

3. Digital Electronics

3.1 Introduction

In the previous section we discussed operational amplifiers which are *analogue* devices that respond to a continuous range of voltages. Here we begin to examine *digital* circuits, which in contrast, respond to *two discrete voltage levels* known as either ‘high and low’ or ‘true and false’ or ‘1 and 0’. Essentially, the two voltage levels are identified by either being above some threshold (‘high’, ‘true’ or ‘1’) or below some threshold (‘low’, ‘false’ or ‘0’). For example, in TTL (transistor-transistor logic) circuits, ‘high’ corresponds to $> 2V$, whilst ‘low’ corresponds to $< 0.8V$.

Digital circuits now form part of everyday life for all of us: in computers, cameras, phones and televisions, the internet and fibre-optic telecommunications. The list is seemingly endless. Given that digital data is just a series of 1’s and 0’s, the great advantage over analogue systems is that it can be stored, transmitted and reproduced relatively immune from noise, which can sometimes become a problem in analogue systems.

In this section we will study the basic building blocks of digital electronic circuits – namely *logic gates*. We will see how logic gate circuits can be used to perform digital operations both as *combinational logic* (where the logic gates are ‘combined’ to produce a circuit that depends only on the input at a particular time), and as *sequential logic* (where the output will depend upon the state of the system at an earlier time). The latter system is used in *memory* circuits.

We will also study how *truth tables* become a useful tool when designing a circuit for a particular application. Moreover, we will see how truth tables can be used in combination with *Boolean Algebra* (a special kind of mathematics which allows us to manipulate logic expressions) and *Karnaugh maps* to simplify and/or optimise digital circuits. Finally, we will take a brief look at how logic gates are made, and the pros and cons of the various logic gate designs.

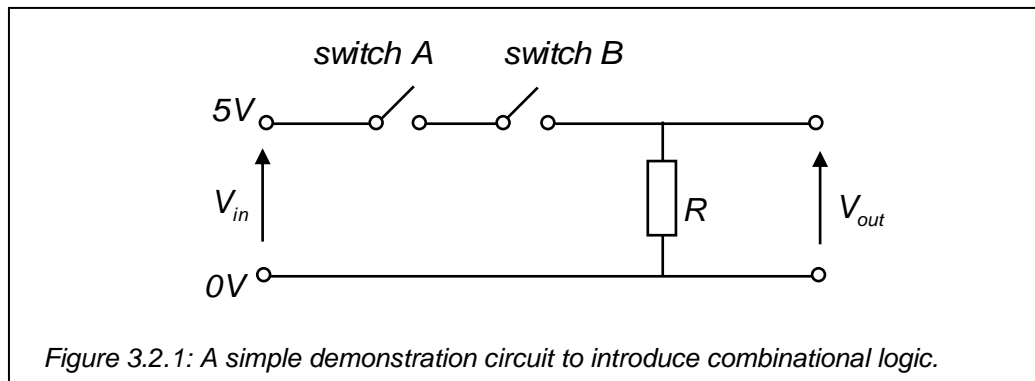
3.2 Combinational logic circuits

As mentioned above, combinational logic is used when the output of a circuit is to be a function of the current input state only. The following section begins by illustrating this with a simple example.

3.2.1 Introduction to logic gates, truth tables and Boolean expressions

We begin by studying the circuit shown in Figure 3.2.1. We could consider this circuit to be for example, a simple safety system in a particular vehicle where a 5V output would allow the engine to be started (perhaps via a relay circuit). Therefore, for the engine to be energised, both switch A and switch B need to be closed. The two closed switches could for example, correspond to the driver’s door being closed and

the driver's safety belt being in position. Only when the door is closed and the safety belt is worn can the vehicle be driven.



Clearly this is a very simplistic example but demonstrates very well the notion of a logic circuit. Let an open switch (A or B) in the circuit be represented by the binary digit 0, and a closed switch be represented by the binary digit 1. When the output of the circuit is at +5V, the output function F , can be represented by the binary digit 1, and when the output is at 0V, then the output function F can be represented by the binary digit 0. Given this information, we can first of all construct a truth table, which is normally the starting point in transforming a circuit design into a logic circuit. (A truth table can also be the starting point in the process of simplifying existing logic circuits). The truth table will display the output for all of the possible combinations of input.

The truth table for the circuit shown in Figure 3.2.1 is given in Figure 3.2.2. As can be seen, there are 4 possible input combinations corresponding to the 2 input variables (A and B). Later in these notes we will see that 3 input variables (A, B & C) will correspond to 8 possible input combinations and 4 input variables (A, B, C & D) will correspond to 16 possible input combinations. The number of input combinations correspond to the full complement of binary numbers that exist within the amount of 'columns' available. In fact, the truth table can be filled out by writing down all binary numbers from zero onwards within the columns available. This is no coincidence as we shall see. Binary data transfer through digital systems is limited (amongst other factors) by the available inputs and outputs of a system, which in turn determine the number of binary digits (or *bits*) that can be used.

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

Figure 3.2.2: The truth table for the circuit shown in Figure 3.2.1.

For the truth table shown in Figure 3.2.2, it can be seen that the output function F is only at a logic 1 level, when both switches A *AND* B are closed. George Boole, a

nineteenth century mathematician developed a system of algebra to deal with these two state (0,1) devices. In Boolean Algebra, the above example is written as

$$F = A.B \quad (3.1)$$

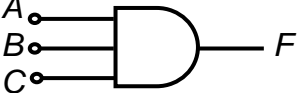
where the right-hand-side of the equation is pronounced “A and B”. The function requires a single logic gate (an AND gate) to allow such a circuit to work. In fact, our design of the vehicle safety circuit shown in Figure 3.2.1, followed by the truth table and subsequent identification of the Boolean expression has enabled us to complete the task of the digital circuit design – we will require a single ‘AND’ logic gate to be connected between the two inputs, and the output.

Before developing further our understanding of logic circuits, and how we can use truth tables and Boolean algebra to help design these circuits, the individual logic gate circuit symbols, truth tables and corresponding Boolean expressions are shown below, beginning with a 3-input AND gate. There are four basic types of logic gate, each having a corresponding inverse gate with reversed output. The four inverse gates are shown after the basic gates have been introduced.

Immediately following this section are some example questions which should help to make clear how the gates function.

The AND gate

The AND gate



The output of an AND logic gate can only be at a logic 1 when *all* of the inputs are at logic 1. In Boolean form:

$$F = A.B.C$$

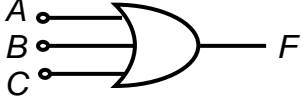
The AND truth table

INPUTS			OUTPUT
<i>C</i>	<i>B</i>	<i>A</i>	<i>F = A.B.C</i>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Note: $A.B.C$ is pronounced “A and B and C”.

The OR gate

The OR gate



The output of an OR gate will be at a logic 1 when *any* of the inputs are at a logic 1. In Boolean form:

$$F = A+B+C$$

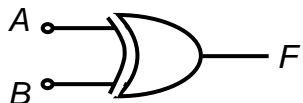
The OR truth table

INPUTS			OUTPUT
C	B	A	$F = A+B+C$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note: $A+B+C$ is pronounced “A or B or C”.

The XOR gate

The XOR gate



The output of an XOR gate can be at logic 1 when *only one* of the inputs is at logic 1. In Boolean form:

$$F = A \oplus B$$

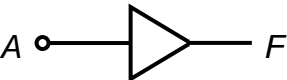
The XOR truth table

INPUTS		OUTPUT
B	A	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Note: $A \oplus B$ is pronounced “A ex-or B”.

The buffer gate

The buffer gate



The output of the buffer is identical to the input. In Boolean form:

$$F = A$$

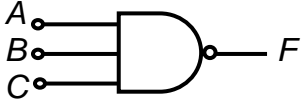
The buffer truth table

INPUT A	OUTPUT F = A
0	0
1	1

Note: The buffer gate is used to boost signals in a circuit lost through resistive elements. We don't want the 'digital 1' voltage to fall too close to the 'high' threshold.

The NAND gate

The NAND gate



The output of the NAND logic gate is the reverse of the output of the AND gate. In Boolean form:

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

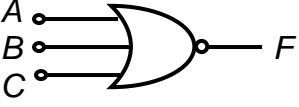
The NAND truth table

INPUTS			OUTPUT
C	B	A	F = $\overline{A.B.C}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Note: The output of the NAND gate is the reverse of the AND gate. The line above the right hand side of the Boolean expression for the NAND gate basically reverses what it would have been without the line. (This is a general rule. Whenever the line is present, it reverses the expression). The line above the expression in the NAND gate example has the same function as the tiny circle at the output of the circuit symbol. Without the tiny circle, the symbol would be an AND gate.

The NOR gate

The NOR gate



The output of the NOR gate is the reverse of the output of the OR gate. In Boolean form:

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


The NOR truth table

INPUTS			OUTPUT
C	B	A	$F = \overline{A + B + C}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Similarly, the NOR gate's output is the inverse of the OR gate. Like the NAND gate it has the line above the right-hand-side of the Boolean expression and the tiny circle on the output of the circuit symbol.

The XNOR gate

The XNOR gate



The output of the XNOR gate is the reverse of the output of the XOR gate. In Boolean form:

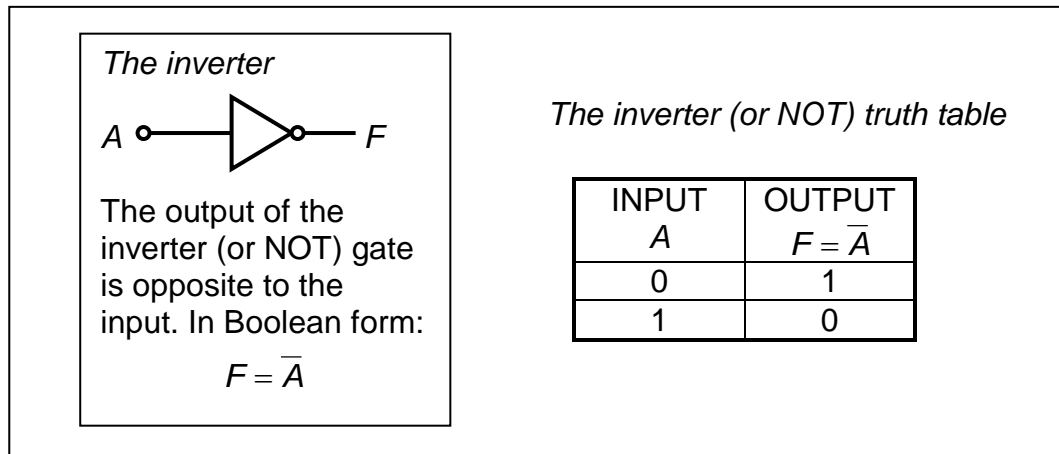
$$F = \overline{A \oplus B}$$

The XNOR truth table

INPUTS		OUTPUT
B	A	$F = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

Again, this gate is the inverse of the XOR gate.

The inverter gate



... and the inverter is the inverse of the buffer.

3.2.2 Combinational logic worked examples – Part A

Boolean expressions can be manipulated to become either sum of products expressions or product of sums expressions which are sometimes easier to deal with algebraically than other forms of expression. The expressions take the following structure:

sum of products: $F = \bar{A}.B.C + A.\bar{B}.C + A.B.\bar{C}$

product of sums: $F = (\bar{A} + B + C).(A + \bar{B} + C).(A + B + \bar{C})$

In Example 2 below, we will see how a 'sum of products' expression can be taken directly from a truth table.

Example 1

In a chemical process, an alarm sounds ($F = 1$) if the temperature rises above a specified level (A), or the pressure rises above a specified level (B), or the supply of raw materials is below a specified minimum (C). High temperature, high pressure and sufficient materials are all represented by logic 1.

Write down the Boolean expression and complete the truth table for the required output F .

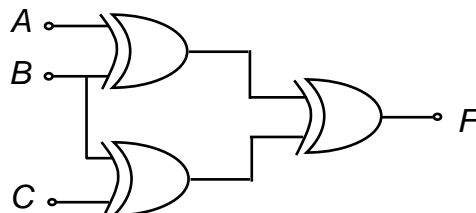
Example 1 (solution)

The alarm will sound if $A = 1$, or $B = 1$ or $C = 0$. The Boolean expression is therefore $F = A + B + \bar{C}$. The truth table, which has 8 possible combinations of the 3 inputs is written as

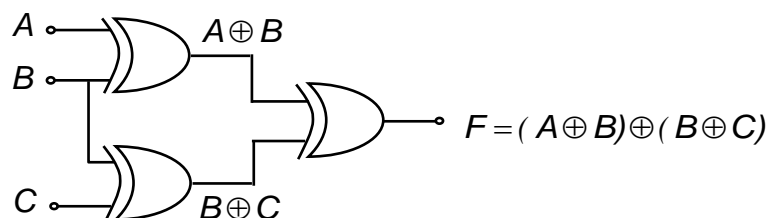
INPUTS			OUTPUT
C	B	A	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Example 2

For the three-input logic circuit shown below obtain the Boolean expression for the output. Also, complete a truth table for the circuit and from this derive an alternative 'sum of products' expression.

Example 2 (solution)

The good way to obtain an error-free Boolean output expression is to write it on the Figure, noting the output of all logic gates i.e.



therefore

$$F = (A \oplus B) \oplus (B \oplus C) \quad (3.2)$$

A truth table may be completed by including some extra columns if necessary i.e.

INPUTS			$A \oplus B$	$B \oplus C$	$F = (A \oplus B) \oplus (B \oplus C)$
C	B	A			
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	1	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	0	0	0

From the truth table we can derive an alternative 'sum of products' expression which is typically an easier expression to manipulate algebraically. We achieve this by writing down the input in each case which results in F being equal to 1. For example, $F = 1$ when $A = 1$ and $B = 0$ and $C = 0$.

Therefore we could say that

$$F = 1 \text{ when } A = 1 \text{ and } B = 0 \text{ and } C = 0$$

OR

$$F = 1 \text{ when } A = 1 \text{ and } B = 1 \text{ and } C = 0$$

OR

$$F = 1 \text{ when } A = 0 \text{ and } B = 0 \text{ and } C = 1$$

OR

$$F = 1 \text{ when } A = 0 \text{ and } B = 1 \text{ and } C = 1$$

which leads us to the 'sum of products' expression - equivalent to the expression shown in Equation (3.2) i.e.

$$F = A.\bar{B}.\bar{C} + A.B.\bar{C} + \bar{A}.\bar{B}.C + \bar{A}.B.C \quad (3.3)$$

We will return to Equation (3.3) in Part B of the worked examples.

3.2.3 Rules of Boolean algebra

As you may have expected, before we begin to manipulate Boolean expressions with Boolean algebra, there are some rules that we must follow. The most useful of these are listed below in Table 3.2.3. All of the rules are identities that can be proven with truth tables. In particular it is worth noting and remembering the identities which constitute DeMorgan's Theorems, i.e. identities (15) and (16), as these tend to be rather useful.

(1) $A + \bar{A} = 1$	(2) $A + A = A$	(3) $A + 0 = A$	(4) $A + 1 = 1$
(5) $A \cdot \bar{A} = 0$	(6) $A \cdot A = A$	(7) $A \cdot 0 = 0$	(8) $A \cdot 1 = A$
(9) $A = \bar{\bar{A}}$	(10) $\bar{A} + (A \cdot B) = \bar{A} + B$	(11) $A + (A \cdot B) = A \cdot (1 + B) = A$	
(12) $A \cdot (A + B) = A$	(13) $A + (\bar{A} \cdot B) = A + B$	(14) $(A \cdot B) + (\bar{B} \cdot C) + (A \cdot C) = (A \cdot B) + (\bar{B} \cdot C)$	
(15) $\overline{A + B} = \bar{A} \cdot \bar{B}$	(16) $\overline{A \cdot B} = \bar{A} + \bar{B}$	(17) $(A \cdot \bar{B}) + (\bar{A} \cdot B) = A \oplus B = (A + B) \cdot \bar{A} \cdot \bar{B}$	
Table 3.2.3: Some of the more frequently-used rules of Boolean algebra		(18) $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	

3.2.4 Combinational logic worked examples - Part B

Example 3

Simplify the 'sum of products' expression given in Equation (3.3) above with Boolean algebra to show that the output F does not depend upon the input B .

Example 3 (solution)

Starting with Equation (3.3) and by utilising the identities in Table 3.2.3 we have

$$F = A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C \quad (3.3)$$

Now, Boolean expressions can be 'factored' just like normal algebra in the way shown in identity (18) to give

$$F = A \cdot \bar{C} \cdot (B + \bar{B}) + \bar{A} \cdot C \cdot (B + \bar{B}) \quad (3.4)$$

but from identity (1), $(B + \bar{B}) = 1$ and therefore

$$F = A \cdot \bar{C} \cdot (1) + \bar{A} \cdot C \cdot (1) \quad (3.5)$$

and from identity (8) we get

$$F = A.\overline{C}.(1) + \overline{A}.C.(1) = A.\overline{C} + \overline{A}.C \quad (3.6)$$

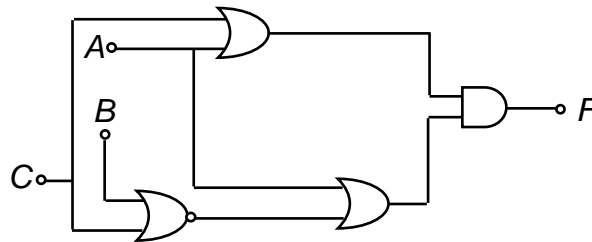
and finally from identity (17) we get

$$F = A \oplus C \quad (3.7)$$

which confirms that the output F does not depend upon the input B .

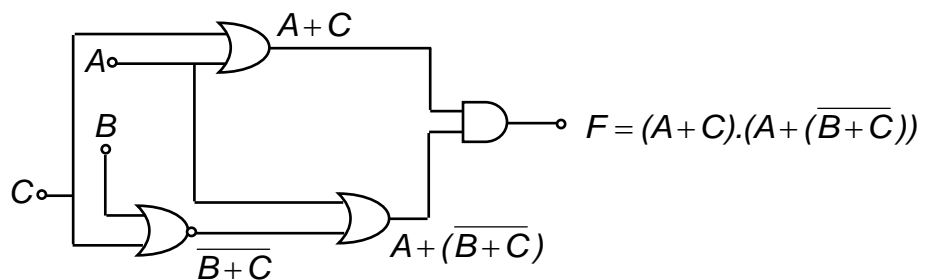
Example 4

For the 3-input logic circuit given below, obtain the Boolean expression for its output and show that the expression is equivalent to $F = A$ with Boolean algebra.



Example 4 (solution)

From the circuit, the Boolean expression is found as follows:



Therefore the output expression is

$$F = (A + C).(A + (\overline{B + C})) \quad (3.8)$$

Expanding the brackets – identity (18) gives

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

Identity (6) and factoring gives

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

Identities (4) and (8) and some more factoring gives

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

Identities (4) and (8) again give

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

Identity (15) – DeMorgan's theorem gives

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

Finally identity (5) leads us to

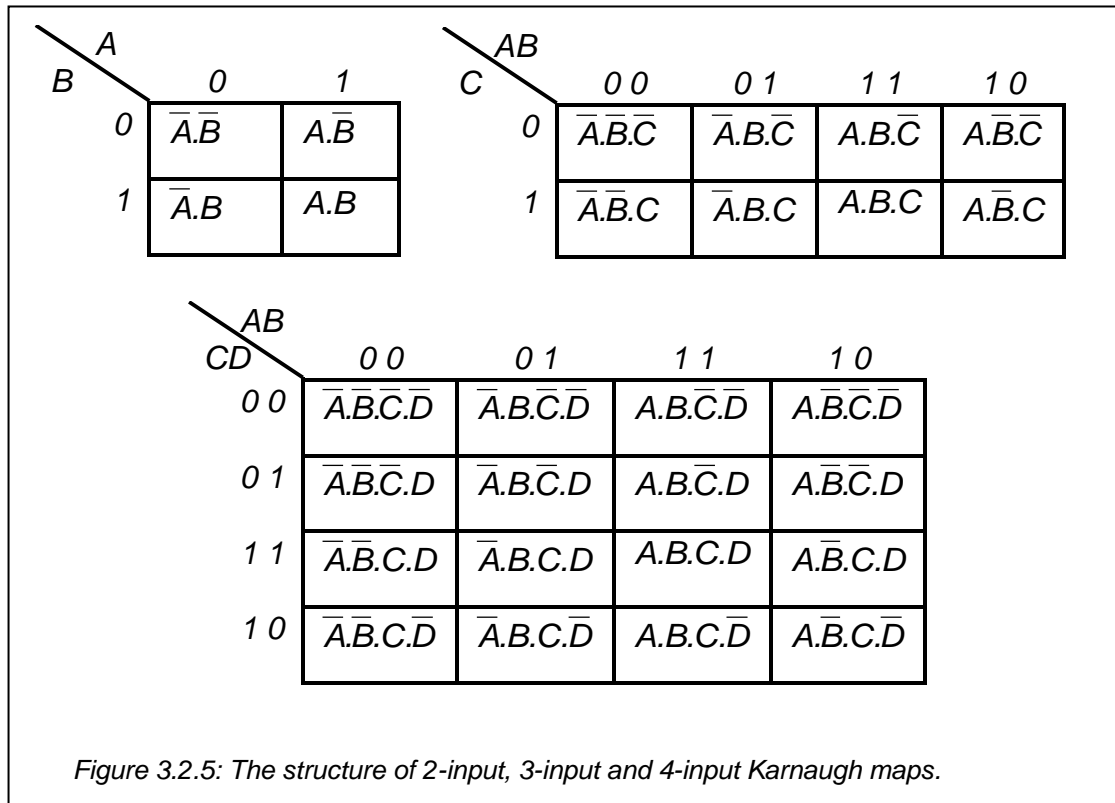
$$F = A \quad (3.9)$$

3.2.5 Karnaugh maps

Karnaugh maps provide an alternative method of arriving at the *simplest* possible logic expression – enabling the simplest of circuits to be constructed. In practical terms, the maps can be used with up to five input variables - above this number will result in the maps becoming too large to manage sensibly. A maximum of four input variables (A, B, C, D) will be considered during this module.

To implement a Karnaugh map, the Boolean expression is presented in its 'sum of products' form.

Figure 3.2.5 shows the structure of 2-input, 3-input and 4-input Karnaugh maps. As can be seen, Karnaugh maps show all of the possible combinations of the input variables, and are arranged so that only a single change of variable occurs between any pair of adjacent cells. The axes of the 3-input and 4-input Karnaugh maps follow the order 00, 01, 11, 10, for this reason. Adjacent cells include joining together the cells on the right of the map with the cells on the left of the map, and the cells at the top with the cells at the bottom, i.e. the map forms the surface of a toroid. Another way of thinking about this is if you go off the edge of the map on the right, you immediately appear at the cell on the left of the map. If you step off the bottom of the map, you immediately step onto the cell at the top.



In order to complete the map, a number '1' should be written in the appropriate cell if the term exists in the sum of products expression, and '0' should be written if the term does not exist in the sum of products expression. This is equivalent to reading directly from a truth table where there is a '1' at the output and transferring the input in each case to the map – *a similar process to forming the sum of products expression itself from the truth table!* Indeed, with practice, the sum of products expression can be ignored completely when the starting point is a truth table.

Once the sum of products expression has been transferred to the Karnaugh map, then we group together adjacent '1's. Adjacent cells (in 1 or 2 dimensions) totalling a number 2^n where n is an integer (0, 1, 2 ...etc) are of use only i.e. 1, 2, 4, 8 etc. We should aim to group together the *most* amount of '1's into the *least* amount of groups. *Diagonal groups are of no interest.*

The Karnaugh mapping process is identical to the process of factoring a sum of products Boolean expression.

The best way to consolidate understanding of Karnaugh maps is by studying some example problems. These follow in the next section.

3.2.6 Combinational logic worked examples - Part C

Example 5

Simplify the Boolean expression

$$F = A.B.C + \bar{A}\bar{B}\bar{C} + A.B.\bar{C} + \bar{A}\bar{C} \quad (3.10)$$

with a Karnaugh map.

Example 5 (solution)

This is a 3-input system so requires the 3-input Karnaugh map. Moreover, there is one term $\bar{A}\bar{C}$ in Equation (3.10) that needs expanding. From identities (8), (1) and (18), in Table 3.2.3, $\bar{A}\bar{C} = \bar{A}\bar{C}.(B + \bar{B}) = \bar{A}.B.\bar{C} + \bar{A}.\bar{B}.\bar{C}$. Therefore our expression now becomes

$$F = A.B.C + \bar{A}\bar{B}\bar{C} + A.B.\bar{C} + \bar{A}.B.\bar{C} + \bar{A}.\bar{B}.\bar{C} \quad (3.11)$$

However, now one of the expressions is repeated. This doesn't matter – we just ignore one of them. Indeed we could have put the expression (Equation (3.10)) straight into the Karnaugh map without adapting the expression to Equation (3.11), but this takes a bit of practice! We would need to fill a cell with a '1' wherever there is an \bar{A} and \bar{C} . Equations (3.10) and (3.11) are both sum of product expressions.

Let's complete the map:

<div style="display: inline-block; transform: rotate(-45deg);">AB</div> <div style="display: inline-block; transform: rotate(45deg);">C</div>		0 0	0 1	1 1	1 0
		0	1	1	0
	1	0	0	1	0

We must now group together the '1's in 2^n adjacent cells. In this case, $2^1 = 2$ is appropriate. We have made the groups with two circles – red and green.

The red circle contains expressions $\bar{A}\bar{B}\bar{C}$ and $\bar{A}.B.\bar{C}$. You can see that they have only one difference, i.e. B and \bar{B} .

But this means that the two terms could have been factorised thus:

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

The same principle applies to the green circle i.e.

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

As can be seen therefore, the red group and the green group are doing the work of factoring the expression. All we have to do is look for the input variable that has more than one form within the group (e.g. B and \bar{B}) and ignore it.

Therefore, the simplified expression becomes:

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

(The number of expressions in the solution = the number of circles on the map).

Example 6

Simplify the Boolean expression

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

with a Karnaugh map.

Example 6 (solution)

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

In this case, $2^2 = 4$ cells are appropriate for the group. If we look at all of the input variables within the red circle, the only variable that does *not* have more than one form is C .

Therefore the solution is

$$F = C$$

Example 7

Simplify the Boolean expression

$$F = A.B + \bar{C}.D + C + \bar{B}.\bar{C}.D \quad (3.12)$$

with a Karnaugh map.

Example 7 (solution)

If we expand this expression, we could be here all day since it will be large! Therefore it's much better to use the given sum of products expression in Equation (3.12).

It is possible to put Equation (3.12) straight onto the Karnaugh map – it is after all, in sum of products form. The procedure is as follows:

Let's first of all deal with the term $A.B$. These variables are both '1s' so we know this term goes in the $AB=11$ column. In fact if we expanded $A.B$, we would end up with 4 'sum of products' terms where $A=B=1$ for each term. We should therefore fill this column up thus:

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

Now for $\bar{C}.D$. By the same reasoning, we need to fill up the $CD = 01$ row thus:

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

The next term is C . Here we put '1's where $C = 1$, i.e. in the bottom two rows thus. (We would end up with another 8 terms here if we expanded the expression):

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

Finally the term $\overline{B}\overline{C}.D$ would normally expand to two terms thus:

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

Finally, we have our completed Karnaugh map:

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

... which can be reduced to the three-term solution:

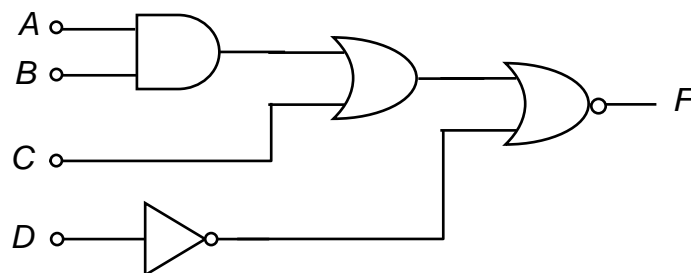
$$F = A.B + C + D$$

Remember – we require the *least* amount of circles with the *most* amount of ‘1’s in those circles, even if we use the ‘1’s twice!

Example 8

It is possible to reduce any logic circuit to a circuit which consists of only NAND gates or only NOR gates. For this reason, NAND and NOR gates are called ‘universal gates’. A circuit consisting just one type of gate can be advantageous for simplicity, cost and efficiency given that integrated circuits can be produced to consist of several of one type of logic gate. The example below addresses this issue by requiring a logic circuit to be converted to a NAND gate circuit.

Convert the logic circuit below into a circuit which consists of NAND gates only. Sketch the new NAND gate circuit.



Example 8 (solution)

From the circuit above

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

and we are seeking to change this expression into a circuit that consists of NAND gates only. We can do this by first of all removing the ‘OR’s with DeMorgan’s Theorem (see Section 3.2.3) i.e.

$$F = \overline{\overline{A.B} \cdot \overline{C + \bar{D}}}$$

A useful mnemonic for DeMorgan’s Theorem is “break the line and change the sign”

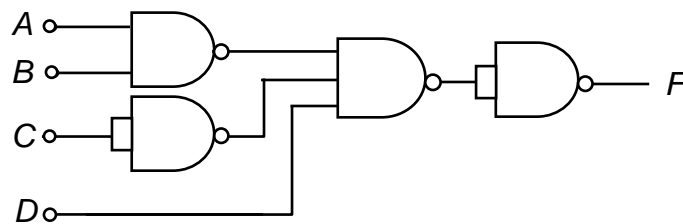
From identity (9) in Table 3.2.3

$$F = \overline{A.B.C.D}$$

Finally, we know that the *last* logic gate in the circuit (at the output) will be a NAND gate which will negate the complete expression – so there needs to be a ‘line’ above the complete expression to reflect this. The ‘trick’ is to put *two* lines above the complete expression – where one negates the other i.e.

$$F = \overline{\overline{A.B.C.D}}$$

This expression will now convert to a NAND gate logic circuit thus



Note: in the above circuit, where a single input variable is to become the input to a NAND gate, the connecting wire is split into two as can be seen in order to service two inputs. (There must be a least two inputs to a NAND gate). The circuit can be designed in this way because of identity (6) in Table 3.2.3.

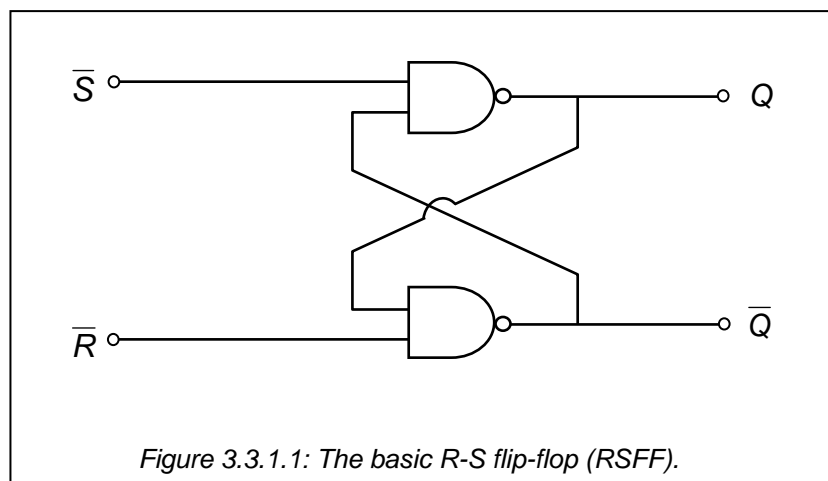
3.3 Sequential logic

We have found in the previous section that the outputs from combinational logic circuits depend upon the input at a given time. However, *sequential logic circuits* form the basis of memory circuits and therefore their output doesn't necessarily only depend upon the input state at a particular time but may also depend upon a previous input state. Indeed the output from the circuit may *only* depend upon some previous state.

Digital devices that retain information about previous input states are known as *information registers*.

3.3.1 The basic R-S flip-flop

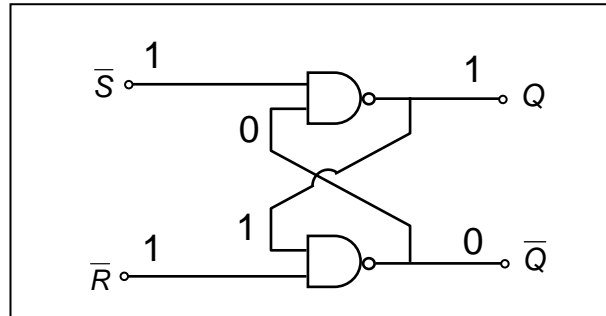
The basic R-S flip-flop is the simplest form of information register and consists of two cross-coupled NAND gates. It can be considered as a one-bit memory device, and is commonly used in memory circuits. The circuit diagram for the R-S flip-flop is shown below in Figure 3.3.1.1.



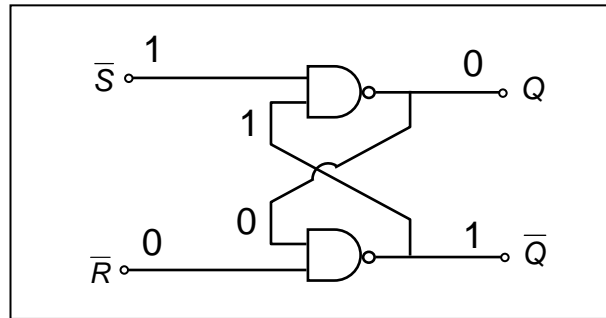
The R-S flip-flop has two inputs S (Set) and R (Reset), and two outputs Q and \bar{Q} . When the S and R inputs are held at logic 1, the outputs Q and \bar{Q} will consequently either be at '0' or '1', but will not be the same i.e. if Q is '0' then \bar{Q} will be '1' and vice versa. Thus the circuit has two stable output states and is therefore given the term *bistable*.

In order to change the output state, i.e. to flip Q and \bar{Q} over so that the one that was held at '0' goes to '1' and vice versa, then either the Set or Reset input has to be momentarily 'pulsed' to '0'. (Indeed the 'line' above the ' S ' and ' R ' in the diagram represents this 'triggered low' function. If there were no lines, we would know that the inputs would need to be momentarily pulsed to '1' from '0', i.e. 'triggered high' – a slightly different circuit design). In other words, the voltage on one of the inputs has

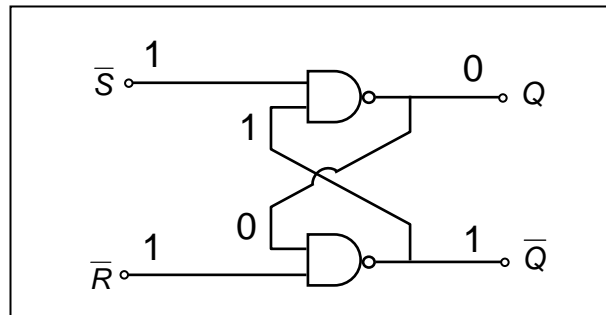
1. S and R are both at logic 1; Q and \bar{Q} are at logic 1 and 0 respectively. The initial output values are determined entirely by the order that the two inputs went to '1', even if one input became '1' a fraction of a second before the other!



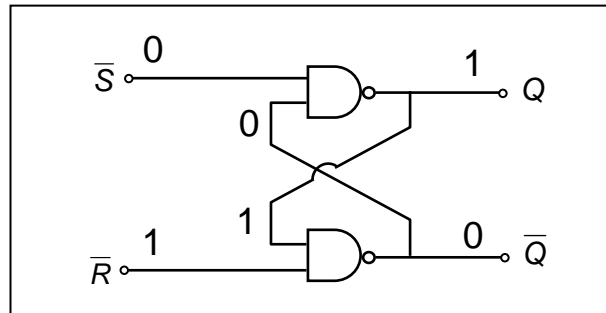
2. R is momentarily set to 0. \bar{Q} subsequently changes to 1. The new voltage from \bar{Q} is then 'fed' to the logic gate which seeds the output Q . Q subsequently changes to logic 0, and this voltage is then fed back to the input of the gate which seeds \bar{Q} . The circuit then becomes stable.



3. R is changed back to 1, with no further change to the outputs. Any subsequent change to R will yield no change to either output. The outputs are now opposite to what they were in (1) above. *The device has 'memorised' that R was the last input to become '0'.*



4. S is momentarily set to 0. Q subsequently changes to 1. The new voltage from Q is then 'fed' to the logic gate which seeds the output \bar{Q} . \bar{Q} subsequently changes to logic 0, and this voltage is then fed back to the input of the gate which seeds Q . The circuit then becomes stable.



5. S is changed back to 1, with no further change to the outputs. Any subsequent change to S will yield no change to either output. The outputs are now opposite to what they were in (3) above, and have returned to their values in (1) above. *The device has 'memorised' that S was the last input to become '0'.*

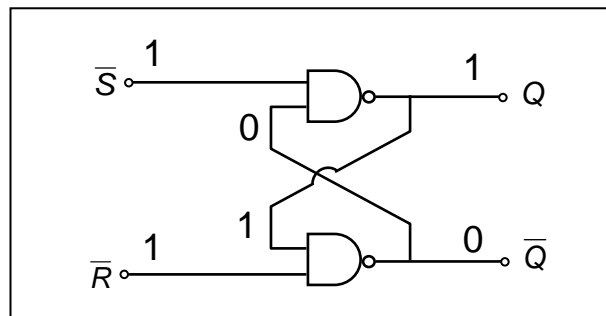


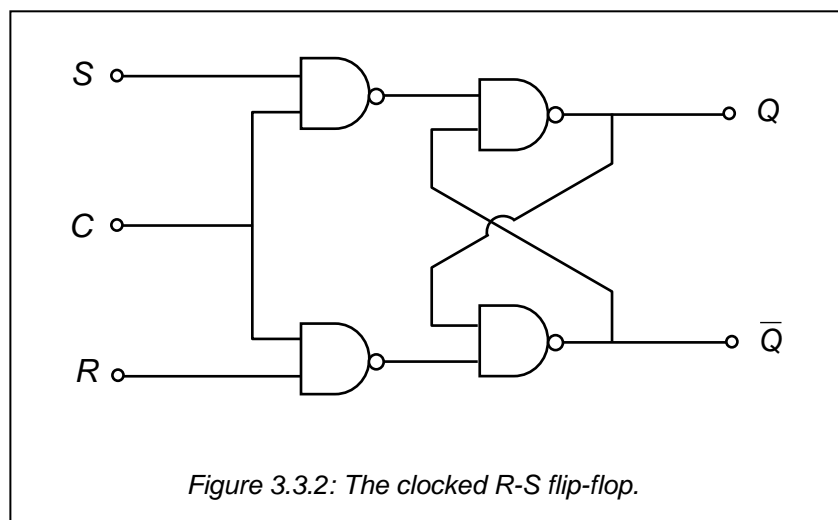
Figure 3.3.1.2: The R-S flip-flop sequence of events.

to be reduced to 0V for a moment. Any subsequent 'pulse' to 0 on the *same* input will have no effect. In order to flip the values of the outputs again, the *other* input must be pulsed to '0', and so on. When the device is first powered up, the output values depend on the order in which S or R reaches the required input voltage (for logic 1), even if the events occur a fraction of a second apart!

The full *sequence* of events is summarised in Figure 3.3.1.2.

3.3.2 The clocked R-S flip-flop

The *clocked R-S flip-flop* takes the basic R-S flip-flop a step further towards becoming a practical memory device. Whilst the R-S flip-flop was able to remember which input was low last, this data can be lost when further changes to inputs occur. The clocked R-S flip-flop, shown in Figure 3.3.2, offers an improvement to this design.



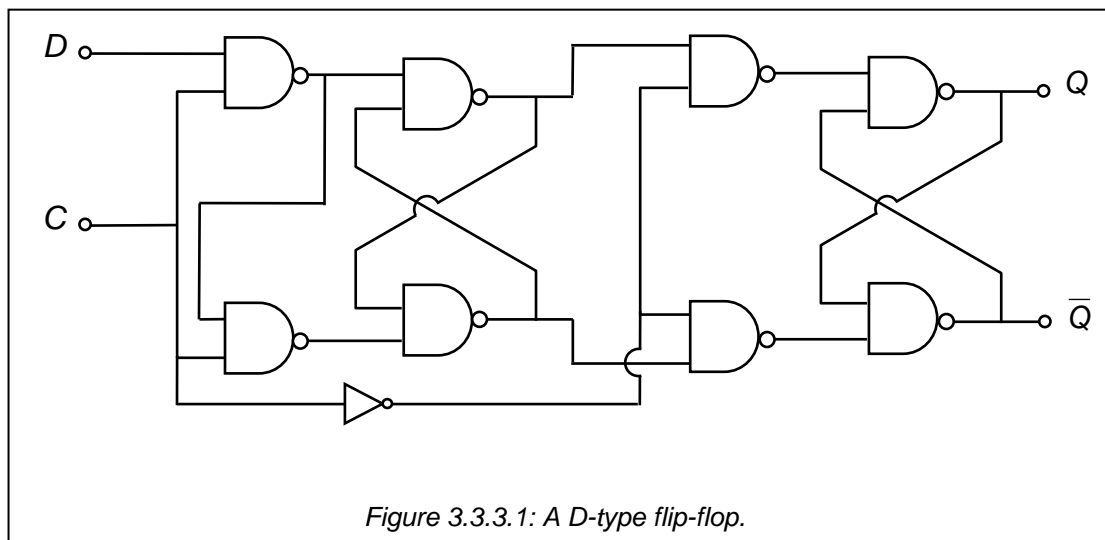
The outputs of the clocked R-S flip-flop are activated and deactivated by a clock (represented by the input 'C' in the figure). If the clock is at logic '1', the flip-flop operates as a normal R-S flip-flop, but with *positive pulse* capability. However, the outputs can be 'frozen' by forcing the clock to logic '0', whereby the flip-flop is essentially deactivated, whilst memorising the final output state reached. Thus the input information is only read and saved when the clock is activated. The 'clocked' memory is therefore permanent.

Recall that the basic R-S flip-flop had inputs R and S held high at '1' when in one of its stable states. The outputs were changed by pulsing one of the inputs momentarily to '0'. The opposite occurs with the clocked R-S flip-flop. Here, the inputs R and S are normally held low at '0' and pulsed high to '1' when the output is required to change. Pulsing R or S to '1' when C=1 will have the effect of pulsing to '0' an input to the right-hand-side of the circuit. (The right-hand-side of the circuit is equivalent to a basic R-S flip-flop).

However, the problem with the simple clocked R-S flip-flop is that there may be multiple changes to the inputs in quick succession which could change the output states if the clock is left high for too long. Even if the clock is turned on for only a fraction of a second, how can we be sure that the clocked flip-flop will memorise the single bit of data we are interested in? One solution is described in the next section with a flip-flop variation that is widely used.

3.3.3 The D-type flip-flop

The limitations of the simple clocked flip-flop described above are overcome here by extending the circuit further into a *D-type flip-flop*. The solution to making the clock pulse short enough only to record the data bit that we are interested in lies not with the synchronising of clock with data, but rather allowing the clock to be effective only at the *moment* when it is changing state. A schematic of the circuit is shown in Figure 3.3.3.1.



Having an output that matches the data input only at the precise moment that the clock changes, ensures that any data that follows immediately afterwards will be ignored. Depending on the design, either a 'rising edge' (clock changes from '0' to '1') will trigger the output to match the data input, or a 'falling edge' (clock changes from '1' to '0') will trigger the output to match the data input.

With reference to Figure 3.3.3.1, the data input is represented by '*D*' and the clock is represented by input '*C*'. The particular circuit shown is a 'falling edge' design. In other words, whatever the most recent value of the data input *D* when the clock changes from '1' to '0', will become the output *Q*, at that moment. The output *Q* will not change again unless both *D* changes *and* the clock falls from '1' to '0'. (\bar{Q} will continue to be in an opposite state to *Q*, and depending on the application, can itself be utilised. An example follows later in Section 3.3.5).

A useful method for enhancing understanding of flip-flops is to work through each circuit stage-by-stage, and record the changes that occur at the output of each logic

gate as a function of changes to circuit input. Moreover, there are various websites that ‘animate’ the zeros and ones throughout the circuit to show what changes occur at each stage as a consequence of changes to circuit input. These sites are well worth a look!

An example of the timing of a clock pulse alongside the corresponding data input and output for a ‘falling edge’ D-type flip-flop can be seen in Figure 3.3.3.2.

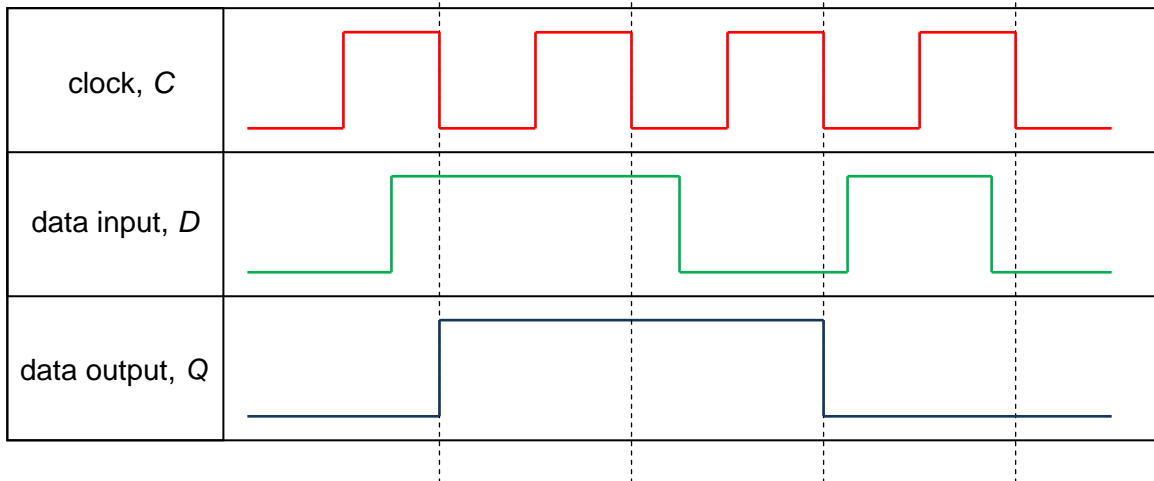
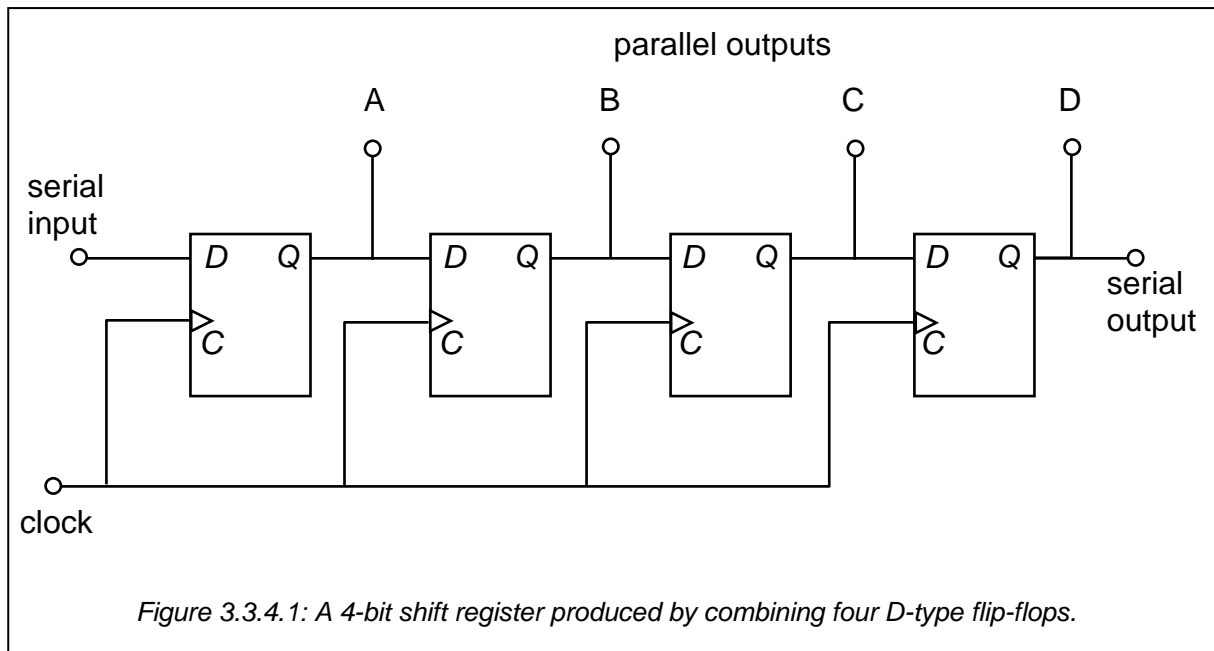


Figure 3.3.3.2: An example of how data is read and stored in a ‘falling edge’ D-type flip-flop. As can be seen, the input data is ‘memorised’ only when its timing coincides with the clock ‘falling’ from high to low.

3.3.4 Shift registers

There are a number of flip-flop types and designs for use in various applications – far too many to study (or even mention) in these notes. However, given that flip-flop circuits are widely used, information about the various designs is easy to find in generic electronics texts books and web pages. Here we will continue developing the particular design that we have begun, in order to introduce a couple of applications based on what we have already studied. Firstly the *shift register* in this section followed by the *binary counter* in the next section.

An array of D-type flip-flops can be used to produce a shift register, as shown in Figure 3.3.4.1. A shift register can store many bits of data to be accessed at a later time. As can be seen, the circuit diagram has been simplified with flip-flop circuit symbols, where *D*, *C* and *Q* have the same function as in Figure 3.3.3.1. The output \bar{Q} is not used in this circuit so is not shown. The small ‘triangle’ in the circuit symbol at each input *C* indicates that each flip-flop is edge-triggered. The circuit has a serial input (data bits input one at a time), parallel outputs A,B,C & D (data bits read simultaneously) and a serial output (data bits read one at a time).

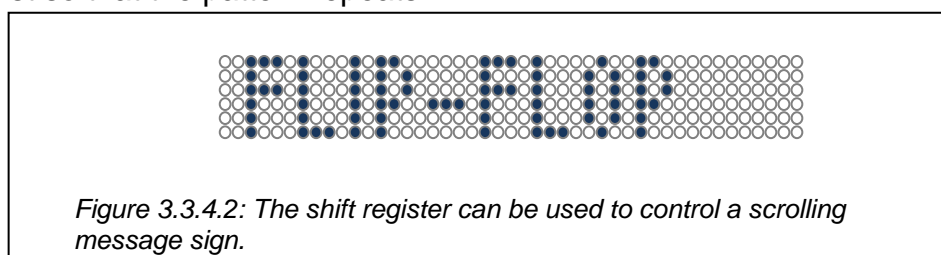


With reference to Figure 3.3.4.1, at the falling edge of the clock pulse, the data bit at each respective *D* input is transferred to the corresponding *Q* output, and hence to the *D* input of the following flip-flop. At each subsequent falling edge of the clock pulse, the data is *shifted* along the chain once more. At the fourth falling edge of the clock pulse, the data at the input of the first flip-flop (on the left of the circuit) will be moved to the parallel output D.

A four-bit data number at the serial input can be read in full from the parallel outputs after four clock pulses, or can be transferred to the serial output after more clock pulses (a slower process). This particular example of a shift register is known as a *serial-to-parallel shift register* due to its parallel output capability.

If the shift register in Figure 3.3.4.1 consisted of eight flip-flops instead of four, then one byte (eight bits) could have been stored. The entire byte would have been present at the flip-flop outputs in parallel format.

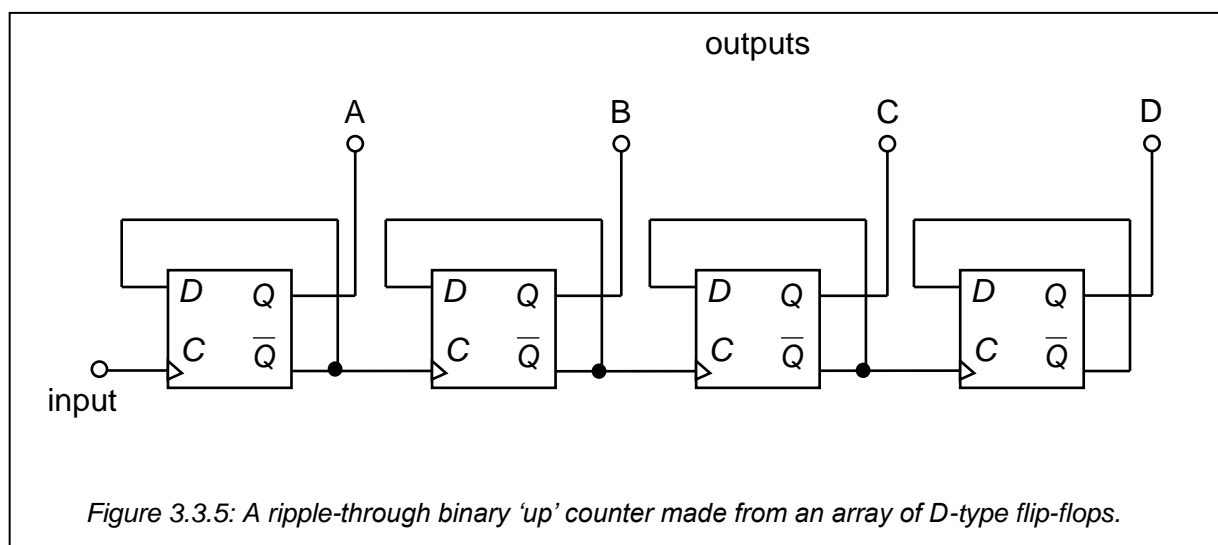
Figure 3.3.4.2 demonstrates another application of a shift-register: the ubiquitous scrolling message sign. By connecting each of the parallel outputs to LEDs (represented by small circles in the figure), we would see a pattern scroll across the display at a rate set by the clock. For each row of LEDs, we would require an additional shift register, and each shift register would be synchronised by the same clock. The output from the last flip-flop in each row could be connected to the input of the first so that the pattern repeats.



3.3.5 Binary counters

If we connect several D-type flip-flops together in the way shown in Figure 3.3.5, we obtain a *binary counter*. Computers for example, are required to perform a number of operations and one of the most fundamental to all mathematical processes is counting.

The particular design shown is known as a 'ripple-through' binary 'up' counter because there is one input which changes from 1-0-1-0.... etc. and will effect a 'ripple through' of 1s and 0s. The outcome is that the parallel outputs A, B, C & D will count in binary in an 'upwards' direction from binary 0000-1111 (decimal 0-15). By connecting more flip-flops to the chain, counts up to a greater binary number can be achieved.



The design of the binary counter shown in Figure 3.3.5 is such that the inverse output \bar{Q} is connected to its own data input D , and also to the clock input of the next flip-flop. Q in each case constitutes the output.

In this particular example, the flip-flops are wired to change their output on a 'rising edge'.

The sequence of events of the counter shown in Figure 3.3.5 is shown in Table 3.3.5. For clarity, the table is written in the order A, B, C, D. However, the actual four-bit number is the opposite way around: D, C, B, A.

input	A (\bar{Q}, D)	B (\bar{Q}, D)	C (\bar{Q}, D)	D (\bar{Q}, D)	Decimal
0	0 (1)	0 (1)	0 (1)	0 (1)	0
1	1 (0)	0 (1)	0 (1)	0 (1)	1
0	1 (0)	0 (1)	0 (1)	0 (1)	1
1	0 (1)	1 (0)	0 (1)	0 (1)	2
0	0 (1)	1 (0)	0 (1)	0 (1)	2
1	1 (0)	1 (0)	0 (1)	0 (1)	3
0	1 (0)	1 (0)	0 (1)	0 (1)	3
1	0 (1)	0 (1)	1 (0)	0 (1)	4
0	0 (1)	0 (1)	1 (0)	0 (1)	4
1	1 (0)	0 (1)	1 (0)	0 (1)	5
0	1 (0)	0 (1)	1 (0)	0 (1)	5
1	0 (1)	1 (0)	1 (0)	0 (1)	6
0	0 (1)	1 (0)	1 (0)	0 (1)	6
1	1 (0)	1 (0)	1 (0)	0 (1)	7
0	1 (0)	1 (0)	1 (0)	0 (1)	7
1	0 (1)	0 (1)	0 (1)	1 (0)	8
0	0 (1)	0 (1)	0 (1)	1 (0)	8
1	1 (0)	0 (1)	0 (1)	1 (0)	9
0	1 (0)	0 (1)	0 (1)	1 (0)	9
1	0 (1)	1 (0)	0 (1)	1 (0)	10
0	0 (1)	1 (0)	0 (1)	1 (0)	10
1	1 (0)	1 (0)	0 (1)	1 (0)	11
0	1 (0)	1 (0)	0 (1)	1 (0)	11
1	0 (1)	0 (1)	1 (0)	1 (0)	12
0	0 (1)	0 (1)	1 (0)	1 (0)	12
1	1 (0)	0 (1)	1 (0)	1 (0)	13
0	1 (0)	0 (1)	1 (0)	1 (0)	13
1	0 (1)	1 (0)	1 (0)	1 (0)	14
0	0 (1)	1 (0)	1 (0)	1 (0)	14
1	1 (0)	1 (0)	1 (0)	1 (0)	15
0	1 (0)	1 (0)	1 (0)	1 (0)	15
1	0 (1)	0 (1)	0 (1)	0 (1)	0
0	0 (1)	0 (1)	0 (1)	0 (1)	0

Table 3.3.5: The sequence of events which describes the operation of the binary counter shown in Figure 3.3.5.

NOTE: This is probably a good moment to mention that one should always be careful of variations in the way circuits can be presented. In Figure 3.3.5, where wires are connected together, there is a black 'dot'. When the black dots are present in the circuit, this means that if other wires appear to be touching or connected, they are not! Therefore the wire which travels from the output \bar{Q} to the input D does not connect with Q . In many previous circuit diagrams in these notes, there have been no black dots, but instead a 'bridge' was drawn in the wire when it crossed another without being connected. You may see either style depending on which further source you look at. Moreover, the convention for resistors changes a lot too! Here, 'rectangles' have been used to represent resistors but you could just as easily find another source that uses zig-zag lines to represent a resistor! You should become familiar with all of these variations.

3.3.6 Memory chips

For storing a large amount of data we can use a large array of many flip-flops on a single chip. A large array such as this is called a *memory*. Each flip-flop stores a single bit of data. A *RAM chip* (random access memory) may have thousands of flip-flops on it. Once data is stored on a flip-flop in a RAM chip, it remains there until it is either changed by writing new data, or power to the chip is removed. This type of RAM is known as *static RAM* or SRAM.

Dynamic RAM or DRAM is another type of RAM where bits of data are not stored in flip-flops but as charges on the gate of MOSFETs. If the gate is charged, the MOSFET is on and if it holds no charge, it is off. Reading the data consists of checking if the transistor is on or off. The disadvantage with DRAM is that the charge on the gate is not permanently held, and therefore needs to be refreshed (or re-written) at regular intervals. The charge in this type of memory which is 'topped up' when moved from transistor to transistor, is continually being moved around or refreshed – hence the term 'dynamic'.

A *flash memory* needs no power to keep its data stored and is therefore ideal for portability. The memory consists arrays of MOS transistors where some are completely insulated and therefore hold charge indefinitely.

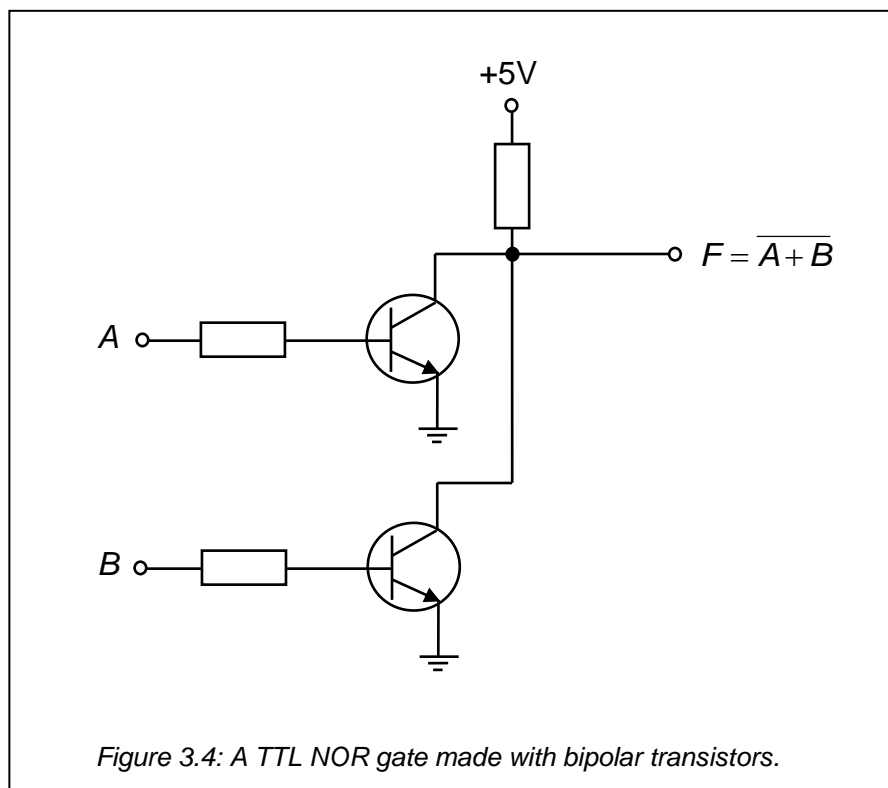
RAM memory can have data read in an order chosen by the user in contrast to a shift register where the data can only be accessed in the order of their position within the register.

Other terms used for memory chips, (which come in a huge variety of sizes and types), are *volatile* and *non-volatile*. Volatile memory devices lose stored information when power is removed whilst non-volatile memory does not.

3.4 Manufacture of logic gates: the pros and cons

Previously in these notes it has been mentioned several times that logic gates are basically made from transistors. Given that there are various types of transistors then it should come as no surprise that logic gates also come in various types. Each type has its own version of the standard logic gate such as NAND or NOR. Moreover, each type or family includes a range of integrated circuits with more complex functions such as flip-flops and counters.

One common type is transistor-transistor logic (TTL). In this family, bipolar junction transistors are used to create the logic gates. An example is given in Figure 3.4 where a TTL NOR gate is shown. If either (or both) A or B are high, then the output is low. If both A and B are low then the output is high.



Another important logic gate family consists of the complementary metal oxide semiconductor (CMOS) family, which are made from field-effect transistors. The 'complementary' derives from the fact that the transistors operate in pairs, one being n-channel and the other being p-channel.

For all but the fastest logic, the choice usually lies somewhere between TTL and CMOS. However, there are some special purpose logic families, for example the fast-switching emitter coupled logic (ECL) family.

The pros and cons of the families that have been mentioned here are summarised in Table 3.4.

family	pros	cons
TTL	Common, fast, cheap.	High power consumption.
CMOS	Low power consumption. Suitable for large-scale integration.	Relatively slow.
ECL	Fastest.	High power consumption. Low noise immunity.

Table 3.4: The pros and cons of the various families of logic.

A final note ...

This part of the module has offered an introduction to electronics and in doing so has covered several topics necessary for the development of knowledge and problem-solving in this field. However, the fast-paced advance in electronic technology that we see all around every day informs us that the field of electronics is vast and quickly expanding. For students that are perhaps interested in developing their electronics knowledge further, there are various important topics that form a natural step beyond this introductory course: binary adders, display devices, multiplexers and demultiplexers, digital to analogue and analogue to digital converters, and oscillators are all important topics and should be straightforward to follow with self-study.

I hope that you have enjoyed the course!

John Mills