7472 (master-slave JK) and connecting one inverter of a 7404 between J and K so that $K = \bar{J}$. Apply an input to J and a clock in the usual way and verify that D type behaviour results.

Although a JK is sometimes used in this way it is clearly a waste of the facilities a JK provides and normally a type 7474 (edge-triggered D type) would be used.

The JK as a binary counter

Another way of using a JK is to connect both J and K permanently to binary '1' so that it toggles at every clock pulse, thus producing a binary counter as mentioned earlier in this assignment.

Exercise 6.1

Make up a table to summarize the behaviour of the types of flip-flop listed below. This should show on which edge of the clock the circuit responds and whether input changes have any effect during clock high. Also show typical commercial I.C numbers.

Clocked SR
D types latch (e.g 7475)
Edge triggered D types
Master-slave JK
Edge-triggered JK.

Practical considerations & applications

The JK flip-flop is probably the most flexible kind available and can be put to virtually any use in control circuits, counters and shift registers.

It is, however, quite rare to find its facilities being used fully and where this is so it will often be more economic to choose a flip-flop of another kind, especially if multiple circuits are needed, when simpler circuits such as the D type latch are packaged in fours or eights instead of ones or twos as are JK flip-flops.

One area of design in which the JK facilities are fully used is in the construction of synchronous and asynchronous counters to perform non-binary counting, where the use of JK flip-flops often eliminates a considerable amount of gating which would otherwise be necessary. You will study these circuits later in Assignments 10 to 12.

You will have noticed that some of the commercial JK circuits are provided with AND gates at their J and K inputs (e.g fig 6.1) although the circuits you constructed did not use them. They are there because it often turns out, in the use of JK's as described in the last paragraph, that such gating is needed. When inputs are not used the proper procedure is to connect them to binary '1' to minimise the risk of spurious pick-up, although for experimental purposes it is usually sufficient to leave them unconnected.

Further Reading

COURSE EE333 LABORATORY

LAB 3 Equivalence, Non-equivalence and other circuits and Binary Addition

OBJECTIVE

To introduce a group of important circuits based upon two inputs to perform equivalence, non-equivalence, equality, inequality and half-adder functions.

To study circuits for the numerical addition of Binary digits.

QUANTITY	EQUIPMENT
1	CK353 with Leads
3	Mod. 7400
1	Mod. 7404
1	Mod. 7410
1	Mod. 7472
3	Mod. 7420

Discussion and Experimental Procedure

In fig 2.6 of Assignment 2 there was a table showing truth tables for AND, OR, NAND and NOR. They all showed two inputs and therefore $2^2 = 4$ rows and they all had one output.

The pattern of 1's and 0's in the output column was different in each case, thus:-

INPUTS		OUTPUT Z			
A	B	AND	OR	NAND	NOR
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Obviously, there are other patterns that could appear in the Z column, each one representing some function of A and B, just as the patterns shown above represent the functions AND, OR, NAND and NOR.

● Q7.1 How many different patterns for Z are possible?

If you are uncertain of the answer think of Z as being a four-bit number and then of how many binary numbers can be represented by four bits.

Exercise 7.1

In the table below write in all the possible patterns for Z as a series of ascending binary numbers, of which the first two have been completed for you. There will, of course, be sixteen columns, which is the answer to Q7.1.

Now write in at the head of each column the function of A and B that each pattern represents. You could use Karnaugh maps to help you but the functions are all simple and can be written down by inspection.

		Z =									
A	B	0	AB								
0	0	0	0								
0	1	0	0								
1	0	0	0								
1	1	0	1								
		0	1	2	3	4	5	6	7		

		Z =									
A	B										
0	0										
0	1										
1	0										
1	1										
		8	9	10	11	12	13	14	15		

If you completed the table correctly you should have found that

Columns 0, 3, 5, 10, 12 and 15 are functions of none or only one of the variables e.g column 5 is B.

Columns 1, 7, 8, 14 are respectively AND, OR, NOR and NAND.

Column 2 is $A\bar{B}$ and 13 is its complement $\bar{A} + B$

Column 4 is $\bar{A}B$ and 11 is its complement $A + \bar{B}$

Column 6 is $\bar{A}B + A\bar{B}$ and 9 is its complement $\bar{A}\bar{B} + AB$.

All of these last six functions are of importance in practice and have particular names as follows:-

$Z_2 = A\bar{B}$ is called 'A greater than B' or $A > B$ since the output is '1' only when A is greater than B.

$Z_{1,3} = \bar{A} + B$, by a similar argument, is 'A less than or equal to B' or $A \leq B$.

$Z_4 = \bar{A}B$ is 'A less than B' or $A < B$.

$Z_{1,1} = A + \bar{B}$ is 'A greater than or equal to B' or $A \geq B$.

$Z_6 = \bar{A}B + A\bar{B}$ is 'A not equivalent to B' or $A \neq B$ because it is '1' only when A and B have opposite values.

$Z_9 = \bar{A}\bar{B} + AB$ is 'A equivalent to B' or $A \equiv B$ since it is '1' when A and B have the same values.

Examine these functions carefully to make sure you understand them fully.

Exercise 7.2

Show by the application of De Morgan's theorem and algebraic manipulation that Z_6 and Z_9 are complementary.

We shall now study the NAND logic circuits for each of these functions and check them on INTIKIT.

$A > B, A \leq B, A < B, \text{ and } A \geq B$

These are all very simple functions, none of which needs more than three NAND gates, including inverters.

The FEEDBACK INTIKIT

Practical 7.1

Design, construct and verify the truth tables of logic circuits for each of the above functions.

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

This function, properly known as non-equivalence, is also commonly referred to as the EXCLUSIVE-OR because it is '1' when A or B but not both are at '1'.

A symbol frequently used to express the non-equivalence function is \oplus and it means the same as \neq , e.g. $A \oplus B$ is the same as $A \neq B$.

A straightforward circuit for this function is shown in fig 7.1 which also shows the Karnaugh map.

	A	0	1
B	0	0	1
1	1	0	

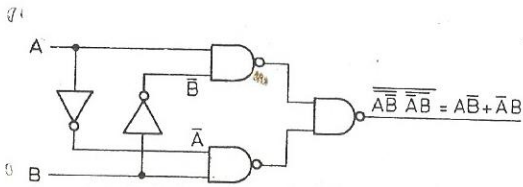


Fig. 7.1

Practical 7.2

Construct the circuit of fig 7.1 and verify its truth table.

It is possible to simplify the above circuit slightly by a method which is not apparent from a Karnaugh map but shows up after algebraic manipulation.

$$\begin{aligned} Z_6 &= \overline{A}B + A\overline{B} \\ &= (\overline{A} + A)(\overline{A} + \overline{B})(B + A)(B + \overline{B}) \\ &\quad \text{distribution} \\ &= (A + B)(\overline{A} + \overline{B}) \text{ - rearrangement} \\ &= (A + B) \overline{A\overline{B}} \text{ i.e. (A OR B) AND NOT} \\ &\quad \text{(A AND B) *} \\ &= A(\overline{A\overline{B}}) + B(\overline{A\overline{B}}) \text{ - distribution} \end{aligned}$$

Figure 7.2 shows a circuit to realize this form of the function and uses only four NAND gates.

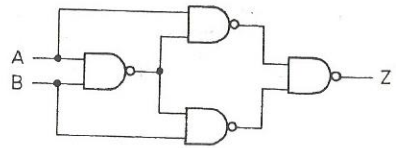


Fig. 7.2

Practical 7.3

Construct and verify the truth table of fig 7.2.

Exercise 7.3

Devise a NAND logic circuit to realize the expression for exclusive-OR marked by * in the above algebraic derivation. Your solution should use 3 NAND and 3 INVERTERS.

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

Obviously this equivalence function can be generated simply by complementing the non-equivalence function and, using the circuit of fig 7.2, would use five NAND gates. Alternatively the direct approach illustrated by fig 7.3 can be used although this also uses five gates.

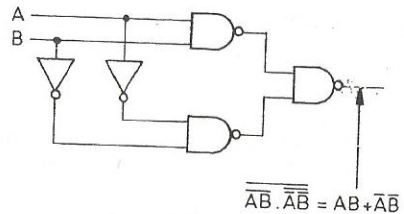


Fig. 7.3

Practical 7.4

Construct and verify the truth table of fig 7.3.

Exercise 7.4

Try working out for the equivalence function a circuit similar to that of fig 7.2 by following a similar algebraic manipulation. Is the resulting circuit any simpler than fig 7.3?

Binary Number Relations

All the functions we have studied in this assignment are concerned with the relationships between two binary variables A and B. Since binary numbers as used in digital computers are simply collections of binary variables of different numerical significance these same functions, if applied separately to each pair of digits from two binary numbers, will be able to indicate the relationships between the two numbers.

A most important relation in computing is the arithmetic sum of two numbers and, being so important, the subject of binary addition has the the next assignment, No. 8, to itself.

Other important relations between two positive binary numbers A and B are

- X numerically equal to Y $X = Y$
- X numerically greater than Y $X > Y$
- X numerically less than Y $X < Y$

We shall now take a look at the logic necessary to indicate these functions and for the sake of simplicity we shall assume that X and Y are three-bit numbers only.

X = Y—equality detector

Two numbers X and Y are numerically equal only when each pair of digits of equal significance are themselves equal.

Thus if $X = A_1, B_1, C_1$
 (A_1 the most significant)
 and $Y = A_2, B_2, C_2$
 (A_2 the most significant)

Then $X = Y$ if

$$(A_1 \equiv A_2) \text{ AND } (B_1 \equiv B_2) \text{ AND } (C_1 \equiv C_2)$$

In fig 7.3 we had a logic circuit for equivalence and now we need three such circuits plus an AND to produce a signal which will be '1' only when $X = Y$. The result is shown in fig 7.4.

Practical 7.5

Construct the logic of fig 7.4 using two sets of three toggle switches for the inputs, confirming that for all eight values of the binary

numbers X and Y, Z is at '1' only when $X = Y$ (use one 7400 for each stage and one 7410 for the final gate and its inverter).

$X > Y$ and $X < Y$ inequality detector

Looking back to the beginning of this Assignment you will find that the simple expressions for inequality of two binary variables A and B were

$$A > B \quad \text{function } A\bar{B}$$

$$A < B \quad \text{function } \bar{A}B$$

Practical 7.6

Using three identical circuits for $(A_1 > A_2)$, $(B_1 > B_2)$ and $(C_1 > C_2)$ feeding into an AND gate (similar to fig 7.4) construct a detector for $X > Y$ and check its operation for various values of X and Y as shown in the following table, filling in the output that is indicated by the output lamp.

X	Y	OUTPUT
0 0 0	0 0 1	
0 1 0	0 1 0	
1 1 1	0 0 0	
1 0 0	1 0 1	
1 0 1	1 1 0	
1 0 0	0 1 1	
1 1 0	0 0 0	

● Q7.2 Do you get the output that you expect in every case? If not, why do you think this is so?

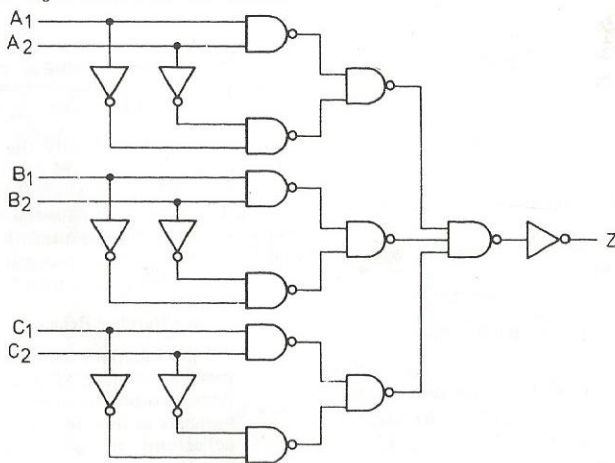


Fig. 7.4 Equality detector

The FEEDBACK INTIKIT

You should have found that the result was correct whenever X was less than or equal to Y but when X was greater than Y the result was correct only for $X = 111, Y = 000$. This is because the circuit was set up to detect only when every digit of X was greater than the corresponding digit of Y.

How can we arrive at the correct circuit? Supposing we used an OR gate instead of an AND gate to combine the results of each digit comparison—then we should have an output when any one digit of A exceeded the corresponding digit of B.

Exercise 7.5

Check each of the above examples in the table to see whether the use of an OR gate would give correct solutions in every case.

You should find that the result is correct for every case except when $X = 101$ and $Y = 110$ because although $C_1 > C_2$, X is actually less than Y.

What we have omitted to take account of is the fact that the different digits have different numerical significance; this did not matter when looking for equality since there is only one condition of equality, but it does matter when looking for inequality.

The rule we have to follow in determining if $X > Y$ is to look first at the most significant digits for inequality. If they are unequal the lower order digits do not matter but if they are equal we look at the next digits down for inequality and so on.

In symbols

$X > Y$ if $(A_1 > A_2)$

OR if $(B_1 > B_2)$ AND $(A_1 = A_2)$

OR if $(C_1 > C_2)$ AND $(A_1 = A_2)$ AND $(B_1 = B_2)$

In practice what we must do is to detect not only when $A_1 > A_2$ but also when $A_1 = A_2$ and use the latter signal to enable the next comparison between B_1 and B_2 to be carried out. A little earlier in this Assignment it was pointed out

that equality could be detected simply by following an inequality detector by an inverter and the inequality circuit of fig 7.2 is ideal for this purpose. Fig 7.5 shows the result.

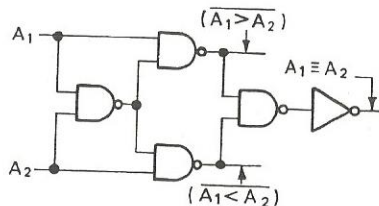


Fig. 7.5

This circuit has the advantage that it gives simultaneous outputs for $(A_1 > A_2)$, $(A_1 < A_2)$ and $(A_1 = A_2)$. It will serve exactly as it stands to compare the most significant digits but for lower digits it must be arranged to produce no outputs unless the next higher digits are equal.

To do this it is modified to the circuit of fig 7.6.

Exercise 7.6

Substitute P for $(A_1 = A_2)$ in fig 7.6 and hence show that the outputs given in the diagram are correct.

The third and any subsequent stages are identical with fig 7.6 and it should by now be obvious that

(a) The centre output of the last circuit will be

$(A_1 = A_2)(B_1 = B_2)(C_1 = C_2)$ etc or in other words $X = Y$.

(b) The upper and lower outputs of the various stages, if suitably inverted and combined in two separate OR gates, will give signals corresponding to $X > Y$ and $X < Y$.

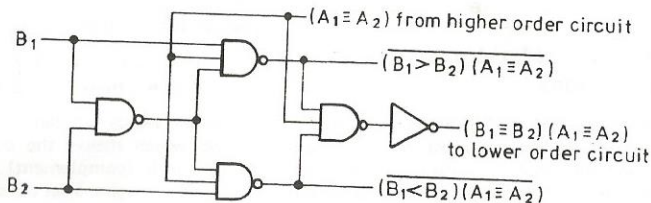


Fig. 7.6

The full circuit is shown below in fig 7.7.

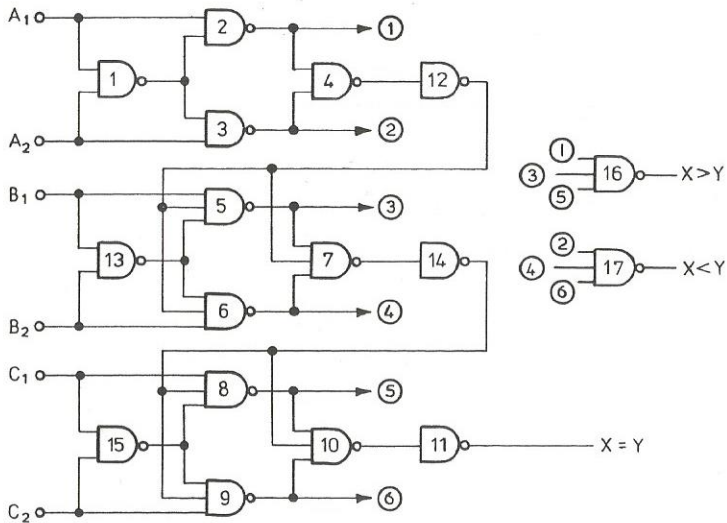


Fig. 7.7 3-bit numerical comparator

Practical 7.6

Construct the logic of fig 7.7 on INTIKIT using modules as follows:-

Gates	Module
1, 2, 3, 4	7400
5, 6, 7	7410
8, 9	7420
10, 11	7420
12, 13, 14, 15	7400
16, 17	7420

All these modules are available in the basic INTIKIT set. For all the 64 combinations of the 8 values each of X and Y confirm that the outputs give correct indications of the relative values of X and Y.

● Q7.3 What is (a) the maximum and (b) the minimum number of outputs of fig 7.7 which can simultaneously be at '1'?

Practical considerations & applications

The circuits studied in this Assignment and in the next are among the most important and widely used ones. For this reason they have been manufactured in integrated form by many suppliers.

For example, discounting binary adders (see Assignment 8) reference to a Series 74 catalogue will show the following

- 7486 Quadruple exclusive-OR (non-equivalence)
- 7485 4-bit numerical comparator giving 'greater than', 'less than' and 'equal' outputs with provision for cascading as many modules as necessary to compare numbers of any length (identical in principle with the logic of fig 7.7).
- 7487 4-bit true/complement circuit.

The meaning of this last circuit requires explaining. Suppose we have a non-equivalence circuit for two inputs A and B, then as we have just seen, the output is

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

If we now put one of these inputs, say A, successively equal to '0' and '1' we obtain

$$\begin{aligned} \text{if } A = 0 \text{ then } Z &= 0 \cdot \bar{B} + 1 \cdot B = B \\ \text{if } A = 1 \text{ then } Z &= 1 \cdot \bar{B} + 0 \cdot B = \bar{B} \end{aligned}$$

In other words we can consider A as a control signal which makes the output equal either B (true) or \bar{B} (complement) when the input is B. If we now imagine a set of, say four such circuits, all having the same control signal A but different

The FEEDBACK INTIKIT

input signals B_1, B_2, B_3, B_4 (representing a binary number) then the value of A decides whether the output number is equal to the input or equal to the complement of the input.

Applications of numerical comparator circuits occur in digital computers to determine the relative magnitude of numbers and in numerical control systems such as are used in the control of machine tools, for instance, to see whether the machine position, represented by a binary number is greater than or less than the required position, also set by a binary number.

Another application of the non-equivalence circuit is to the conversion of pure binary code to Gray code and vice-versa. (Gray code, often called cyclic binary code, is used in some devices for the digital measurement of shaft position). See Assignment 13 for details.

Further Reading

Discussion and Experimental Procedure

In this Assignment we are going to study circuits for adding together two binary numbers, but before we can decide what logic is required we must first decide what binary addition means.

In the decimal number system, when we add together two digits we have to consider two distinct possibilities:-

- (a) The sum is less than or equal to 9 (radix 10 minus 1)
- (b) The sum is greater than 9.

If (a) is true we simply give the sum as the appropriate decimal digit: $4 + 3 = 7$ or $2 + 7 = 9$, to give two examples.

If (b) is true, however, we cannot express the sum just as one digit but we have to carry over into the next higher column. Thus $7 + 8 = 5$ carry 1 and $9 + 9 = 8$ carry 1, for example.

Translating this into the binary system we have two similar possibilities

- (a) The sum is less than or equal to 1 (radix 2 minus 1)
- (b) The sum is greater than 1.

Exercise 8.1

Following the same argument as used in decimal write down the sum and carry obtained when the two digits added are (a) 0 and 0 (b) 0 and 1 (c) 1 and 0 (d) 1 and 1.

Now consider the two digits to be added as binary variables A and B. We want to construct logic which will produce the correct sum and carry terms for each of the four cases of Exercise 8.1 so we must first construct the two truth tables and then see if we can select suitable logic to realize them.

A	B	SUM	CARRY
0	0		
0	1		
1	0		
1	1		

Exercise 8.2

Complete the above table from the results of Exercise 8.1

You should find that the sum function is identical with a function we met in Assignment 7, namely the non-equivalence (exclusive-OR) whilst the carry function is just AND. If you did not get this result ask your instructor to explain.

Thus we have

$$\begin{aligned} \text{SUM} &= \Sigma \text{ (sigma)} = \overline{A}B + A\overline{B} \\ \text{CARRY} &= C_0 = AB \end{aligned}$$

The straightforward NAND logic for these functions is shown in fig 8.1 which is just a combination of fig 7.1 with an AND gate.

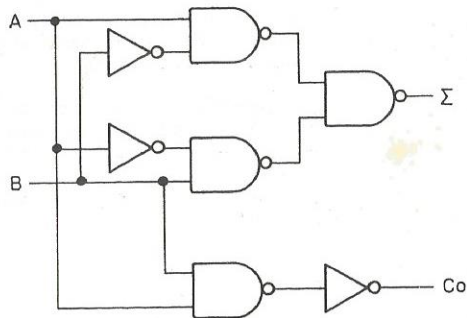


Fig. 8.1 Simple adder

Practical 8.1

Construct the logic of fig 8.1 using one 7400 and one 7404. Connect lamps to the Σ and C_0 outputs and confirm the operation in accordance with the truth table.

In Exercise 7.3 of the previous Assignment you were asked to devise a logic circuit for the Exclusive-OR function as expressed by

$$A \oplus B = (A + B) \overline{AB}$$

In this form we can see that the expression for $C_0 = AB$ appears in complement form and this leads to a slightly different form of adder as shown in fig 8.2. shown in fig 8.2 overleaf.

Practical 8.2

Construct and confirm the truth table for the logic of fig 8.2.

In the circuits just studied C_0 was the carry out to the next higher digit position. In its next position the logic circuit will have to accept, not only the digits of the two numbers, but also the carry from the next lower stage. In other words it must add together three inputs instead of two.

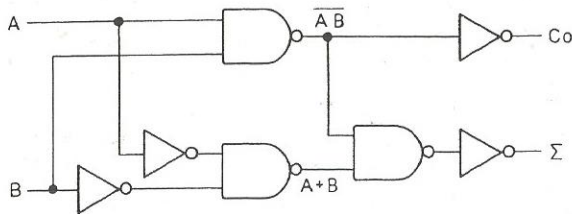


Fig. 8.2 Alternative adder

One obvious way of doing this is to first add together two of the three, using one of the circuits above and producing a partial sum and a partial carry, and then use another similar circuit to add the third input to the results of the first addition.

Thus if the three digits to be added are A, B and C (C = carry in)

$$(A + B) = \Sigma_1 \text{ with a possible carry out } C_0$$

$$\text{and } (\Sigma_1 + C) = \Sigma_2 \text{ with a possible carry out } C_0$$

Notice that a carry out can only occur in one of the stages because if A and B are both '1', $\Sigma_1 = 0$ so no carry can occur in the second stage; similarly if Σ_1 and C are both '1' no carry out can have occurred in the first stage.

Because each simple adder is performing only half of the complete process it is usually referred to as a HALF-ADDER. The complete circuit made up of two such half-adders is called a FULL ADDER.

A. Full Adder

Fig 8.3 is a block diagram of a full adder composed of two half-adders. You should notice two things about the diagram

- The Σ output of the first half is used as an input to the second half.
- The two carry outputs C' are combined in an OR gate to produce C_0 .

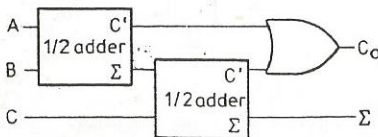


Fig. 8.3 A full adder

Exercise 8.3

Draw the complete logic circuit for the full adder of fig 8.3, using the half-adder circuit of

fig 8.2. You will find that the inverter for C_0 in fig 8.2 can be dispensed with, as also can one of the two input inverters of the second half-adder. Your final circuit should use 7 NAND gates and 5 inverters and can be built with two 7400 modules and one 7404 module.

Practical 8.3

Construct the full adder you have just designed and use it to provide the data to complete the truth table below for Σ and C_0 for all eight possible combinations of A, B and C.

A	B	C	Σ	C_0
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

Exercise 8.4

From your truth table for the full adder fill in the missing numbers in the following statements

- The sum output is '1' when any — or all — of the inputs are at '1'
- The carry output is '1' when any — or all — of the inputs are at '1'.

There are many different ways of realizing the truth table for a full adder apart from the two half-adder method. For example if you draw Karnaugh maps of the two functions for Σ and C_0 from your truth table they should lead you to the following sum of products expressions.

$$\Sigma = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$$

$$C_0 = AB + AC + BC$$

Exercise 8.5

Confirm these expressions by drawing the Karnaugh maps and note how they tally with the statements in Exercise 8.4

These expressions would need 9 NAND gates and 3 inverters and are thus not quite so economical as the two half-adder method. In fact they require 5 modules because several gates must have 3 or 4 inputs, whereas the two half-adder method needed only 3 modules. Of all the variations the two half-adder circuit turns out to be the simplest.

Addition of multi-digit numbers

When two multi-digit numbers are to be added there are basically two ways of performing the addition.

First we can construct a separate adding circuit for each digit as shown in fig 8.4.

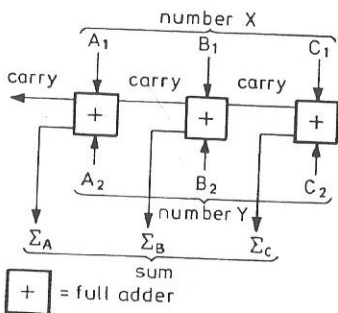


Fig. 8.4

This method is called PARALLEL ADDITION because all digits are added simultaneously.

● Q8.1 Does the first (least significant) adder need to be a full adder?

A point of great importance concerning parallel adders for digital computers is that they are fast in operation because all digits of the sum are generated nearly simultaneously. But not quite simultaneously because each adder in the chain cannot produce its outputs correctly until it knows whether or not the previous adder has produced a carry. In other words the full output is not obtained until a carry has 'propagated' fully along the chain.

How fast a carry propagates depends upon how quickly logic gates respond to changes and how many gates are included in the carry path. Another factor is that a carry only has to propagate along the full length of the adder when all the digits of both X and Y are at '1'.

A two-bit parallel adder

Practical 8.4

On the INTIKIT deck you should have a full adder constructed. Add to this a half-adder for the least significant digit using the circuit of fig 8.2 constructed with one 7400 for the inverters and one 7410 for the gates. Arrange the inputs in two separate groups of two switches each for A_1 B_1 and A_2 B_2 and use three lamps on the carry out, Σ_A and Σ_B . Of course the carry out is actually the same as the third digit sum in the absence of any higher input digits. Thus the addition of two 2-digit numbers can give rise to at most a 3-digit sum. When your circuit is complete check that every combination of the four values of each input X and Y gives the correct sum output.

● Q8.2 What is the highest sum you can obtain with this arrangement?

We said earlier that there were two basic methods of addition. The second method is to use only one adder circuit and to present the digits of the numbers to be added one pair after another, starting with the least significant. If a carry is generated at any step it is remembered by a binary store and presented to the adder at the same time as the next higher digits.

This method of addition is called SERIAL and it is much more economical in logic but slower because of the time taken to present all the digits sequentially.

The block diagram of a serial adder is shown in fig 8.5 using a D type flip-flop (see Assignment 5) as the carry store. X_n and Y_n are the nth bits of X and Y (counting from the least significant end).

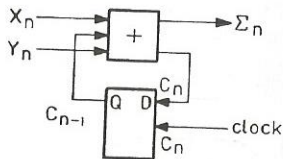


Fig. 8.5 A serial adder

The diagram shows the conditions just before a clock pulse occurs. When the clock pulse arrives it will shift C_n to the Q output of the flip-flop and hence to the adder input.

At the same instant arrangements will be made to present digits X_{n+1} and Y_{n+1} whilst the existing output Σ_n is stored elsewhere to make way for the new sum Σ_{n+1} .

The FEEDBACK INTIKIT

Exercise 8.6

In the table below are shown two numbers X and Y, least significant bit at the right. Fill in the rows marked Σ and C for each column, starting from the right.

n	9	8	7	6	5	4	3	2	1	0
X	0	1	0	0	1	0	1	1	1	0
Y	0	0	1	1	1	0	0	1	1	1
Σ										
C										

A Serial Adder

Practical 8.5

On INTIKIT remove the half-adder logic you constructed for Practical 8.4 and add instead a D type flip-flop constructed from a JK module 7472 plus an inverter as described in Practical 6.6 Assignment 6.

Connect the carry output of your full adder to the \bar{D} input and the Q output to the carry input of the adder as in fig 8.5. Use the left-hand push-button as a clock input to the flip-flop and connect a lamp to the sum output of the adder.

Initially set the X and Y input switches to '0' and initiate one clock pulse to clear the carry store.

Now set X and Y sequentially from the right as in the table above, operating the clock button between settings and noting the sum output. The exact sequence you should follow is:-

- Set X_n and Y_n digits
 - Note sum output Σ_n
 - Operate clock button
 - Set X_{n+1} and Y_{n+1}
 - Note sum output Σ_{n+1}
- etc.

You should obtain the same sequence of digits that you have already entered in the table above. Try different values of X and Y, ensuring that you clear the carry store before each test as described above.

Practical considerations & applications

The applications of adder circuits are fairly obvious because addition (and subtraction, which is only a minor modification of addition) is the basic operation of all digital computation.

The practical implementation of adding is a matter for considerable ingenuity on the part of logic designers, however, in their attempts to achieve higher speed and lower cost. Very many

schemes have been devised, for example, to reduce the carry propagation time of a parallel adder, some relying on ultra-fast circuits design, some on rearrangement of the logic so that the carry path takes it through fewer stages of gating and some on 'looking ahead' to the higher order stages to see how far a carry will need to be propagated.

It is not possible in this manual to describe any of these methods in detail but the suggestions for further reading mention many of them.

The serial adder you tested in Practical 8.5 is not a fully representative one because it contained no circuits for presenting the two numbers sequentially nor for storing the result. In practice this would usually be done by shift registers, which will be studied in the next assignment.

Because addition is so important you will find a number of integrated circuits providing 1, 2 and 4 bit full addition. Some of these are as follows and are worth looking up in a maker's data book.

7482 - 2-bit full adder—14 pin

7483 - 4-bit full adder—16 pin

Both of the above can be cascaded to give any length parallel adder.

7480 - Gated full adder—1-bit adder with gated inputs and complementary outputs (Σ , $\bar{\Sigma}$, C_0 , \bar{C}_0).

Further Reading

OBJECTIVE

To study the various kinds of registers in common use, including simple stores, unidirectional and bidirectional shift registers.

To define the terms Asynchronous and Synchronous in relation to counting circuits and to study the design and construction of Synchronous Counters.

To study design methods of Asynchronous Counters and to construct various examples.

QUANTITY	EQUIPMENT
1	CK353 with Leads
3	Mod. 7400
1	Mod. 7404
1	Mod. 7410
4	Mod. 7472
3	Mod. 7420

Discussion and Experimental Procedure

In digital computing a very common requirement is to be able to store temporarily a binary number so that later on it may be used again for further calculation. More permanent storage is usually done in a large scale memory such as a ferrite core store, disc file or magnetic tape but for shorter term storage it is usual to employ a 'register'.

This is no more than a set of one-bit memories such as the flip-flops we met in Assignment 5, one for each bit in the number to be stored. Fig 9.1 illustrates this idea for a typical four-bit number, the least significant bit being at the right.

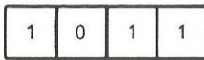


Fig. 9.1 A register

A register like that of fig 9.1 could be made from, for example, D type or JK flip-flops, but it is not enough simply to have the register—we must also be able to put numbers into it and get them out again.

There are two ways of putting data into a register and two ways of reading it out again. The two methods are serial and parallel, defined as follows.

In PARALLEL operation, all stages of the register are loaded (set) simultaneously and are read out simultaneously.

In SERIAL operation the number to be loaded is applied sequentially bit by bit to one end of the register and shifted or 'slid' along it until all stages are loaded. Conversely to read a number out it is shifted to one end again and the output of the end stage observed bit by bit.

In a moment we shall take a closer look at the meanings of these operations and the methods of achieving them in practice but you should note first that the two options for input and two for output give a total of four possible types of register:-

Parallel in — parallel out
Serial in — serial out
Parallel in — serial out
Serial in — parallel out

Actually there are even more than four because the shifting operation in the serial mode may

be in either of two directions, right or left. Indeed some registers are capable of being shifted in either direction (called 'reversible shift').

However, before going on to study the details of shifting registers we must examine the somewhat simpler problems of parallel input and output.

Parallel Input

The very simplest type of register is composed of simple binary latches. To control the inputs to several stages it is necessary to use a common control line to be energized when data is to be set into the register. A little thought will show that this purpose is achieved by the D type flip-flop (see fig 5.6 of assignment 5).

Practical 9.1

Construct a three stage register from three D type flip-flops as shown in fig 9.2, connecting input toggles to each of the D (data) input lines and lamps to each of the Q outputs. Use the left-hand push-button Q output for the clock. Use three 7400 and one 7404 for this circuit.

Set various numbers on the toggle switches and then apply a clock pulse. At the leading edge of the clock the register should take up the applied pattern and retain it.

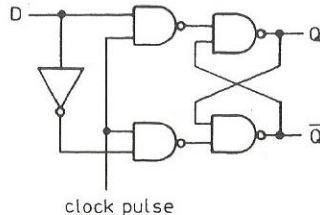


Fig. 9.2 Simple register stage

Parallel Output

In Practical 9.1 the output pattern appeared on the lamps as soon as the register was loaded but in some applications it must be possible to gate the output so that it appears only when some other clock pulse occurs. This is easily achieved by adding an AND gate to each Q output as shown in fig 9.3.

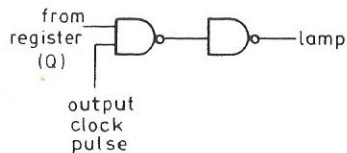


Fig. 9.3 Output gating

The FEEDBACK INTIKIT

Practical 9.2

Add output gates to all three register stages, commoning the output clock pulse lines and supplying them from a fourth toggle switch. Use one 7420 for each stage.

Enter various numbers into the register as before and then display them on the lamps by applying an output clock pulse.

● Q9.1 Without alteration to the register logic how would you go about 'clearing' the register, that is setting all stages to binary '0'?

Exercise 9.1

Assume that the application demanded the ability to clear the register independently of the input clock pulse and work out how this could be done. Check your solution on the register already built (Hint: the simplest solution requires the addition of one module type 7410 to the circuit).

Shift Registers

We shall now study the action of a shift register to ensure understanding of the basic principle. Fig 9.4 shows a four-stage register containing the number 1011 and the successive stages involved in entering the number 1100 into the register from the left, commencing with the rightmost digit of the new number.

At each step every stage of the register takes up the state of the stage on its left and passes its own state on to the stage on its right.

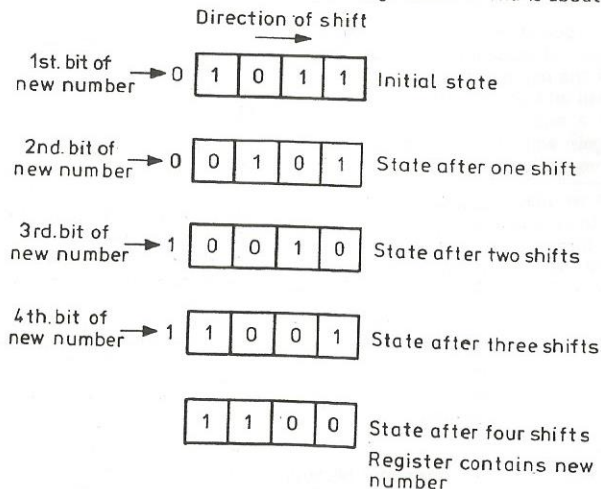


Fig.9.4 Shift register action

Note two important things from fig 9.4.

- (a) The rightmost stage in the initial state and after each of the first three shifts contains the digits of the original number in sequence, rightmost first. In other words as the new number is entered the old number appears serially in the rightmost stage thus giving serial output.
- (b) The old number, once it has passed through the rightmost stage, is lost.

Exercise 9.2

Draw diagrams similar to fig 9.4 for the following:-

- (a) Old number 0 0 0 0, new number 0 1 1 1, right shift.
- (b) Old number 1 1 1 0, new number 1 0 0 0, left shift.

In (b) the new number will be presented left-most bit first.

How can the shift registers like those of fig 9.4 actually be implemented and what kind of flip-flop would be suitable?

To answer this we must take a close look at the behaviour required of a typical stage for all possible conditions. These conditions are as follows:-

- (a) stage holds '0' and is about to accept a '0'
- (b) stage holds '1' and is about to accept a '1'
- (c) stage holds '0' and is about to accept a '1'
- (d) stage holds '1' and is about to accept a '0'

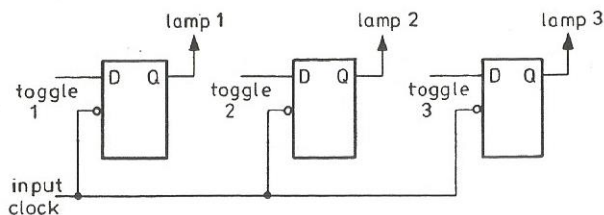


Fig. 9.5

Suppose we try to use the D flip-flops of fig 9.2 for this, connecting the output of each stage to the D input of the stage on its right and feeding all stages with a common clock pulse.

Practical 9.3

On your three-bit register either remove the output gating or set the output clock line permanently to '1'. The general configuration will then be as fig 9.5, where a rectangular symbol for the D flip-flop has been used for simplicity.

Now use the toggles and the clock to enter the number 100 into the register. When this is done and without switching off, reconnect the D inputs of stages 2 and 3 to the Q outputs of stages 1 and 2 respectively instead of to the toggles. Set toggle 1 to '0' and apply a clock pulse.

You would expect the resulting pattern to be 0 1 0 if the right shift occurred as it should.

● Q9.2 What actual result did you find?

Repeat the test but this time leave toggle 1 at '1' when applying the clock pulse. You should get a pattern 1 1 0.

● Q9.3 What do you actually get?

You will have discovered by now that the shift register constructed of these D type flip-flops does not behave as hoped for. Instead all stages become '1' or '0' depending only on the state of toggle 1.

The reason is not difficult to see; as soon as the clock becomes '1' all stages of the register are effectively connected together in a continuous chain so that the input to the leftmost stage

(toggle 1) is rapidly propagated to all stages. What is needed is a flip-flop which can remember temporarily the new state being presented to it whilst retaining its old state at the output to present to the next stage.

● Q9.4 Can you suggest which of the types studied in Assignments 5 and 6 might be suitable?

The answer to this is:-

(a) Edge-triggered D type (or JK type connected as a D type.

These will work because the output responds to the input only during the clock leading edge. Once the clock is fully up the input can change without affecting the output.

(b) Master-slave JK type connected as a D type.

This is suitable because the 'master' flip-flop remembers the input whilst the clock is up but passes it on to the output only at the trailing edge of the clock.

As there are no edge-triggered D type modules in a basic INTIKIT we shall concentrate on the use of the master-slave JK module 7472 connected as a D type (by making $J = D$ and $K = \bar{D}$).

Practical 9.4

Remove all modules and leads from the deck and insert instead four type 7472 and one 7404, connecting them as in fig 9.6. (In this figure the J and K AND gates are not shown for simplicity. Any one of the three inputs may be used, leaving the others unconnected).

The FEEDBACK INTIKIT

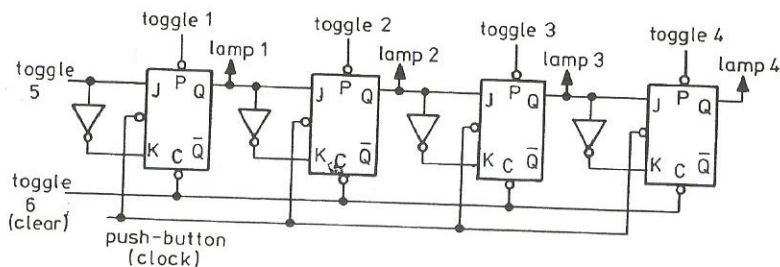


Fig. 9.6 Basic right shift register

The flip-flops have separate preset and clear inputs (which were not used when you were studying the JK in Assignment 6) and these may be employed as in the diagram to provide parallel inputs (ungated) and to clear the register. They must be left at binary '1' whilst a clock is applied and are taken to '0' to achieve preset or clear.

Proceed as follows to examine the various forms of shift register listed earlier in the Assignment.

(a) Serial in — serial out

In this case the initial state of the register is of no importance so we can leave all the preset inputs and the clear line at '1'.

Set toggle 5 successively to the four digits you wish to enter (rightmost first) and after each setting apply one clock pulse. After four pulses the desired number will be in the register. This is serial input.

Now, observing the lamp on the rightmost stage and setting toggle 5 to '0', apply three more clock pulses when the stage will take up successively the four digits of the stored number. This is serial output.

Repeat for various numbers.

(b) Serial in — parallel out

This is the same as (a) but when considering output all lamps are observed simultaneously instead of only the rightmost one.

(c) Parallel in — serial out.

For this mode the register is first loaded using the clear and preset lines. (Clear all stages and then preset those which are to be '1'). Then three clock pulses will present the bits in turn at the rightmost position.

● Q9.5 How can you clear the register without using toggle 6, the clear line?

● Q9.6 How could you eliminate three of the four inverters of fig 9.6?

We have now encountered all of the main register modes but there are some refinements which are worth examining.

Practical 9.5

Gated parallel inputs

To the register you have constructed add gating of the parallel inputs as in fig 9.7, using two 7400 modules as shown. Study the circuit for a moment; when the INPUT CONTROL is at '0' both preset and clear inputs of all stages are held at '1' but if the CONTROL is at '1' the preset and clear inputs are complementary and controlled by the inputs A B C D, which will come from the toggles 1 - 4.

Set the serial input, toggle 5, and the CONTROL input to '0'. Set the desired pattern to be entered on the toggles 1 - 4 and then set CONTROL to '1' and back to '0'. Then apply clock pulses to obtain the serial output.

Exercise 9.3

Write down the logical expression for the CLEAR and PRESET inputs of stage A in fig 9.7 using the symbol X for the control line.

Reversible Shifting

Sometimes an application calls for a register in which the contents may be shifted in either direction, the direction to be selected by application of a 'direction control' signal, say '1' for right shift and '0' for left.

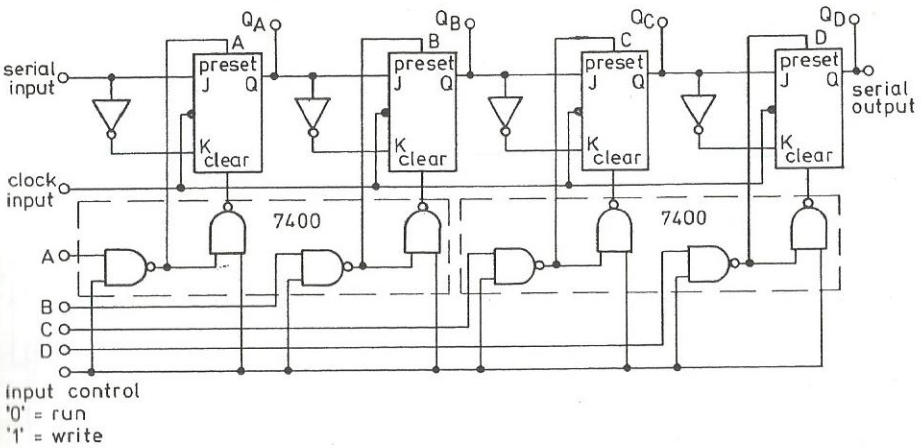


Fig.9.7 Register with input gating

The logic is very simple;

Let E be the direction control signal

$D_n = D$ input of the nth stage

Q_{n-1} = output of stage to the left of n

Q_{n+1} = output of stage to the right of n

Then $D_n = Q_{n-1}$ when $E = 1$ (right shift)

or $D_n = Q_{n+1}$ when $E = 0$ (left shift)

Thus $D_n = \bar{E}.Q_{n+1} + E.Q_{n-1}$

and the logic, which is a straightforward sum of products expression, is shown in fig 9.8.

Practical 9.6

Fig 9.9 shows the logic of fig 9.8 added to the register of fig 9.7. Modify the register you already have to incorporate the additional logic

as shown. When complete you will have a total of 12 modules on the INTIKIT deck. Notice that the circuit now has two input points, one for entry from the left (forward or right shift mode) and one for entry from the right (reverse or left shift mode).

A convenient way to observe the behaviour of this register is to connect it so as to circulate its contents continuously. This is done by coupling the output of the rightmost stage to the forward mode input and the output of the leftmost stage to the reverse mode input. Thus, once a pattern has been set into the register via the parallel entry logic, it may be circulated right or left according to the direction control signal.

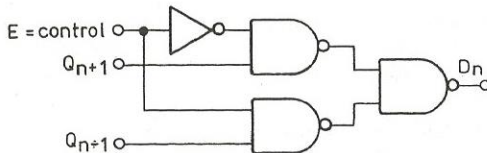


Fig.9.8 Reversible shift logic

The FEEDBACK INTIKIT

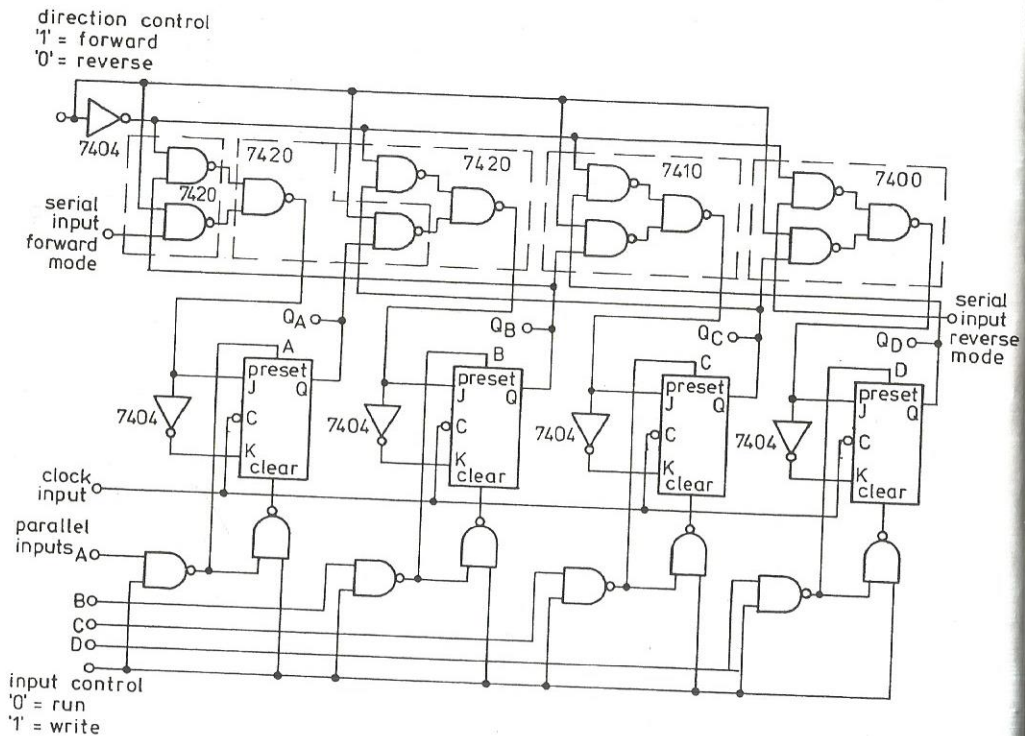


Fig. 9.9 Reversible shift register

Practical considerations & applications

Parallel input-output registers are really no more than simple storage devices which, whilst important, are usually described as such in I.C manufacturers' data rather than as registers.

One limitation on the number of bits which can be given full individual access (input and output) is of course the number of terminations available in a given package. Some examples are

- Type 7475 - Quad D type latch - 14 pin package
- Type 74118 - Hex SR latch - 16 pin simple two-gate latches
- Type 74100 - Dual quad D-type latch - 24 pin package.

Where larger numbers of bits are involved individual access is no longer possible and coded access becomes necessary. To explain this let us assume that an I.C contains 64 bits of storage.

To provide individual access to these, assuming a common clear line, would need 64 set inputs

and 64 outputs. However if the I.C also contained suitable selection logic, inputs and outputs could be applied and observed respectively to a sub-group only of the 64.

For example if the 64 bits were considered as 16 groups of 4 bits each, then we would need 4 inputs, 4 outputs and 4 'address' lines. These latter carry a binary number from 0 through 15 to indicate which group is to be selected. Such an arrangement is usually called a 'random access memory' or RAM for short.

A typical example is type 7489 which is a 64 bit RAM and which, as well as the inputs and outputs listed above, has two control inputs to control the input of data (writing) and the access to outputs (reading). Many other devices are available having different detailed organisation.

Shift registers are also very important system elements and many configurations may be found in I.C form. Pin limitations apply here to the use of parallel input and output but in many applications neither are required and only one serial input and one serial output are needed.

In these cases a large number of stages can be incorporated into one package.

Some examples are

Type 7491 8-bit serial in and out	14 pin
Type 7494 4-bit parallel in, serial out	16 pin
Type 74164 8-bit serial in, parallel out	14 pin
Type 74198 8-bit parallel in, parallel out, reversible	24 pin

Shift register applications are too numerous to be fully listed, but included are

Arit..metic operations (with adders)
 Temporary storage
 Time delay of binary signals
 Counting (see Assignment 10)
 Generation of pseudo-random binary sequences (PRBS)

The last item is an interesting application and whilst it is not possible here to explain fully the meaning of PRBS, the reasons for wanting to generate them or the theory of their generation, it is very easy to set up a demonstration of a typical PRBS. Very briefly PRBS are sequences of binary digits which bear a resemblance in certain respects to random (or 'white') noise and for this reason can be used as source signals in the 'correlation' method of determining the dynamic response of control systems (servo-mechanisms, industrial plant etc).

A matter of general interest is what patterns a shift register will generate when the input to the first stage is a function of (i.e determined by) the states of one or several of its stages. In the circuit of fig 9.10 the input is the non-equivalence (exclusive-OR) function of the first and last stage contents and it so happens that this causes the register to pass through every one of its 16 possible states except the all-zero state, that is it has a cycle length of 15 states. The all-zero state is self-perpetuating.

Practical 9.7

Construct the circuit of fig 9.10; this can be done by removing from your reversible register of Practical 9.6 the reversing logic and substituting the exclusive-OR logic. When constructed, clear the register and apply clock pulses—you will find that no change occurs. Then set any non-zero pattern into the register and apply clock pulses.

Exercise 9.4

Write down the states of the register as they occur. Then try connecting the upper input of the exclusive-OR to stages 2 and 3 successively and noting the sequences generated in each case. For feedback from stage 2 start the register in state 1111, then repeat the trial starting at 0010 and again at 1011. What do you find?

Compare the sequence of '1' and '0' executed by any one stage of the register when feedback is from stage 1 with that executed when feedback is from stage 3. What do you notice?

Further reading

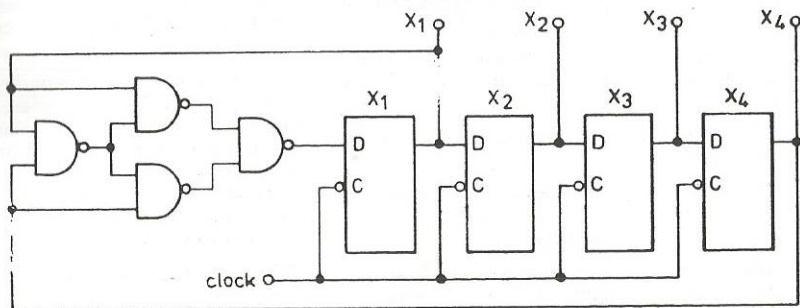


Fig. 9.10 PRBS generator

Discussion and Experimental Procedure

Counters

A counter is simply a circuit which progresses step by step through a series of recognizably distinct states when driven by a series of input pulses. Since each state of the counter must be represented by a set of binary digits all counters must basically consist of collections of flip-flops connected together in various ways. Obviously too the total number of distinct states of the counter (in other words its maximum counting capacity) is a simple function of the number of flip-flops.

$$N = 2^n - 1$$

where N = max count capacity
 n = number of flip-flops

Thus 4 flip-flops can count to 15_{10} etc.

Within this framework there are very many possible counter configurations. The following tables representing different counting progressions possible with three flip-flops, are only a small selection of the possibilities.

COUNT

	a	b	c	d	e	f
0	000	100	000	000	111	000
1	001	010	001	001	110	100
2	010	001	010	011	101	110
3	011	100	011	010	100	111
4	100	ETC	100	110	011	011
5	101		101	111	010	001
6	110		000	101	001	000
7	111		ETC	100	000	ETC
8	000			000	000	
	ETC			ETC	ETC	

In counter (a) the digits follow an ordinary numerical sequence in which the rightmost bit represents $2^0 = 1$, the centre bit $2^1 = 2$, and the left most $2^2 = 4$, and returns to the start on the 8th count.

Counter (b) only goes through three states before going back to the start on the 3rd count.

Counter (c) is a pure binary counter repeating on the 6th count.

Counter (d) is in a unit-distance code repeating on the 8th count. (See Assignment 13 for an explanation of unit-distance codes).

Counter (e) is in reverse binary i.e it counts down, but also stops at 000 and does not repeat.

Counter (f) is a 6 state counter but different from counter (c).

- Q10.1 If you have carried out Assignment 9, can you recognize a common feature in counters (b) and (f)?

Exercise 10.1

Write down some arbitrary counter sequences of your own along the lines of the table above.

Unused States

We have just seen that some counters do not go through all the possible states. Counters (b), (c) and (f) were examples of this. Some counters too will always be started in some predetermined state, like counter (e).

But any counter which is required to repeat continuously but does not use all the possible states is in danger of starting off in one of the unused states and never getting into the desired sequence.

In these cases it is necessary to design the counter so that if it should start in one of these states when first switched on, it will be directed to one of the desired states after a few input pulses. Later on we shall see how to do this.

Synchronous and Asynchronous Counting

Since all counters contain flip-flops of some sort, each flip-flop or stage will need a drive or clock pulse applied to it. The actual behaviour of the stage depends, as we saw in Assignments 5 and 6, on what inputs are applied at D, J, K etc.

When the clock inputs of all stages are driven from the input pulses the counter is called 'synchronous'.

When the input pulses are applied to only one stage (nearly always the 'first' or least significant stage) the counter is called 'asynchronous'. Because in this case later stages receive their clock pulses indirectly from the earlier stages such counters are often referred to as 'ripple-through'.

The very simplest type of counter uses sequence (a) above and happens to be an asynchronous type so we shall start this assignment by building one, but the design of asynchronous counters for other sequences is slightly more difficult than that of synchronous counters. The remainder of the assignment will therefore be devoted to a study of synchronous counter design.

Types of flip-flop used in counting

The basic requirement of a flip-flop for counting purposes is that it must be able to change state ('0' to '1' or '1' to '0') on receipt of a clock pulse.

The FEEDBACK INTIKIT

If you refer back to Assignments 5 and 6 you will find that the types which can do this are

- Edge-triggered D type (with D fed from \bar{Q})
- Master-slave JK type (with $J = K = 1$)
- Edge-triggered JK type (with $J = K = 1$)

In this Assignment we shall use master-slave JK flip-flops type 7472 throughout.

Practical 10.1

The very simplest possible counter consists of a series of flip-flops, each with $J = K = 1$, connected in cascade with the clock input of the first stage fed from the input pulses and that of each subsequent stage fed from the Q output of the previous stage.

This is shown in fig 10.1; notice that the flip-flop symbols are the opposite way round from the normal in order to put the least significant bit at the right to conform with the conventional way of writing numbers.

Construct this counter on INTIKIT using type 7472 for the flip-flops and leaving all J and K inputs unconnected so as to effectively apply binary '1' to them. Use the left-hand push-button Q output for the input pulses and a toggle for the reset line.

Switch on power and set the toggle from '1' to '0' and back again to reset the counter to all zeros. Now apply clock pulses, recording in the table below the states of all stages after each pulse.

Input Pulse	A	B	C	D	Input Pulse	A	B	C	D
0	0	0	0	0	9				
1					10				
2					11				
3					12				
4					13				
5					14				
6					15				
7					16				
8					17				

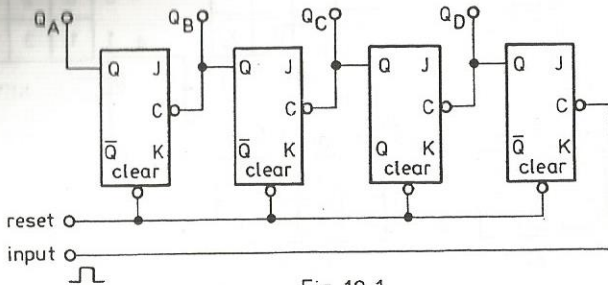


Fig. 10.1

- Q10.2 Does the counter state change as you press the button (leading edge of input pulse) or as you release it (trailing edge)?

Master-slave JK flip-flops change state at the trailing edge of the clock pulse. Therefore, on examining the table of states you should find that:-

Stage D changes state every time the input goes from '1' to '0'

Stage C changes state every time D goes from '1' to '0'

Stage B changes state every time D goes from '1' to '0'

Stage A changes state every time B goes from '1' to '0'

This generates the normal binary count pattern.

Exercise 10.2

Study the table you constructed in Practical 10.1 and, considering the state changes required to count backwards, complete the following

Stage D changes state every time the input goes from '1' to '0'

Stage C changes state every time D goes from to

Stage B changes state every time goes from to

Stage A changes state every time goes from to

Practical 10.2

Remembering that when Q goes from '1' to '0', \bar{Q} goes from '0' to '1', modify your binary counter to count down instead of up and confirm that it does so.

- Q10.3 Can you see how the counter could be arranged to count downwards without any alteration to the internal connections?

Synchronous Counters

The counter of fig 10.1 was a ripple-through or asynchronous counter. The state changes do not occur simultaneously but 'ripple' along the counter at a finite speed dependent upon the reaction speed of the logic employed.

Often it is desirable that all state changes should occur as nearly as possible simultaneously and in this case synchronous counters are employed in which the input pulses are applied to the clock inputs of all stages.

The method by which synchronous counters are designed is briefly to determine from the bit sequence desired what signals must be applied to the J and K inputs of all stages at every step in order to produce the requisite next state. The next state possibilities are listed in the table below, together with the J and K signals which produce them. In this table X indicates a 'don't care' condition as explained in Assignment 4. You may also need to remind yourself of the truth table for the JK flip-flop from Assignment 6 to fully understand the table below. This table is very important and worth memorizing.

Present State	Next State	J	K
0	0	0	X
0	1	1	X
1	1	X	0
1	0	X	1

		AB			
C		00	01	11	10
	0		1	1	1
1		X	X	X	X

$J_C = 1$

		AB			
C		00	01	11	10
	0		X	X	X
1		1	1	1	1

$K_C = 1$

		AB			
C		00	01	11	10
	0		0	X	X
1		1	X	X	1

$J_B = C$

		AB			
C		00	01	11	10
	0		X	0	0
1		X	1	1	X

$K_B = C$

		AB			
C		00	01	11	10
	0		0	2	6
1		1	3	7	5

KEY

		AB			
C		00	01	11	10
	0		0	0	X
1		0	1	X	X

$J_A = BC$

		AB			
C		00	01	11	10
	0		X	X	0
1		X	X	1	0

$K_A = BC$

If you are still uncertain about this table set up a single JK flip-flop with toggles on the J and K inputs and verify it experimentally.

Three-stage Natural Binary Synchronous Counter

We shall now apply this method to the design of a three-stage counter to realize the natural binary sequence (sequence (a) in the table at the start of this Assignment). The sequence is repeated here

Count	A	B	C
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

We first construct Karnaugh maps for each J and K input, each cell in the map corresponding to one counter state. The entries in the maps show the J and K inputs required to ensure that the present state leads correctly to the next when a clock pulse occurs. These maps are shown in fig 10.2.

Here are two examples of the determination of the entries in fig 10.2.

Fig.10.2 Maps for binary counter

The FEEDBACK INTIKIT

(a) At count 3 the next pulse must cause:

C to go from '1' to '0', therefore

$$J_C = X \quad K_C = 1$$

B to go from '1' to '0', therefore

$$J_B = X \quad K_B = 1$$

A to go from '0' to '1', therefore

$$J_A = 1 \quad K_A = X$$

(b) At count 6 the next pulse must cause:-

C to go from '0' to '1'

$$J_C = 1 \quad K_C = X$$

B to go from '1' to '1'

$$J_B = X \quad K_B = 0$$

A to go from '1' to '1'

$$J_A = X \quad K_A = 0$$

When all entries have been made loops are drawn to obtain the simplest possible expressions for J and K and these are shown under the maps. The resulting logic circuit is in fig 10.3 where, because type 7472 JK flip-flops are provided with AND gates at each J and K input, no additional gating is needed.

Practical 10.2

Construct the counter of fig 10.3 using three type 7472 modules and check its operation.

A Modulo - 6 Counter

Counter (c) in the table at the start of this assignment had 6 states and progressed up to 101 in normal binary code. The bit sequence is repeated here

Count	A	B	C
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	0	0	0

Exercise 10.3

Apply the method that has just been explained to this sequence to complete the Karnaugh maps of fig 10.4 and hence derive the simplest expressions for the J's and K's.

Notice that because this counter returns to zero from count 5 the two states

$$ABC = 110 \text{ and } 111$$

should never occur. That is they are 'can't happen' conditions and may be filled with X as shown. However, after finding the simplest expressions for J and K we must check to make sure that if, by chance, the counter should start in one of these two states, it will eventually enter the desired sequence. If it proves not to it will be necessary to replace the X's in some or all of the can't happen states by 0's or 1's to ensure correct operation and to amend the logic accordingly.

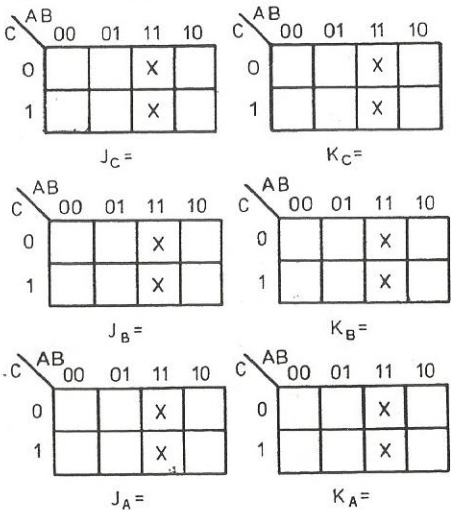


Fig. 10.4 Maps for modulo - 6 counter

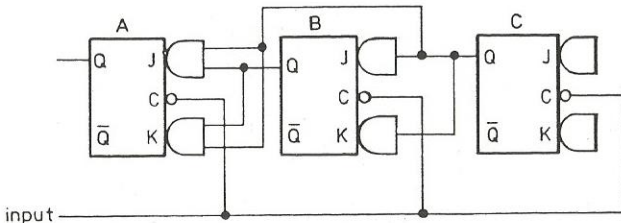


Fig. 10.3 Synchronous binary counter

You should have obtained the following expressions.

$$\begin{aligned} J_C &= K_C = 1 \\ J_B &= \bar{A}C & K_B &= C \\ J_A &= BC & K_A &= C \end{aligned}$$

Let us use these to see what happens if the counter starts at ABC = 110 (one of the redundant states)

Present State								Next State			
A	B	C	J_A	K_A	J_B	K_B	J_C	K_C	A	B	C
1	1	0	0	0	0	0	1	1	1	1	1
1	1	1	1	1	0	1	1	1	0	0	0

Thus the counter returns after two input pulses to state ABC = 000 and no lock-up can occur.

The logic circuit is shown in fig 10.5.

Practical 10.3

Construct and check the counter of fig 10.5. Using the preset inputs deliberately set the counter into state ABC = 110 to confirm that it will return to zero properly.

Exercise 10.4

Perform the complete design of a counter to execute the sequence (d) in the table at the start of the assignment. Making full use of the \bar{Q} outputs and the J and K input AND gates you will need one 7400 module in addition to the flip-flops. Hint: To avoid errors in the mapping stage construct a key like that used in fig 10.2 (similar, not the same).

Practical 10.4

Construct the resulting circuit and test it.

Exercise 10.5

Design a synchronous counter to perform B C D counting, that is, counting to 9 in natural binary and then returning to 0. This will of course require four flip-flops A B C D and will have $16 - 10 = 6$ redundant states which you must check will lead into the desired sequence. The Karnaugh maps for J_A and K_A are given in fig 10.6. Draw the remaining maps, derive the logic and the logic circuit. Hint: No logic additional to the J and K input AND gates should be needed.

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

plot of J_A
 $J_A = BCD$

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

plot of K_A
 $K_A = D$

Fig. 10.6 Maps for BCD counter

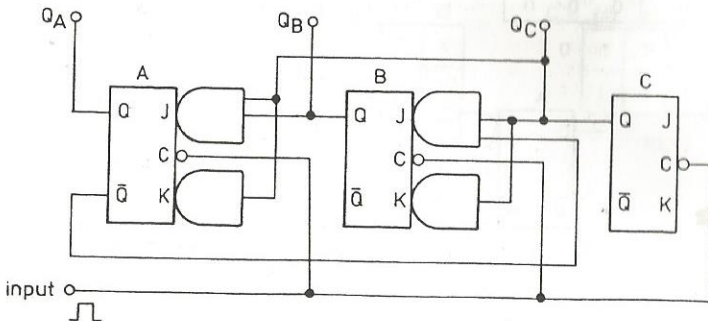


Fig. 10.5 Modulo 6 counter

The FEEDBACK IN TIKIT

Practical 10.5

Construct the BCD counter and check its behaviour, especially when forced into any of its six redundant states.

Practical considerations & applications

The reason for using synchronous as opposed to asynchronous counting is worth some further consideration.

Counters are very frequently used in conjunction with decoding logic to energize a number of lines in sequence. Imagine two such lines which will be selected as follows:

	A	B	C
Line 3	0	1	1
Line 4	1	0	0

As the counter progresses from 3 to 4, unless all stages change state simultaneously it is possible to get brief existence of quite different states. For example in an asynchronous counter the actual sequence of changes between 3 and 4 would be

3	0	1	1
	0	1	0
	0	0	0
4	1	0	0

Thus lines 2 and 0 could be spuriously selected for a short time. The more stages in the counter the worse is the effect.

In a synchronous counter this effect is greatly reduced because all stages change state virtually simultaneously. However, since absolute simultaneity is not possible it can still happen that very brief spurious signals appear. Because they are very brief they can often be suppressed by moderate capacitive loading but if no spurious signals of any length can be tolerated it is usually necessary to do one of two things:-

- Redesign to use a counter progression in which one only bit changes at each step. You built such a counter in Practical 10.4. Obviously in such a counter no spurious transitional states are possible.
- If solution (a) is not practicable for some reason the various output lines are AND gated by the clock signal so that no signals can appear on them, until the transitions have all been completed. This is illustrated in fig 10.7

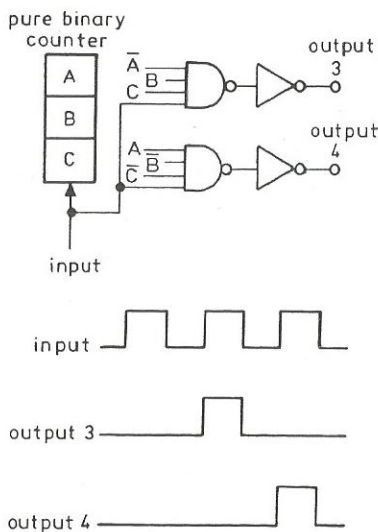


Fig. 10.7

The counter state changes at the trailing edge of the input but no output appears on the selected line until the input goes up again. The disadvantage is that there are periods between input pulses when no output appears on any line.

Counting Speed

The use of synchronous counting does not in itself increase the potential speed of the counter because though the state changes do not have to ripple through the counter, nevertheless the logic states applied to the J and K inputs to control successive state changes must be allowed to propagate along before the next input pulse is applied. Usually, however, these signals have to propagate through one gate only whereas the propagation time of an asynchronous counter is the sum of delays in all stages so that in practice synchronous counting does turn out to be somewhat faster than asynchronous for the same circuit elements.

The applications of counters are very numerous but broadly they fall into two categories.

- Signal distribution using decoders (as in fig 10.7) and similar applications.
- Frequency division where the need is to generate one output pulse for every N input pulses.

With the methods studied in this Assignment you are in a position to design any counter

sequence you wish. You may wonder why it is not possible simply to maintain a catalogue of designs for all purposes and the answer lies in the magnitude of the numbers involved. Consider a three-bit counter; there are eight possible different states and if we assume that the design will use all of these the number of ways of arranging them in a sequence is:-

$$\begin{aligned}\text{Factorial } 8 &= 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \\ &= 40,320\end{aligned}$$

This number is further increased if we include all sequences using less than eight states.

However, some sequences are merely others with their ABC columns rearranged whilst every sequence has a counterpart in which one, two or all of its outputs are simply complemented. Thus the number of essentially different designs is reduced. The table below shows some of the sequences which are basically similar to the pure binary sequence.

ABC	$\bar{A}\bar{B}\bar{C}$	$A\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	BAC	ACB	$\bar{A}CB$
000	100	010	111	000	000	100
001	101	011	110	001	010	110
010	110	000	101	100	001	101
011	111	001	100	101	011	111
100	000	110	011	010	100	000
101	001	111	010	011	110	010
110	010	100	001	110	101	001
111	011	101	000	111	111	011

There are 6 ways of arranging 3 columns in order and 8 ways of complementing none, some or all of the columns, so the number given above is divided by $6 \times 8 = 48$ to give 840 essentially different designs.

Now work out for yourself the possible sequences for a four-bit counter and you will find the number to be astronomical. This is why you need to have the design technique at your finger-tips.

Design using other types of flip-flop

Although no discussion was included in the Assignment you can, of course, apply similar methods to the design of counters using types of flip-flop other than JK, provided you alter the ground rules appropriately.

You could for example use D type flip-flops by applying the rules:-

If next state is to be 0, D must be 0

If next state is to be 1, D must be 1

In general you will find the external logic to be appreciably more complex than when using JK flip-flops and there is little advantage when the latter are freely available.

Further reading

Discussion and Experimental Procedure

In Assignment 10 we saw the basic difference between asynchronous or ripple-through counters and the synchronous type. We also found out how to use Karnaugh map methods to design a synchronous counter using JK flip-flops to execute any desired sequence of states.

It is often possible to apply a slightly modified form of the same method to the design of asynchronous counters, which will sometimes have cost advantages over the synchronous implementation.

The reason for the potential cost advantage is that if some stage derives its clock input from a previous stage instead of from the input, then when the previous stage suffers no change the J and K inputs of the later stage are redundant (don't care). This often permits simplification of the J and K logic beyond what would be possible in a synchronous design, where every stage receives every clock pulse.

However there are some types of sequence for which an asynchronous design is not possible whereas all sequences may be designed synchronously.

The method will be illustrated by some examples.

A Modulo-6 asynchronous counter

In Assignment 10 one of the exercises (E10.3) was to complete the design of a synchronous modulo-6 counter and the resulting logic, shown in fig 10.5, used three type 7472 JK flip-flops with J and K input AND gates, two of which were actually used as such. We shall now see that an asynchronous design avoids the need for these AND gates so that, if desired, the counter could be constructed using type 74107, which is a dual JK flip-flop with no AND gates, thus reducing the package count.

The first stage of the design process is to set down the desired sequence and to examine it to see which stages must change state at each step and what signals are available to initiate these changes. Remember that a master-slave JK changes state when its clock input goes from '1' to '0'.

Count	Present state			Next state			Flip-flops to toggle
	A	B	C	A	B	C	
0	0	0	0	0	0	1	C
1	0	0	1	0	1	0	BC
2	0	1	0	0	1	1	C
3	0	1	1	1	0	0	ABC
4	1	0	0	1	0	1	C
5	1	0	1	0	0	0	AC
6	0	0	0	0	0	1	C
7	ETC			ETC			

Since stage C changes at every input pulse its clock must be the input signal.

Since B changes always coincide with a '1' to '0' transition, of stage C this can thus provide its clock input. At count 5, however, B must not change and the J and K inputs of stage B will be used to inhibit any change at that point.

Stage A also changes on some (but not all) of the '1' to '0' transitions of stage C, which will thus provide the clock signal with suitable inhibition of unwanted changes.

We now construct the Karnaugh maps as before, first entering X in all cells which correspond to:-

- (a) Redundant states—in this case 6 and 7.
- (b) Present states for which a change to the next state does not generate a '1' to '0' transition of the clock pulse. In this case since both stages A and B are clocked by stage C the present states of which this is true for A and B are nos. 0, 2 and 4. It is not applicable to stage C, of course.

The remaining entries in the maps are made exactly as they were for the synchronous design method, giving the results of fig 11.1.

Exercise 11.1

Verify the maps of fig 11.1 by redrafting them for yourself and comparing your result with the figure. Also draw your version of the logic circuit and compare it with fig 11.2.

Exercise 11.2

Check whether the redundant states 6 and 7 lead properly to the desired sequence by completing the table below. Remember that C will change at every input pulse.

Present state			C =						Next stage			
A	B	C	J _A	K _A	J _B	K _B	J _C	K _C	1*0?	A	B	C
1	1	0	1	1	0	1	1	1	No		1	1
1	1	1	?	1	?	1	1	1	?	?	?	?

The FEEDBACK INTIKIT

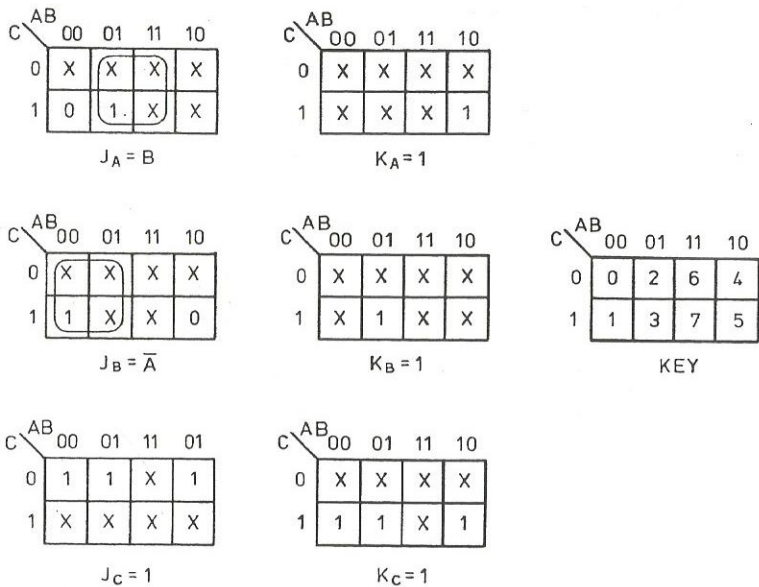


Fig. 11.1

Practical 11.1

Construct the modulo-6 counter designed above and shown in fig 11.2, checking its performance and, by forcing it into the redundant states, that they lead to the desired sequence.

Notice that no J and K input AND gates are used in this circuit.

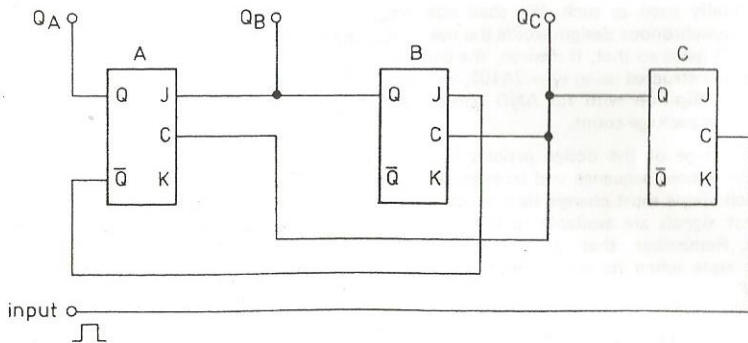


Fig. 11.2 Asynchronous modulo-6 counter

Asynchronous BCD counter

This is to execute the same sequence as the synchronous counter of Exercise 10.5 in Assignment 10.

As before we start by examining the sequence for suitable signals to use as clocks for the various stages.

Dec	Present state				Next state				F/F to toggle
	A	B	C	D	A	B	C	D	
0	0	0	0	0	0	0	0	1	D
1	0	0	0	1	0	0	1	0	C,D
2	0	0	1	0	0	0	1	1	D
3	0	0	1	1	0	1	0	0	B,C,D
4	0	1	0	0	0	1	0	1	D
5	0	1	0	1	0	1	1	0	C,D
6	0	1	1	0	0	1	1	1	D
7	0	1	1	1	1	0	0	0	A,B,C,D
8	1	0	0	0	1	0	0	1	D
9	1	0	0	1	0	0	0	0	A,D

Conclusion

- D may be driven by INPUT
- C may be driven by D
- B may be driven by C
- A may be driven by D

The redundant states are 10 to 15.

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

plot of J_A
 $J_A = BC$

CD \ AB	00	01	11	10
00	X	X	X	X
01	X	X	X	1
11	X	X	X	X
10	X	X	X	X

plot of K_A
 $K_A = 1$

Fig. 11.3 Maps for stage A of BCD counter

The present states for which each stage has no '1' to '0' clock transition are:-

- Stages A and C 0, 2, 4, 6, 8
- Stage B 0, 1, 2, 4, 5, 6, 8, 9
- Stage D None

Entering these first as X and then completing the J and K entries for the remaining states we can construct the Karnaugh maps as before. Fig 11.3 shows the maps for stage A only.

Exercise 11.3

Complete the remaining maps for this counter and then, using the J and K functions so obtained, check that the redundant states lead to the desired sequence. Draw the logic circuit and compare it with fig 11.4.

N.B. When checking the behaviour of the redundant states, remember that you must think about the sequence of events. For example, since D clocks both A and C and C clocks B, if the next input causes $D = 1 \rightarrow 0$ and if J_C, K_C are such that this will in turn cause $C = 1 \rightarrow 0$, then J_B, K_B must be studied to see if the result is a change in B, and so forth.

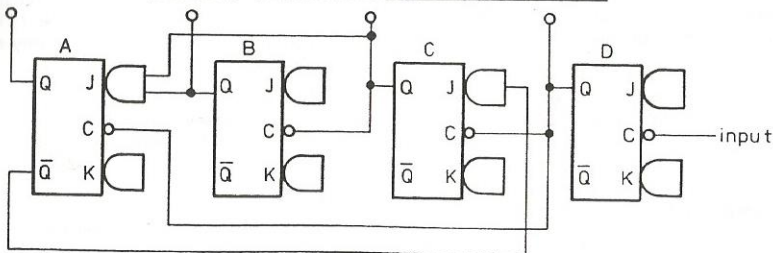


Fig. 11.4 Asynchronous BCD counter using type 7472

The FEEDBACK INTIKIT

Practical 11.2

Construct and test the counter of fig 11.4, including checking the behaviour of the redundant states.

● Q11.1 (a) How many packages does the circuit of fig 11.4 use?

(b) Could you reduce the package count by using type 74107 dual JK flip-flops and additional NAND gates (7400) where necessary?

(c) Look back at your synchronous design for this counter. By using type 74107 and separate gates could an equal simplification to that in (b) be achieved?

An arbitrary sequence

Let us take a look at the following arbitrary sequence with a view to an asynchronous implementation:-

Count	A	B	C
0	0	0	1
1	0	1	0
2	1	0	0
3	1	0	1
4	0	1	1
5	1	1	0
6	0	0	0

Stage A changes at counts 2, 4, 5, 6 etc but whilst stage C = 1 → 0 at count 5 and stage B = 1 → 0 at 2 and 6, neither has a transition at 4. Therefore the input pulses must be used to clock stage A.

Stage B changes at counts 1, 2, 4, 6. A = 1 → 0 at 6 and C = 1 → 0 at 1 but neither has a 1 → 0 transition at 2 or 3. Therefore stage B must be clocked by the input.

Stage C changes at counts 1, 3, 5, 0. A = 1 → 0 at 6 and B = 1 → 0 at 5 but neither has a 1 → 0 transition at 1, 3 or 5. Therefore stage C must be clocked by the input.

In other words all stages must use the input as a clock and the result is a synchronous counter. An asynchronous design is not possible.

Exercise 11.4

Examine the following sequences to see whether an asynchronous design is possible in either case. If so, carry out the complete design for both asynchronous and synchronous modes.

(a) Gray code

(b) Arbitrary

Count	A	B	C
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0
8	0	0	0

etc

Count	A	B	C
0	0	0	0
1	0	0	1
2	1	0	0
3	1	1	0
4	1	1	1
5	0	1	0
6	0	0	0

etc

Practical 11.3

Construct and test any designs arising from Exercise 11.3, comparing the asynchronous and synchronous designs for relative economy.

Practical considerations & applications

The example of asynchronous design in this Assignment showed that minor economic advantage could be achieved over synchronous designs in some cases but that sometimes asynchronous designs are not possible.

Generally speaking the potential advantage of asynchronous design increases with the counter length (number of stages or modulus), which is clearly illustrated by the relative complexity of designs for a pure binary counter.

An asynchronous design for this would have every stage clocked by the previous one with no J, K gating necessary (see fig 10.1 for example).

A synchronous design has the J, K inputs for each stage equal to the AND of all previous stage outputs. Thus an eight-stage counter would need seven input AND gates in its most significant stage (see fig 10.3 for an example).

Since no J, K gates are required for the asynchronous design in this case, type 74107 dual JK flip-flop can be used, considerably reducing the package count.

Because counting is such an important operation there are many integrated circuits devoted just to this end. Some examples are:-

Type 7490 Decade counter (asynchronous)

Type 7492 Divide by 12 counter (asynchronous)

Type 74163 4-bit Binary counter (synchronous)

It makes a valuable exercise to study the maker's data sheets for some of these types and to follow their logic, which in some cases is unlike anything so far included in this manual.

For instance, type 7490 can be redrawn to show it as a modulo-5 counter preceded by a single modulo-2 stage to give modulo-10. The modulo-5 section has two stages (A and C) clocked by the input and the third (B) from stage C so it is an asynchronous design.

N.B. The 74 series data for counters always has the stages lettered in the reverse order to that used in this manual.

The type 74161, on the other hand, is a synchronous binary counter in which J and K inputs are used only for the synchronous clearing and presetting of the counter and play no part in controlling the progress of the count. The latter is achieved in this type by AND gating the clock signal with the outputs of all previous stages.

Further reading

THE UNIVERSITY OF ZAMBIA

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL & ELECTRONIC ENGINEERING

COURSE:EE 333

LAB-MANUAL

AN INTRODUCTION TO MICROPROCESSORS

CONTENTS

	<u>Page</u>
1) LAB 1: Familiarisation with MAT 385	1
2) LAB 2: Data-transfer and Data-manipulation	3
3) LAB 3: Transfer of control	5
4) LAB 4: Input/Output	8

LAB 1

AIM: Familiarisation with the MAT385(8085 microprocessor-8 bit)

The communication between the microprocessor and "you" is controlled by a "monitor-program" which is provided on the MAT385 permanently in a ROM(locations 0000-07FF),and is started by the "reset" key.

The monitor understands certain commands embodied in the KEYS on the 'key-pad'of the MAT385 such as:

KEY	ACTION PERFORMED
RESET	Pressing this key allows the monitor program to start-off from the same address and the monitor displays '-80 85'.The monitor is now ready to accept your commands.When starting off always press this key.
SUBST MEM	Allows you to examine and/or change the values in memory locations.The monitor responds by displaying a decimal point.It is then ready to accept the 4-digit hex address.
NEXT	To move to the next subsequent location.
EXAM REG	Allows you to examine and/or change values in CPU registers(A-F),stack pointer and program counter.
GO	After entering the program it is desired to run it.The 'GO' command does this.After pressing this key the monitor expects a 4-digit hex address from where the program should start running.
SINGLE STEP	Once a program has been entered using the subst-mem command,it is run using the 'go' command.If errors occur then the single-step command is used to facilitate debugging.This runs one program line at a time and thus the processor status ie registers and memory locations maybe examined after each instruction.
EXEC	Executes the previous command.Hence it is used after all the above commands.

****Please look at the front panel of the MAT385 for a schematic description of how to use the commands stated above.

EXPT 1:

Fill in memory locations 8000-8004 with hexa-decimal values 60, A3,5C,BF,2E.Check to see if the locations actually contain the values you have input.Use 'subst-mem'key.

Try loading the same data in locations 0000-0004.What happens? Explain.

EXPT2:

Fill the program-counter(PC)with 2010,the stack-pointer(SP)with 20C2 and the C-register with 6B.Then check the result ie.if the reg's actually contain the values you input.Use 'exam-reg'key.

EXPT2 Now enter the following program.

<u>ADDRESS</u>	<u>DATA</u>	<u>MNEMONIC</u>	<u>COMMENTS</u>
2000	31	LXI SP 20C2	:INITIALISE
2001	C2		STACK
2002	20		POINTER
2003	06	MVI B,13	:(B)<-- 13
2004	13		
2005	3E	MVI A,40	:(A)<--40
2006	40		
2007	80	ADD B	:(A)<--(A)+(B)
2008	47	MOV B,A	:(B)<--(A)
2009	CF	RST 1	:RETURN CONTROL TO MONITOR

Run this program.

Check the contents of register B.

You may single step through the program to see if the program is doing what you expect it to do.

Input different values in locations 2004 and 2006.Run this program and see if the result you expect is found in register B.

Eg. Is $36+15=51$?If not,why not?

**** READ ASSIGNMENT 0-1 OF MAT385 MANUAL VOL 1.IF TIME ALLOWS.

***** THIS IS AN IMPORTANT LAB .GET TO KNOW HOW TO USE THE KEYS.FUTURE LABS WILL BUILD ON THIS.

LAB 2

AIM: Introduction to programming. To use and appreciate the "data-transfer" and "data-manipulation" instructions.

A microprocessor may be thought of as a device which accepts binary coded information from its 'input-ports', manipulates this information according to a program in its memory and outputs the results to its 'output-ports'. The instructions which constitute the program are some or all of those from the 'instruction-set' of the particular microprocessor, required to perform a particular task.

Instructions:

Instructions of any microprocessor may in general be divided into five groups.

- a) Data Transfer.
- b) Data Manipulation.
- c) Transfer of Control.
- d) Input/Output.
- e) Machine Control.

In this lab, we shall look at the Data-Transfer and Data-Manipulation group.

Data-Transfer:

This involves data transfer between the various processor registers or between a processor register and memory location.

eg.

```
MVI A,data      :(A)←--data
MOV A,B         :(A)←--(B)
LXI H,802D      :(H)(L)←--802D(hex)
                :So H contains 80 and L 2D.
MOV A,M         :(A)←--((H)(L))
```

This means that the contents of the memory location whose address is given by the contents of H and L registers is sent to A.

Data-Manipulation:

Instructions in this group carry out arithmetic and logical operations on data in registers or memory locations.

eg.

```
ADD B           :(A)←--(A)+(B)
SUB C           :(A)←--(A)-(C)
ADI,data       :logical "AND"(A)and data.
DCR M          :((H)(L))←-- ((H)(L))-1
ADC B          :(A)←-- (A)+(B)+(CY)
```

EXPT 1

Write a program that reads three, 8-bit numbers from memory, adds the first two and subtracts the third one from the sum of the first two and then stores the result in memory.

Run this program for several values.

EXPT2:

Write a program that adds 2 numbers in double precision arithmetic (this means that 16-bits are used for numerical values instead of 8). The numbers are contained in register pairs BC and DE. Check your result for several values.

Would you get the correct answer if the first two values were E6 and 2A. Explain.

***Take care of the carry that results, since only 8-bit addition is carried out in the accumulator.

Follow the format below to code your program.

MEMORY		ASSEMBLY			
Address	Data	Label:	Mnemonic	Operands	:Comments
~	✓	X	✓	X	~

***You are encouraged to choose any appropriate instruction from the instruction set and experiment with them.

For queries consult Assignment 3-6. of the MAT385 VOL1

LAB 3

AIM: Further programming. To understand the concept of transfer of control within a program and its use.

The flexibility and versatility of computers rests on the ability to transfer program control to an instruction that is not in sequential order.

If some instructions are to be executed repeatedly then it is necessary to jump to the first instruction once those instructions have been executed and so on. Obviously the address of the first instruction is not sequential to the last one, and hence a branch must occur.

A branch may be **conditional** or **non-conditional**.

Non-Conditional Branch: In this case program control is transferred to another location without any pre-conditions. This is done by loading the address of the next desired instruction into the program-counter.

eg.

JMP 20FE : Transfers control of program to location 20FE.

JMP START: : Transfers control to symbolic address START.

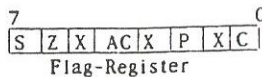
<u>ADDRESS</u>	<u>LABEL:</u>	<u>INSTRUCTION</u>
....
2010	START:
....
....	JMP 20FE
....
....
20FE
....
....	JMP START:
....

**** A label is a symbolic address.

Conditional-Branch:

This transfer takes place only if a certain condition is satisfied else the program continues sequentially

There exists a flag-register in the microprocessor. This register is an indication of the state of the microprocessor. You can use this register for conditional branching. This results in the flag register being examined to see if the branch should be executed.



S-Sign bit. Z-Zero. AC-Auxillary Carry. P-Parity. C-Carry

Subroutines can also be nested, i.e. A subroutine within a subroutine.

*** Since the program has to know where to return to after executing the subroutine it stores the value of the address next to the CALL instruction on the "stack". On encountering the RET instruction it loads the address into the program counter and thus returns to the instruction following the CALL.

EXERCISES

EXPT 1:

Use single-step execution to see the action of the unconditional jump instruction below.

<u>ADDRESS</u>	<u>INSTRUCTION</u>
2800	JMP 2810
.	.
.	.
2810	JMP 2800

What do you think is changing?

EXPT 2:

Write a routine that inputs values 01.....10 into the memory locations 8020 to 8029.

EXPT 3:

Using subst-mem key fill locations 2000-2004 with five arbitrary values. Now write a program that compares the values in the five locations and puts the largest of them in register C.

Hint: Use Compare instructions.

***For queries look at Assignments 7-8 of MAT 385 Manual VOL1.

LAB 4

AIM: To realise input/output on the 8085 microprocessor - the most important feature from the users viewpoint.

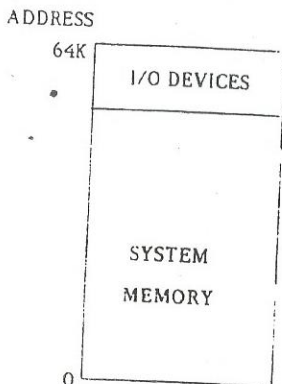
A program resident in memory may require data input to process, and the user more often than not wishes to see the result of such an operation. Hence the need to be able to input and output data.

Input/Output may be **analog** or **digital**. For analog input/output the input has to be converted to digital form before it can be processed by the microprocessor (a digital device) and then the digital output has to be converted to analog form. No such problems exist for digital data.

Input/Output may be implemented in one of two ways :-

Memory Mapped IO:

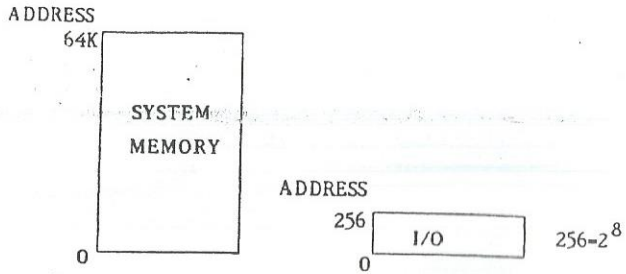
Here a part of the memory is used for input/output. Hence, each input/output operation is similar to a memory-access and all the instructions used for memory-access can be used for input/output. However this ties-up part of the systems memory (since it is set aside for I/O).



Programmed I/O

In this case input/output is achieved by special instructions executed by the processor ("IN and OUT" for the 8085). The processor outputs a request for I/O devices on the bus, indicating to them that the address on the address-bus is for an I/O device. In this way the system does not tie-up memory, and a larger space is available for programs. However this prevents us using the memory instructions.

The system memory is then as shown below.



The 8085 uses the latter option -programmed I/O, although it is possible to configure it for memory-mapped I/O.

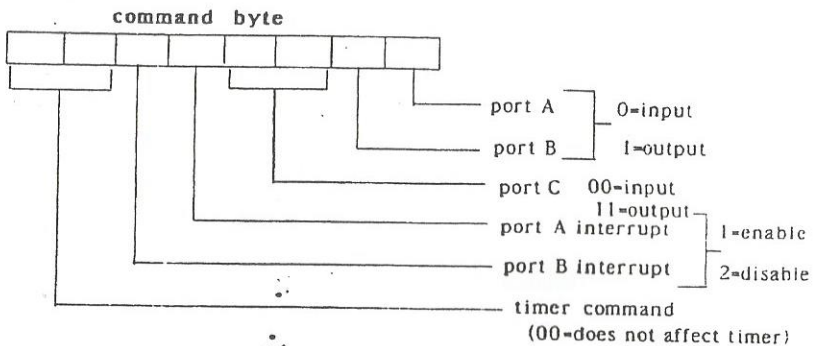
Port Initialisation:

I/O on the MAT is organised as several 8-bit ports. A maximum of 256 ports is possible (we don't need that many!). This assignment is concerned mainly with two of them.

PORT 21: 8 Input switches on the front panel of the MAT.

PORT 22: 8 "Led's" acting as outputs on the front panel.

These ports are programmable as "Input" or "Output" ports and are incorporated into an IO/TIMER memory chip (Intel 8155). To configure these ports in a desired manner (ie. which shall be input and which shall be output), a 8-bit command is sent to the command register (addr 20) which resides in the 8155. Each of the 8-bits has a particular significance.



The command is transferred to 8155 using the "OUT" instruction as demonstrated below.

```
MVI A,0EH :0E=0000 1110
OUT 20H :Send to command register.
```

This sets port A to be 8-outputs, port B as 8-outputs, port C to be an output port and doesn't affect the timer.

Can you see how this happens?

Some common addresses:

ADDRESS(hex)	USE
20	command reg
21	port A data
22	port B data
23	port C Control

TIPS :

- The accumulator is always involved in IN and OUT instruction.
- To be able to single-step in an input/output program save the accumulator in location 20FF, since the monitor routine changes the value of the command byte when single-step is used

EXERCISES

EXPT 1:

Do exercise 9.1 of Assignment 9 in MAT 385 manual VOL 1.

EXPT 2:

Write a program that reads the input from the switches, logically "OR's" it with 1010 1010 (or any other value) and outputs the result to the LED's. Write it such that you can continuously change data using the switches.

EXPT 3:

Write a program that expects two 1-byte numbers from the switches, adds them and displays the result on the LED's. Write it such that you can continuously input data. Extend the program to include a section so that what is shown on the LED's is also shown in Data -field of the display.

Hint: Use a monitor sub-routine UPDDT(address 034E) and take care to save the status of your processor before entering the sub-routine.

... For queries consult Assignment 9 of MAT 385, VOL 1.