Object

To become familiar with the logical operations of AND, OR, NOT, NOR and NAND and the use of truth tables.

Equipment required

Quantity	Equipment
1	Intikit deck CK353 and leads
1	Module 7400 quad NAND
1	Module 7404 hex inverter
1	Module 7410 triple NAND

Approximate time required

One hour

Prerequisites

Assignment 1 or a basic understanding of binary numbers.

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 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.

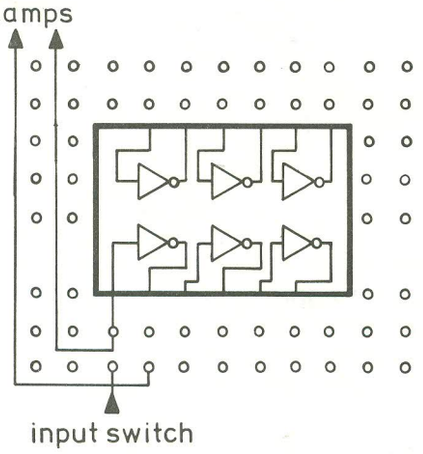


Fig. 2.1

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'.

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

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

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

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

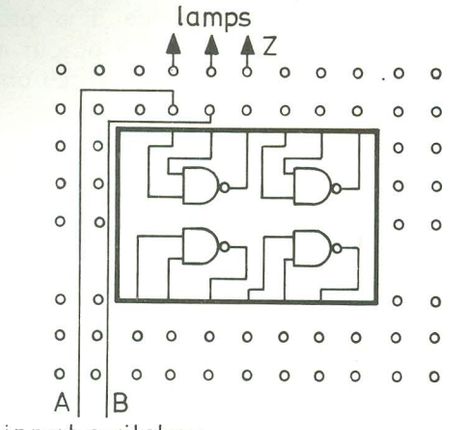


Fig.2.2

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

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

NOT-AND is usually abbreviated to NAND so we have

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

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

This in turn gives us

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

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 $\overline{Z}$. Add another column for $\overline{Z}$ 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

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

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.

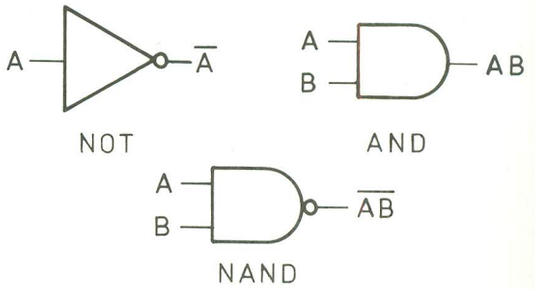


Fig. 2.3

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

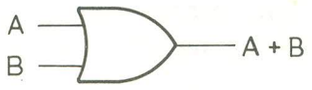
$$Z = A + B$$

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

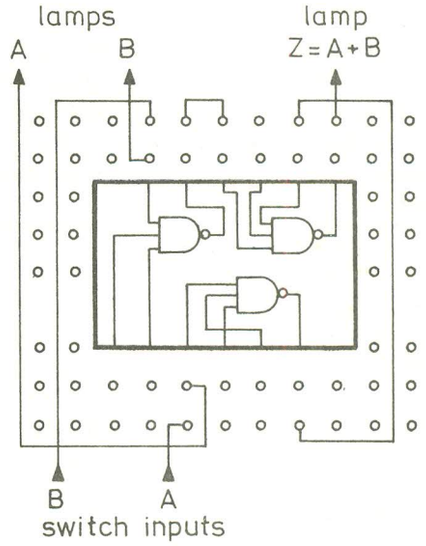
What do you notice about the contents of the last two columns? You should find that they are identical. Since all possible conditions are described in the table we can write

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

But the left-hand side of this equation is actually

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

so that, by first negating both inputs and applying $\overline{A}$ and $\overline{B}$ 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.



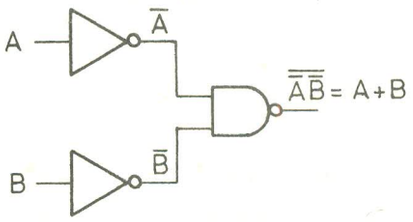


Fig. 2.5.

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

$$Z = \overline{A.B.C} = \overline{A.1.1}$$

But we found above that

$$0.1 = 0$$

and

$$1.1 = 1$$

Therefore $A.1.1 = A.1 = A$

so that $Z = \overline{A}$

Similarly if only input C is unconnected

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

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

In the derivation of the OR operation above we saw that

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

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

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

Examples:

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

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

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

Exercise 2.3

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

(a) $\overline{A\ B}$

(b) $\overline{\overline{A} + \overline{B}}$

(c) $A + \overline{B}$

(d) $\overline{A + B}$

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.

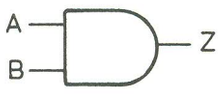
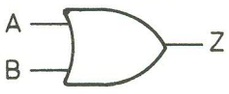
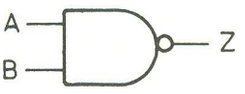
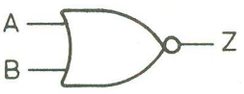
Exercise 2.5

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
0	0	0
0	1	0
1	0	0
1	1	1

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

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

A	B	Z
0	0	1
0	1	0
1	0	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

Object

To study the algebraic representation of general logic operations using AND, OR, NOT and their realisation with NAND modules.

Equipment required

Quantity	Equipment
1	INTIKIT deck CK353 and leads
1	Module 7400
1	Module 7404

Approximate time required

One hour

Prerequisites

Assignments 1 and 2 or an understanding of the operations AND, OR, NOT, NAND plus the basic postulates and concepts of Boolean algebra including De Morgan's theorem.

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

If A = 0 Z = 0 + 0 = 0

If A = 1 Z = 0 + 1 = 1

Therefore Z = A and 0 + 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) $A + \overline{A} = ?$, $A.\overline{A} = 0$

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 unfactored 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

$$A\,B + A\,C = A(B+C)$$

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.

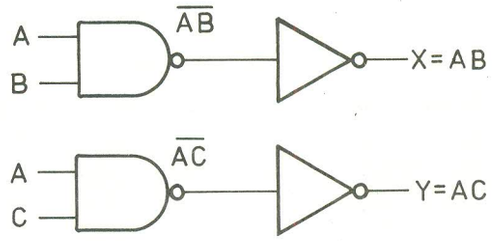


Fig. 3.1

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

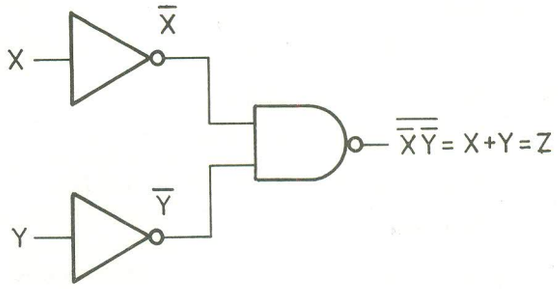


Fig. 3.2

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

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

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

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.

$$(A + B + ...)\ (C + D + ...) =$$
$$AC + AD + BC + BD + ...$$

$$AB + CD + ... = (A+C)\ (A+D)\ (B+C)\ (B+D)\ .....$$

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.

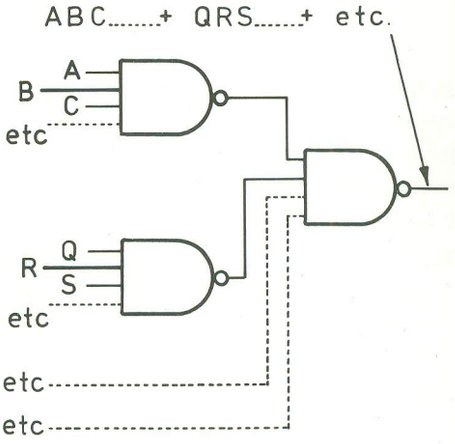
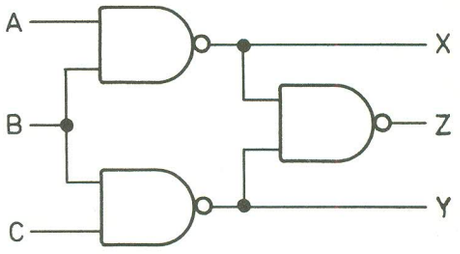


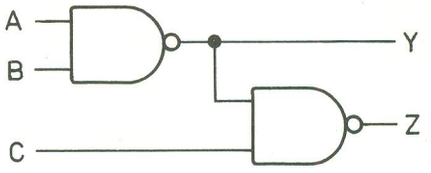
Fig. 3.3

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 B. We need to know how to apply it to expressions such as

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

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

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

Now $Z = Y \text{ NAND } C = \overline{AB} \text{ NAND } C$

$= (\overline{A} + \overline{B})\, C = A\, B + \overline{C}$ from above

● *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 $(\overline{A} + \overline{B})\, C$ and for $A\, B + \overline{C}$ 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 $(\overline{A} + \overline{B})C$ 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) $\overline{(A + \overline{B})\,(\overline{C} + D)} = \overline{A}\, B + C\, \overline{D}$

(b) $\overline{A + B\,(C + \overline{D}\, E)} = \overline{A}\,[\overline{B} + \overline{C}\,(D + \overline{E})]$

Exercise 3.6

Write down sum of products expressions for the complements of

(a) $(A + B)\,(C + D)\,(B + C)$

(b) $\overline{(A + B)C} + \overline{(\overline{A} + \overline{B})\overline{C}}$

simplifying the results as much as possible.

Exercise 3.7

Simplify the following

(a) $A + AB$

(b) $(A + B + C)\,(A + \overline{B})$

(c) $(A + BC)\,(B + AC) + \overline{A}\, B\, \overline{C}$

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)

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

$A\,A + A\overline{B} + BA + B\overline{B} + CA + C\overline{B}$ — distribution theorem

$= A\,(1 + \overline{B} + B + C) + B\,\overline{B} + C\overline{B}$

$= A.1 + 0 + C\overline{B}$ (since $B\overline{B} = 0$)

$= A + C\overline{B}$

Here also is a solution to Exercise 3.6 (b)

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

$= \overline{AB + \overline{A}\,\overline{B} + C + \overline{C}} = \overline{A\,B + \overline{A}\,\overline{B} + 1}$

(since $C + \overline{C} = 1$)

$= \overline{1} = 0$

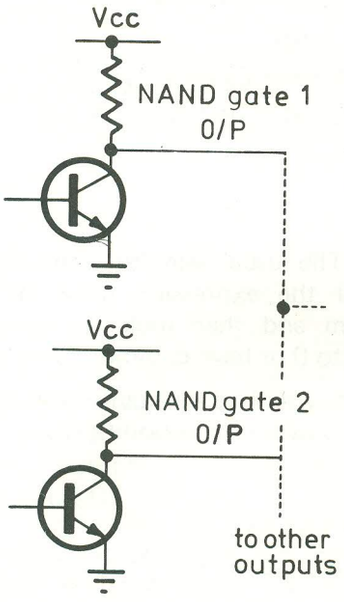
Practical considerations & applications

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.

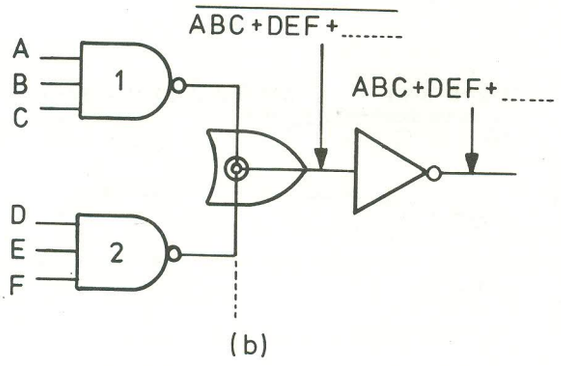
Up to eight inputs are available on some types (e.g 7430) and in other inputs expanders can be connected to increase the number of available inputs.

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 WIRED-OR principle is shown in fig 3.5



(a)



(b)

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.

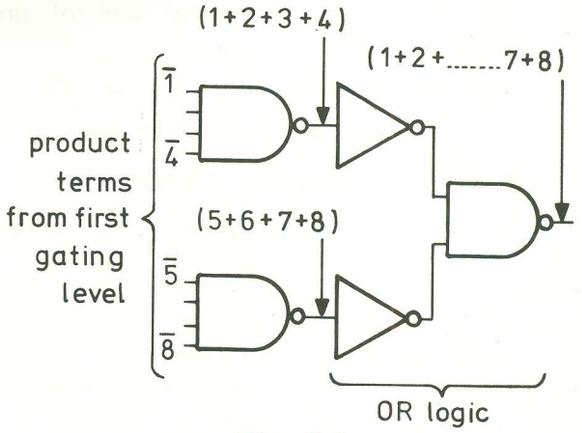


Fig. 3.6

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

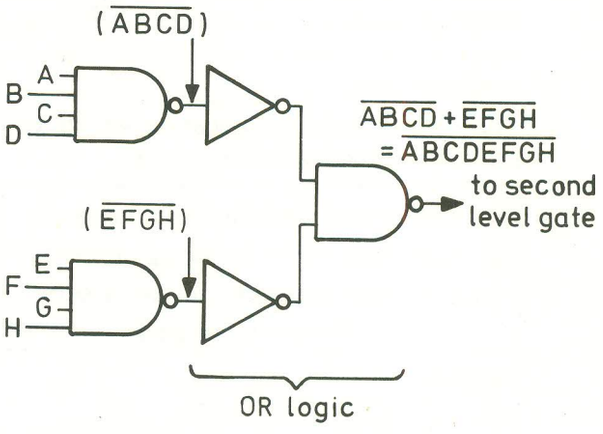


Fig. 3.7

Further Reading

The Simple Latch and Clocked Flip-Flops — Assignment 5

Object

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

Equipment required

Quantity	Equipment
1	Intikit deck CK353 and leads
2	Modules 7400
1	Module 7404
1	Module 7410

Approximate time required

Two and a half hours.

Prerequisites

Assignments 1 and 2 or an understanding of the AND, OR, NAND and NOR operations.

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 occured 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.

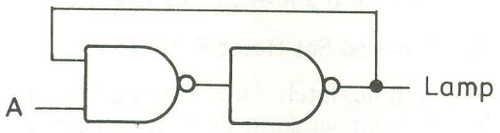


Fig. 5.1

● *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.

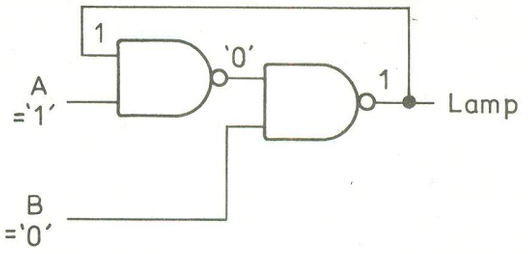


Fig. 5.2

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' an A and a '1' for a momentary '0' an 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.

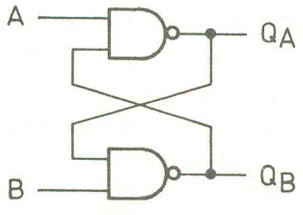


Fig. 5.3

● *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.

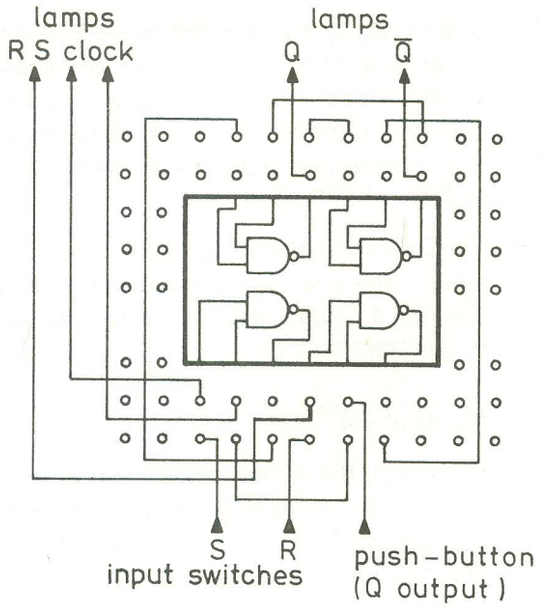
Fig 5.4 shows the resulting circuit, in which the inputs are now called SET (S) and RESET (R) whilst the outputs are labelled Q and $\overline{Q}$ on the assumption that they will always be complementary (that is simultaneous '0' inputs to both sides of the latch will not be allowed).

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?*



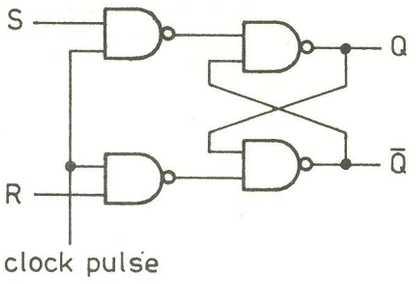


Fig. 5.4

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 $\overline{Q}$ 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.

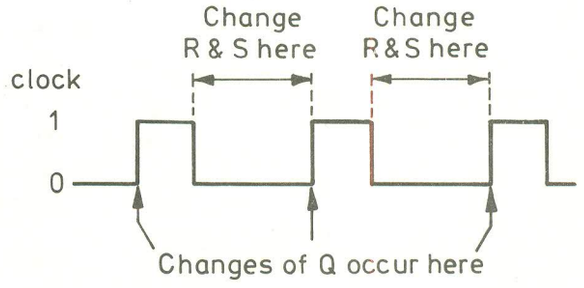


Fig. 5.5

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	?

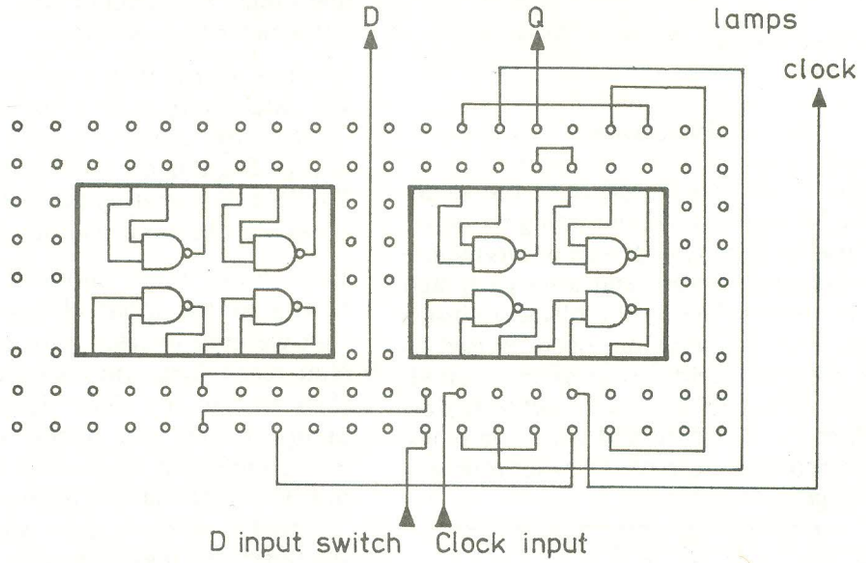
it can be seen that the second and third rows, taken by themselves, describe a circuit whose output Q_{n+1} follows the input S when a clock pulse is applied. It can thus be regarded as a simple, clocked one bit store and is widely used for this purpose. In the two relevant rows S and R are complementary, 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



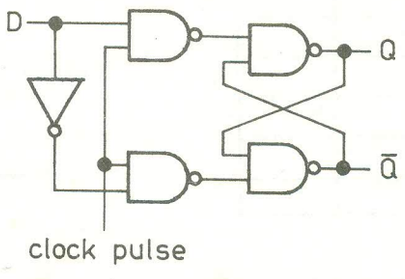


Fig. 5.6

Practical considerations & applications

Although the simple unclocked 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 unclocked 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.

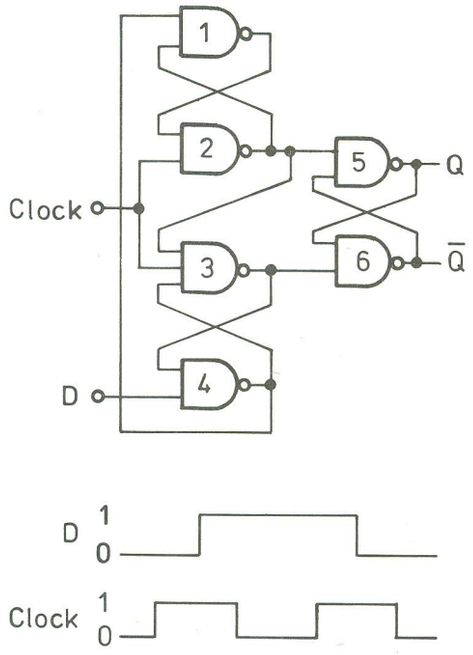


Fig.5.7

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 unclocked latch is to remove the effects of contact bounce from toggle switches and push-buttons as shown in fig 5.8.

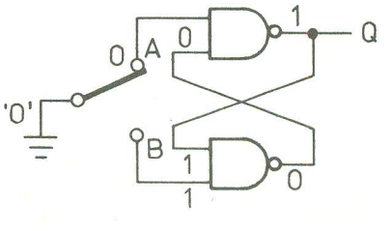


Fig. 5.8 Anti-bounce circuit

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

The J K Flip-Flop

Assignment 6

Object

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

Equipment required

Quantity	Equipment
1	CK353 deck and leads
1	Module 7472
2	Modules 7400
1	Module 7410
1	Module 7404
2	Modules 7420

Approximate time required

Two hours

Prerequisites

Assignment 5 plus a basic knowledge of logic.

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.

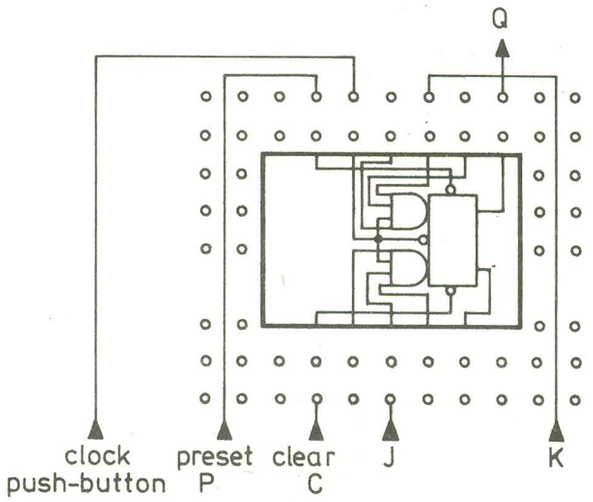


Fig. 6.1

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 (Qn) and after (Qn+1) the clock pulse in each case.

STEP	J	K	Qn	Qn+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 Qn no longer appears.

	J	K	Qn+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?*

● *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.

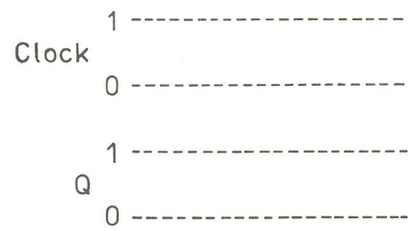


Fig. 6.2

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

a) The clock to the second SR flip-flop is inverted

b) The inputs to the first SR flip-flop are effectively

$S = J.\overline{Q}$

$R = K.Q$

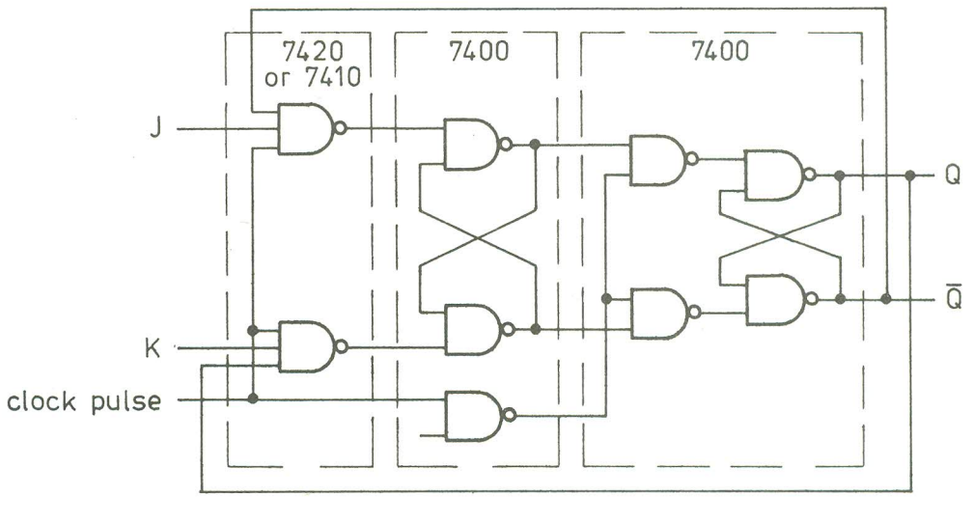


Fig. 6.3 Master-Slave JK

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', the master flip-flop output, for various values of Q, J and K.

Q' of course becomes Qn+1 at the clock trailing edge whilst Q is the same as Qn.

STEP	J	K	Q	S=J.$\bar{Q}$	R=K.Q	Q'
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' 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' = Q.

In rows 3 and 6 exactly the same argument applies.

If we now write Qn instead of Q and Qn+1 instead of Q' 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', 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

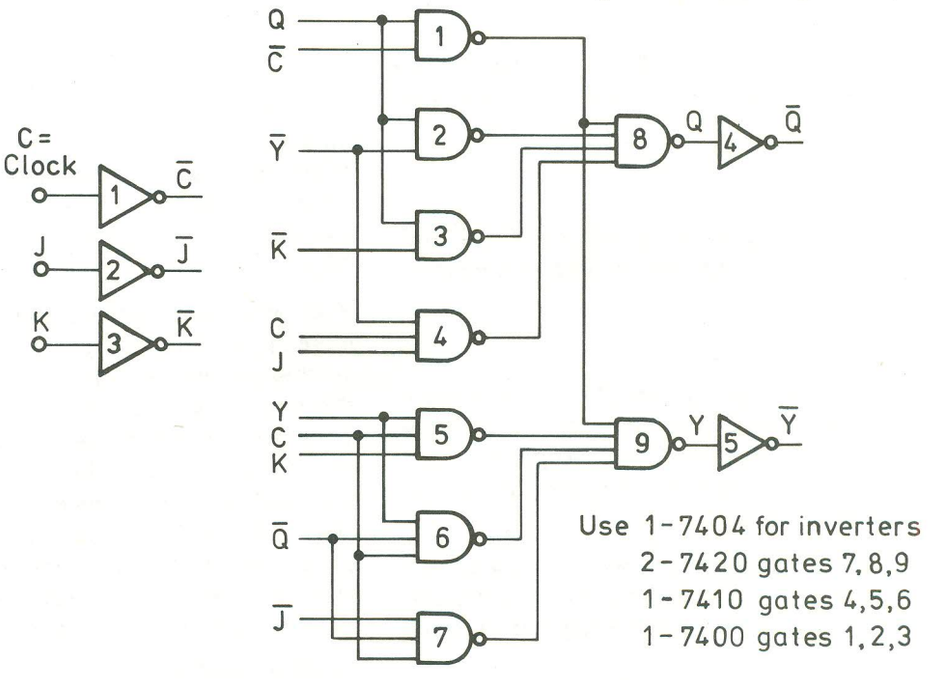


Fig. 6.4 A logical edge-triggered JK

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 $K = \overline{J}$ 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 $K = \overline{J}$. 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

Equivalence, Non-equivalence and other circuits — Assignment 7

Object

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

Equipment required

Quantity	Equipment
1	INTIKIT deck CK353 and leads
3	Modules 7400
1	Module 7404
1	Module 7410
3	Modules 7420

Approximate time required

Three hours.

Prerequisites

Assignments 1 to 4 or a knowledge of the basic logic functions and Karnaugh mapping techniques.

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 $A\overline{B}$ and 13 is its complement $\overline{A} + B$

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

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

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

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

$Z_{13} = \overline{A} + B$, by a similar argument, is 'A less than or equal to B' or $A \leqslant B$.

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

$Z_{11} = A + \overline{B}$ is 'A greater than or equal to B' or $A \geqslant B$.

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

$Z_9 = \overline{A}\overline{B} + AB$ 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.

$A > B$, $A \leqslant B$, $A < B$, and $A \geqslant B$

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

Practical 7.1

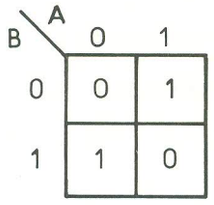
Design, construct and verify the truth tables of logic circuits for each of the above functions.

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

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 $\not\equiv$. e.g A $\oplus$ B is the same as A $\not\equiv$ B.

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



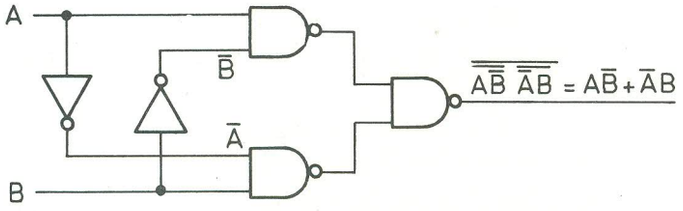


Fig. 7.1

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.

$Z_6 = \overline{A}B + A\overline{B}$

$= (\overline{A} + A)(\overline{A} + \overline{B})(B + A)(B + \overline{B})$ distribution

$= (A + B)(\overline{A} + \overline{B})$ - rearrangement

$= (A + B)\ \overline{AB}$ i.e (A OR B) AND NOT (A AND B) *

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

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

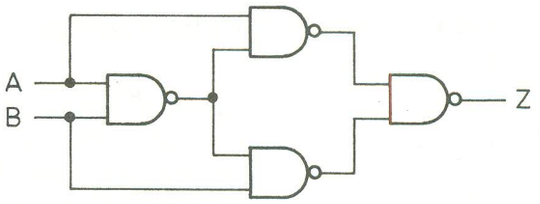


Fig. 7.2

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.

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

Obviously this equivalence function can be generated simply be 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.

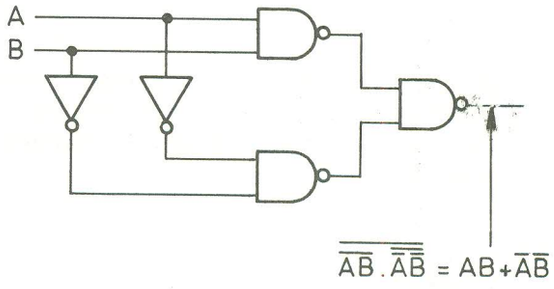


Fig. 7.3

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

$A > B$ function $A\bar{B}$
$A < B$ 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?*

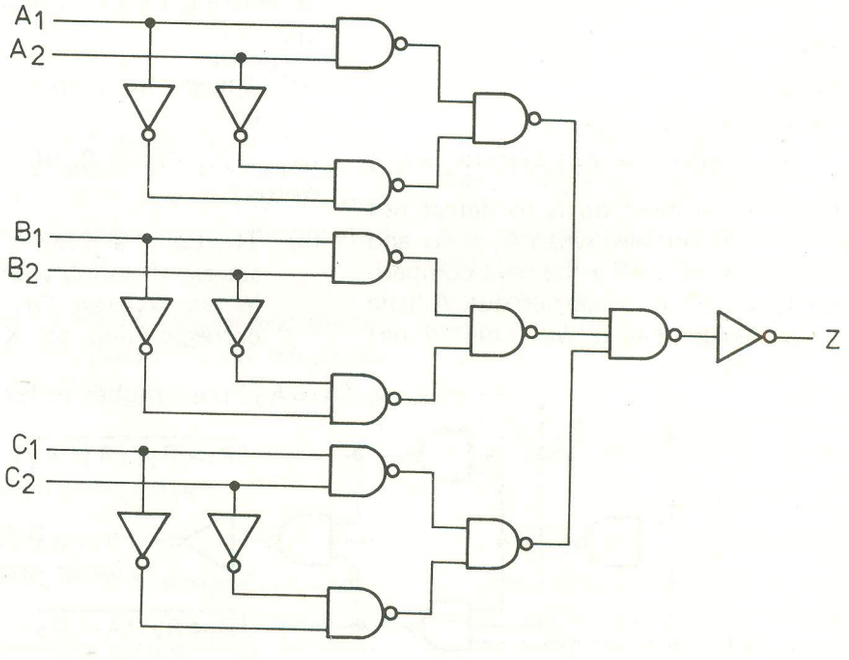


Fig. 7.4 Equality detector

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.

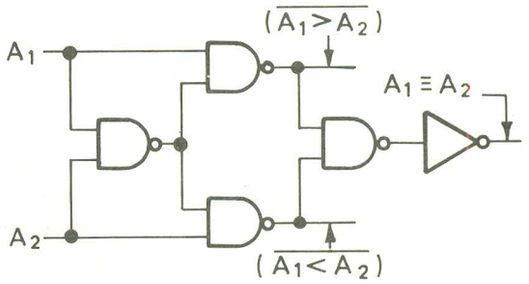


Fig. 7.5

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 $(A_1 \equiv A_2)$ 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

$(A_1 \equiv A_2)(B_1 \equiv B_2)(C_1 \equiv C_2)$ 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 $X < Y$.

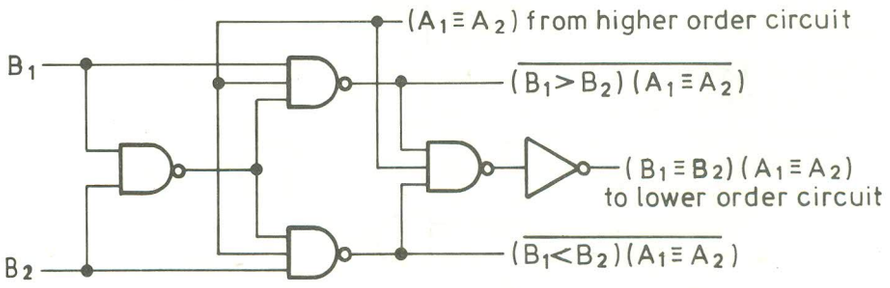


Fig. 7.6

The full circuit is shown below in fig 7.7.

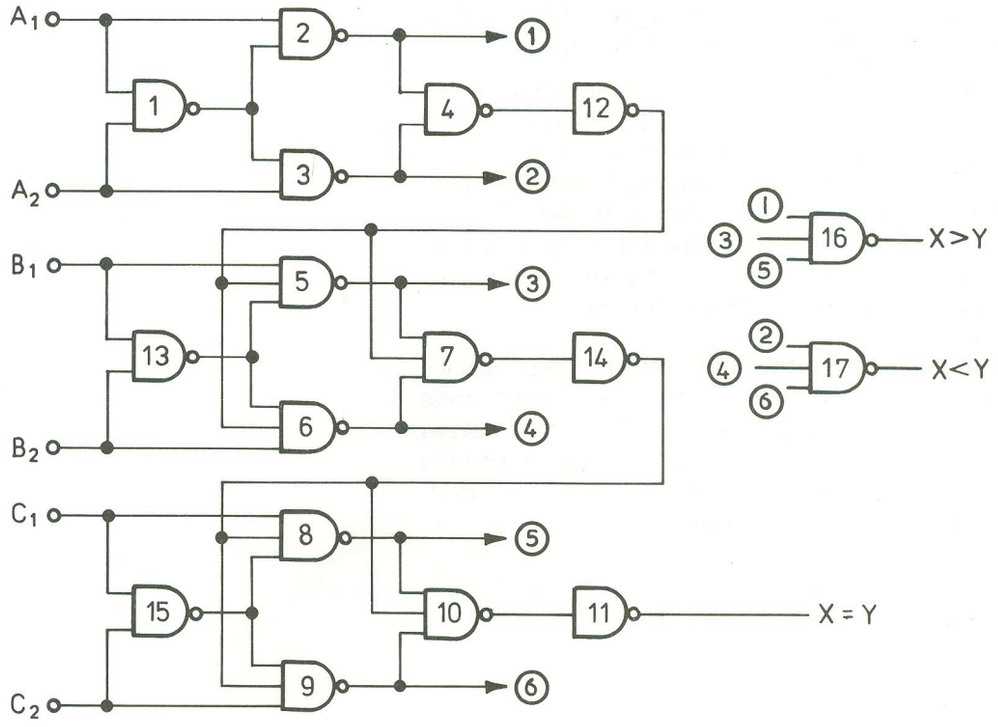


Fig. 7.7 3-bit numerical comparator

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-equivalance circuit for two inputs A and B, then as we have just seen, the output is

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

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

if A = 0 then $Z = 0.\overline{B} + 1.B = B$

if A = 1 then $Z = 1.\overline{B} + 0.B = \overline{B}$

In other words we can consider A as a control signal which makes the output equal either B (true) or $\overline{B}$ (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

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

$$\text{SUM} = \Sigma \quad \text{(sigma)} = A\overline{B} + \overline{A}B$$

$$\text{CARRY} = C_o = AB$$

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.

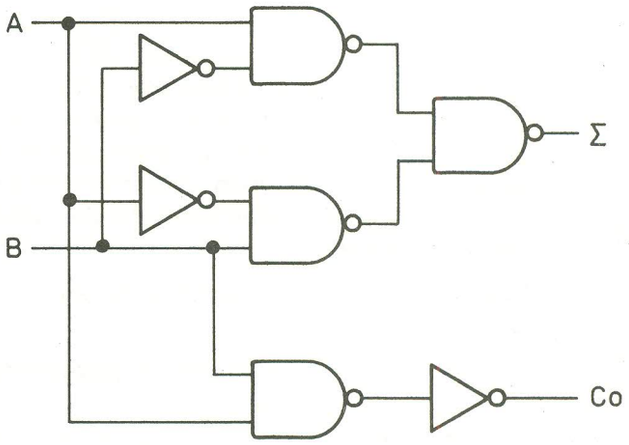


Fig. 8.1 Simple adder

Practical 8.1

Construct the logic of fig 8.1 using one 7400 and one 7404. Connect lamps to the Σ and C_o 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 $C_o = AB$ appears in complement form and this leads to a slightlv 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_o was the carry out to the next higher digit position. In this 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.

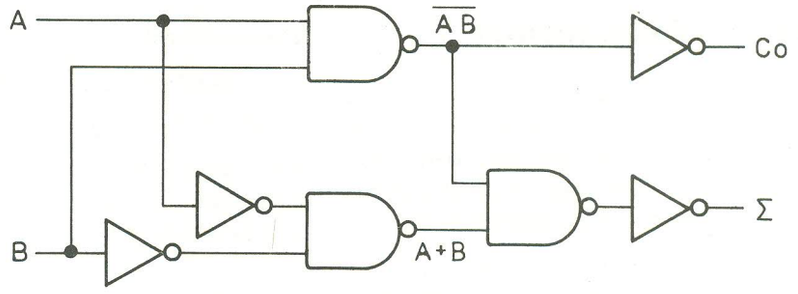


Fig.8.2 Alternative adder

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)

$(A + B)$ = Σ_1 with a possible carry out C_o

and $(\Sigma_1 + C)$ = Σ_2 with a possible carry out C_o

Notice that a carry out can only occur in one of the stages because if A and B are both '1', Σ_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

(a) The Σ output of the first half is used as an input to the second half.

(b) The two carry outputs C' are combined in an OR gate to produce C_o.

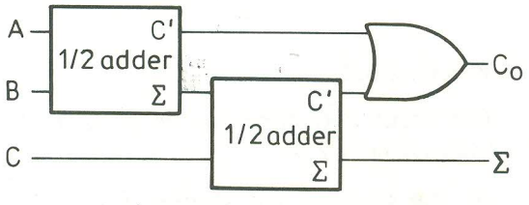


Fig. 8.3 A full adder

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_o 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_o for all eight possible combinations of A, B and C.

A	B	C	Σ	C_o
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

(a) The sum output is '1' when any — or all — of the inputs are at '1'

(b) 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_o from your truth table they should lead you to the following sum of products expressions.

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

$$C_o = AB + AC + BC$$

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.

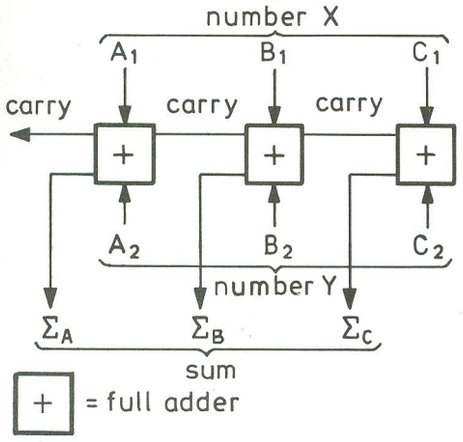


Fig. 8.4

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).

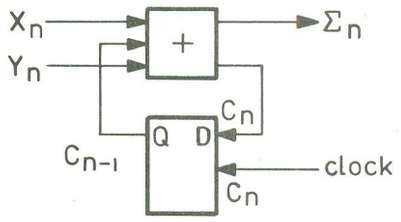


Fig. 8.5 A serial adder

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}.

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 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:-

(a) Set X_n and Y_n digits
(b) Note sum output Σ_n
(c) Operate clock button
(d) Set X_{n+1} and Y_{n+1}
(e) 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, $\overline{\Sigma}$, C_O, $\overline{C}_O$).

Further Reading

Synchronous Counters

Assignment 10

Object

To define the terms asynchronous and synchronous in relation to counting circuits and to study the design and construction of synchronous counters.

Equipment required

Quantity	Equipment
1	INTIKIT deck CK353 and leads
1	Module 7400
4	Modules 7472

Approximate time required

Two and a half hours.

Prerequisites

Assignments 3, 4, 5 and 6 or a knowledge of basic logic, Karnaugh maps, and D and JK flip-flops.

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.

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 $\overline{Q}$)
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				

● *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', $\overline{Q}$ 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?*

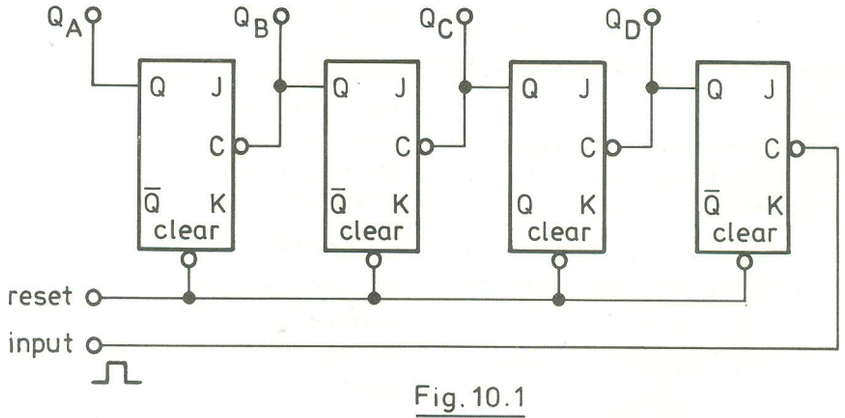


Fig. 10.1

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

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.

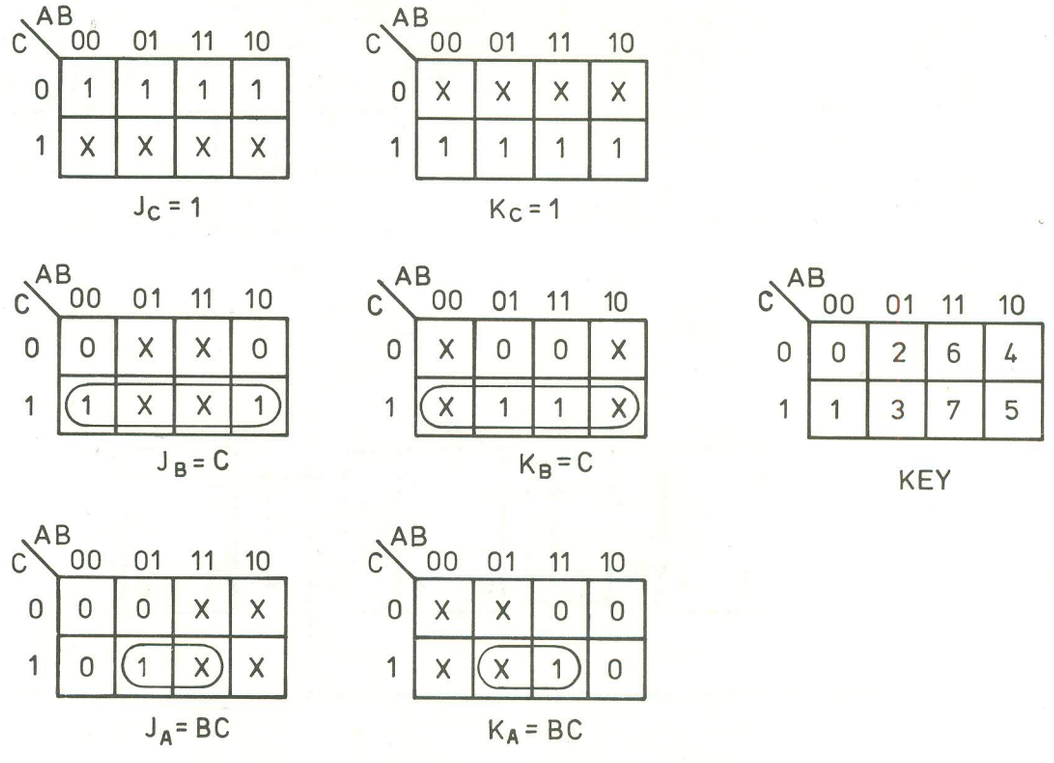


Fig.10.2 Maps for binary counter

(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

A B C = 1 1 0 and 1 1 1

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.

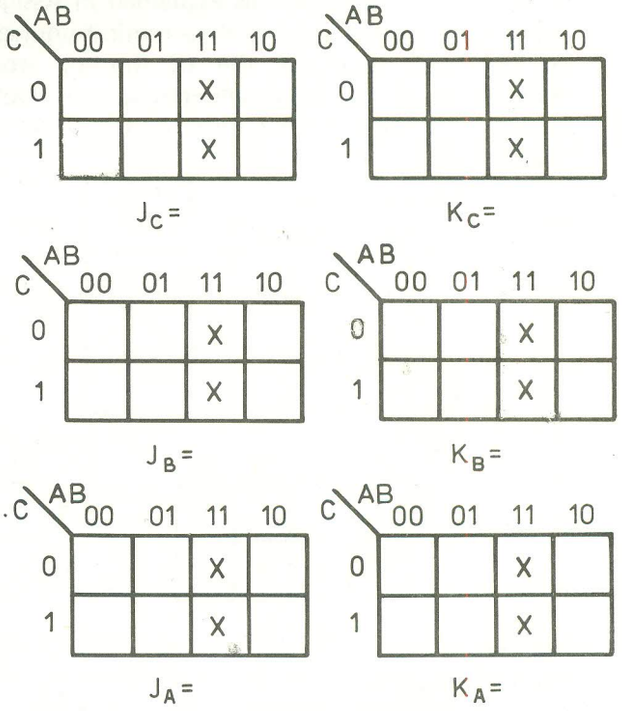


Fig. 10.4 Maps for modulo - 6 counter

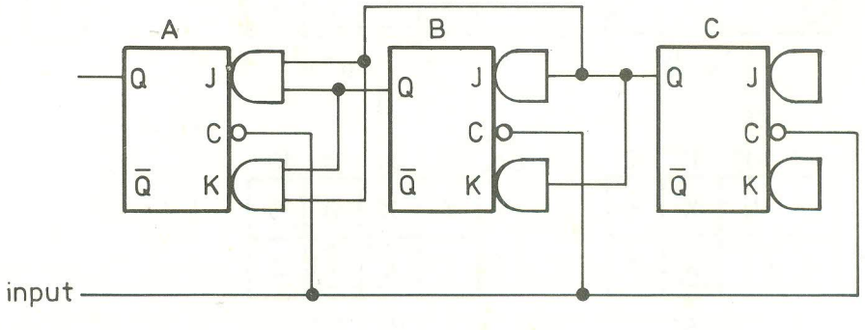


Fig. 10.3 Synchronous binary counter

You should have obtained the following expressions.

$J_C = K_C = 1$

$J_B = \bar{A}C \qquad K_B = C$

$J_A = BC \qquad K_A = C$

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

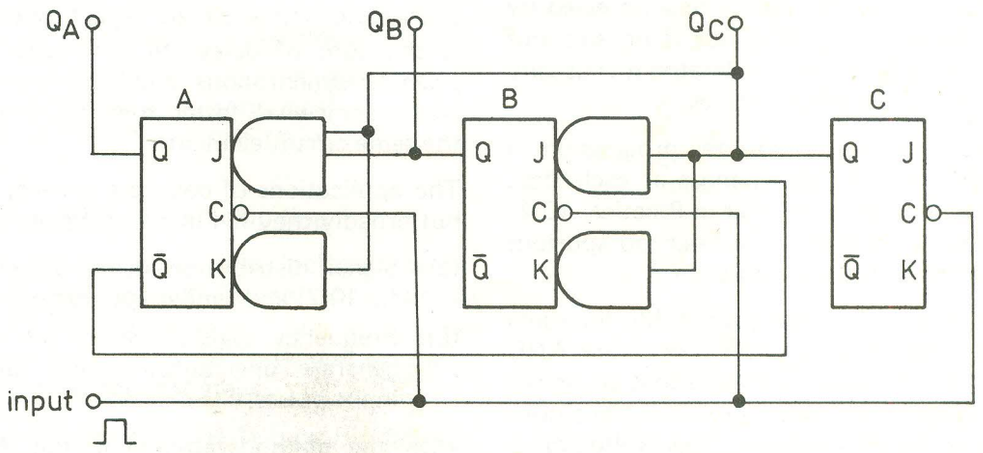


Fig. 10.5 Modulo 6 counter

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:-

(a) 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.

(b) 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

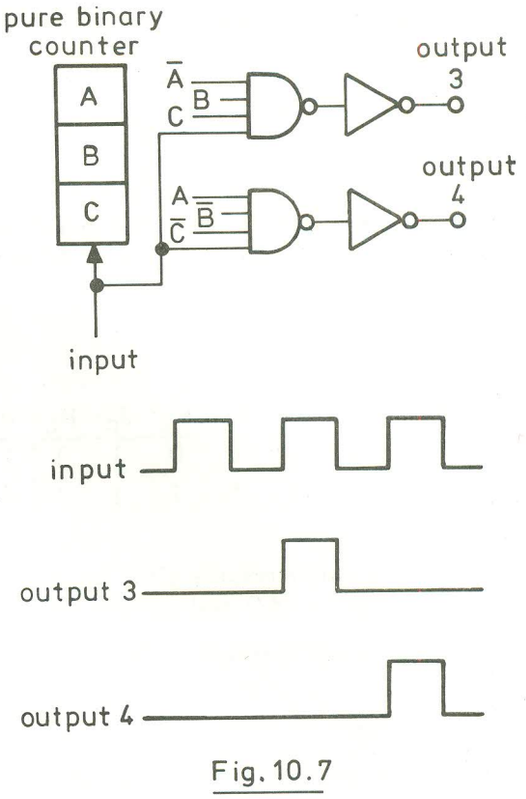


Fig. 10.7

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.

(a) Signal distribution using decoders (as in fig 10.7) and similar applications.

(b) 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:-

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

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.

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

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 x 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

Asynchronous Counters

Assignment 11

Object

To study design methods of asynchronous *counters and to construct various examples.*

Equipment required

Quantity	Equipment
1	CK353 INTIKIT deck and leads
1	Module 7400
4	Modules 7472

Approximate time required

Three hours.

Prerequisites

Assignment 10

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'.

	Present state			Next state			Flip-flops
Count	A	B	C	A	B	C	to toggle
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	?	?	?	?

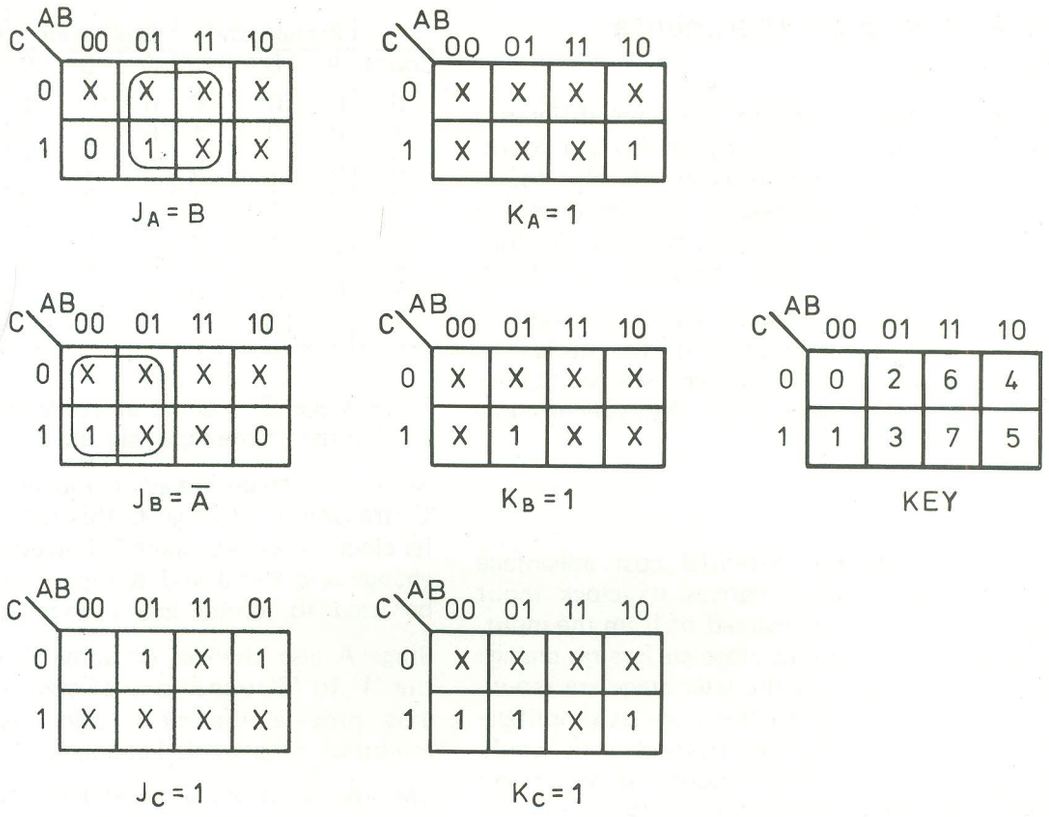


Fig. 11.1

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.

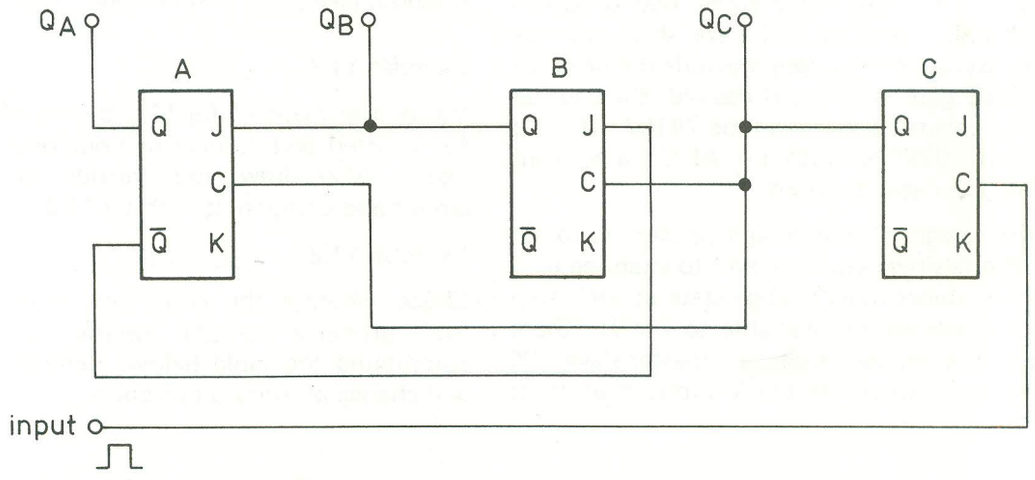


Fig. 11.2 Asynchronous modulo-6 counter

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.

	Present state				Next state				F/F to
Dec	A	B	C	D	A	B	C	D	toggle
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.

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 → 0 and if J_C, K_C are such that this will in turn cause C = 1 → 0, then J_B, K_B must be studied to see if the result is a change in B, and so forth.

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

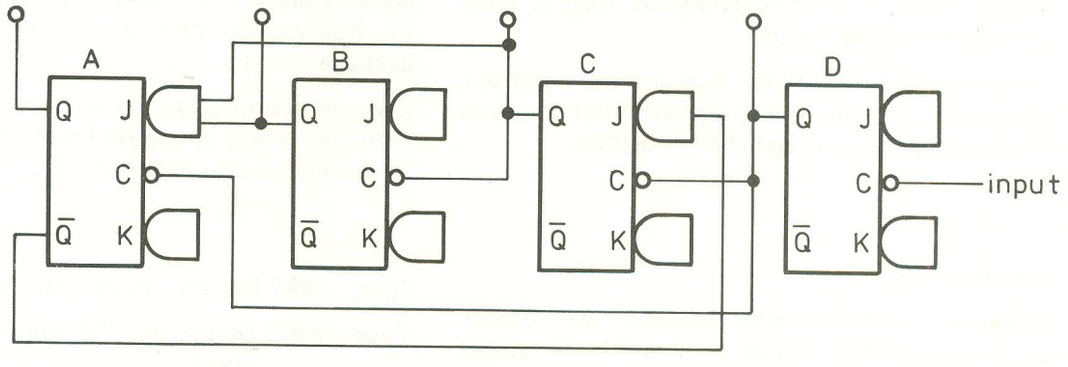


Fig. 11.4 Asynchronous BCD counter using type 7472

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

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

(b) Arbitrary

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 s achieved in this type by AND gating the clock signal with the outputs of all previous stages.

Further reading