

# Chapter 2

## Logic and sequencing

### Chapter summary

This chapter covers Boolean algebra and binary numbers, fundamental gates and truth tables, and simple digital circuits analysis and synthesis. Both combinatorial circuits and sequential circuits are discussed. Finite State Machines (FSM) are introduced as vehicles for sequential circuit design.

### Learning outcomes

After studying this chapter you should aim to test your achievement of the following outcomes by answering the questions at the end of the chapter. You should be able to do the following:

- Outcome 1:** Understand the relationship between the denary and binary number systems, how to represent denary integers as binary numbers and why the binary number system is used in computer systems.
- Outcome 2:** Recognise the symbols of the basic logic gates and define by means of truth tables the basic logic operations.
- Outcome 3:** Use basic identities of Boolean algebra for manipulating logic expressions.
- Outcome 4:** Analyse the functionality of simple digital circuits by using truth tables and synthesise simple digital circuits starting from their truth table.
- Outcome 5:** Have an understanding of the major combinatorial and sequential functional blocks and their possible uses within the computer system.
- Outcome 6:** Be familiar with the basic concepts of finite state machines and their role in the design of sequential functional blocks.

### How will you be assessed on this?

Assessment is likely to be centred on your understanding of definitions and number conversion rules. You may also be assessed on your ability to use Boolean operators and their respective truth tables and logic gate symbols associated with these operators. Also, your ability to analyse such circuits using a truth table approach and your ability of recognising their function, given their corresponding Boolean expression and/or gate diagram may be assessed. Finally, assessment is also likely to concentrate on your understanding of functionality of the fundamental sequential circuits and your ability to analyse such circuits using a state table approach.

### Useful information sources

Knapp, S. (1997) *Frequently-Asked Questions (FAQ) About Programmable Logic*, OptiMagic™, Inc, <http://www.optimagic.com/faq.html#Top>

## Section 1

# The binary number system

The binary, octal and hexadecimal number systems are introduced.

## The binary number system

Our present system of numbers is based on ten separate symbols (0, . . . ,9) and is called the denary (sometimes, incorrectly, the decimal system) number system. A single place that can hold numerical values between 0 and 9 is called a digit. Digits are normally combined together in groups, by using positional notation, to create larger numbers. That means that different digits have different 'powers', according to their position within a number.

### Example 1

2875 has four digits; the 5 here is filling the '1s place', the 7 is filling the '10s place', the 8 is filling the '100s place' and the 2 is filling the '1000s place'.

The number 2875 can therefore be expressed as follows:

$$2875 = 2*1000 + 8*100 + 7*10 + 5*1$$

or, using powers of 10:

$$2875 = 2*10^3 + 8*10^2 + 7*10^1 + 5*10^0$$

Each digit is a placeholder for the next higher power of 10, starting in the first digit on the right with 10 raised to the power of 0.

The general rule for representing numbers in the denary system using positional notation is as follows:

$$a_{n-1}a_{n-2}\dots a_2a_1a_0 = a_{n-1}*10^{n-1} + a_{n-2}*10^{n-2} + \dots + a_2*10^2 + a_1*10^1 + a_0$$

where  $n$  is the number of digits to the left of the decimal point.

### CRUCIAL CONCEPT

The system used in computers for representing and processing data and information is a **binary number system**, or base-2 number system. Any number can be represented using the two symbols 0 and 1, by using the same positional notation as in the denary system, for example  $1101_2$  (the subscript indicates the base in which the number is given).

But how does one figure out what the value of the binary number  $1101_2$  is? In fact it works in the same way as Example 1. The procedure is shown in Example 2.

### Example 2

$$1101_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

Each binary digit holds the value of increasing powers of 2, from the right to the left.

### CRUCIAL CONCEPT

A binary digit is called a bit (short for Binary digit).

**Example 3**

Counting in binary

Binary and denary numbers

Denary	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Note that, in the first two lines of the table, the numbers 0 and 1 are the same for the binary and denary systems. On the third line, at number 2 in denary, carrying takes place in binary: if a bit is 1 and one adds 1 to it, the bit becomes 0 and the next bit to the left becomes 1. Two bits are necessary to represent the numbers from 0 to 3, three bits for 0 to 7 and so on.

**CRUCIAL TIP**

How many **bits** are needed? The general rule is: *with  $n$  bits one can represent the numbers from 0 to  $(2^n - 1)$ .*

In computer systems, a group of 8 bits is called a byte or octet. Following the rule above, a byte can represent numbers from 0 to 255. Bytes are frequently used to hold individual characters in a text document. In the ASCII character set, each binary value between 0 and 127 is given a specific character. The other 128 values handle special things like accented characters from common foreign languages.

To represent lots of bytes, the following prefixes are used:

$$\text{Kilo } 1\text{K} = 2^{10} = 1024$$

$$\text{Mega } 1\text{M} = 2^{20} = 1,048,576$$

$$\text{Giga } 1\text{G} = 2^{30} = 1,073,741,824$$

1Kbyte is about 1000 bytes, 1Mbyte is about one million bytes and 1 Gbyte is about 1 billion bytes.

**The octal and hexadecimal number systems**

The octal (base-8) and hexadecimal (base-16) number systems provide a convenient shorthand representation for multibit numbers in a digital system, as their bases are powers of 2. The octal number system needs 8 digits; it uses digits 0-7 of the decimal system. The hexadecimal number system needs 16 digits; it uses digits 0-9 and the letters A-F. In representing multibit numbers, a string of three bits can take eight different combinations, so each 3-bit string can be uniquely represented by an octal digit. Likewise, a 4-bit string can be uniquely represented by one hexadecimal digit.

## Binary, denary, octal and hexadecimal numbers

Binary	Denary	Octal	3-bit string	Hexadecimal	4-bit string
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	-	8	1000
1001	9	11	-	9	1001
1010	10	12	-	A	1010
1011	11	13	-	B	1011
1100	12	14	-	C	1100
1101	13	15	-	D	1101
1110	14	16	-	E	1110
1111	15	17	-	F	1111

To convert a number from binary to octal, the following steps should be followed:

- start from the right end of the bit string towards the left;
- add zeroes on the left to make the total number of bits a multiple of three;
- separate the bits into groups of three;
- replace each group with the corresponding octal digit.

The procedure for binary to hexadecimal conversion is similar, except that groups of 4 bits are used and replaced.

**Example 4**

Convert the following binary number to octal and hexadecimal number bases.

$$11101110001011_2 = 011101110001011_2 = 011\ 101\ 110\ 001\ 011_2 = 35613_8$$

$$11101110001011_2 = 0011101110001011_2 = 0011\ 1011\ 1000\ 1011_2 = 3B8B_{16}$$

## Why use the binary number system in computer systems?

Two-state circuits (also called binary) or digital, are very resistant to noise, easy to design, simple to understand and extremely reliable. Information/data can be easily manipulated by using very simple electronic circuits called gates, as will be shown in the next section.

## Section 2

# Boolean algebra and logic gates

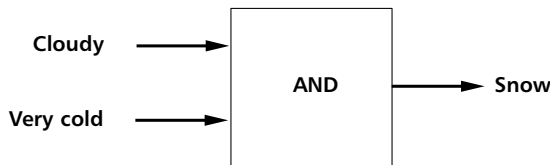
This section talks about gates (the most primitive logic circuit elements used in modern computers) as hardware representations/implementations of the Boolean operators AND, OR, NOT.

## Boolean algebra

In propositional logic propositions may be TRUE or FALSE and are stated as functions of other propositions which are connected by the three basic logical connectives AND, OR and NOT. An example is given below.

### Example 5

The statement 'IF it is cloudy AND it is very cold THEN it will snow' is composed of the input propositions 'it is cloudy' and 'it is very cold' and has as an output the proposition 'it will snow'. The meaning of AND is that the output proposition is TRUE if and only if both input propositions are TRUE.



As there are only two possible values for any input proposition, we can calculate a truth value for the output proposition for all possible input combinations, as described in the following *truth table*.

Truth table for statement in Example 5

Input 1: 'it is cloudy'	Input 2: 'it is very cold'	Output: 'it will snow'
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Logic propositions can be made as complex as required, by using the connectives AND, OR and NOT. Boolean algebra (developed by Boole during the 19th century) simplifies the handling of binary connectives, using ordinary algebra notation and 1 for TRUE and 0 for FALSE. In this book, the symbol \* is used for AND, + for OR and  $\bar{A}$  to denote NOT A.

The truth tables of the three basic binary operations are given in the table below, where A and B are the inputs and Y is the output.

Truth table of AND, OR and NOT operations

AND			OR			NOT A	
*			+			$\bar{A}$	
A	B	Y	A	B	Y	A	Y
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

These truth tables can be used to evaluate the overall truth of more complex expressions. The total number of possible input combinations in a truth table (number of lines in the table) is related to the number of inputs as follows:

CRUCIAL TIP

For  $n$  inputs, the total number of combinations will be  $2^n$ .



A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gates. By implementing such a logic diagram in hardware, we obtain a digital circuit.






## Simple gates

Digital circuits are hardware components that manipulate binary information.

The circuits are implemented using transistors within integrated circuits. Each basic circuit of transistors is referred to as a **logic gate**. This section is not concerned with the internal electronics of individual gates but only with their external logic properties; logic gates operate on one or more input binary signals to produce an output binary signal. Functionality of the logic gates and generally of digital circuits is described using truth tables.

There are seven simple gates, which in combination will implement any functional block within the computer system. Each type of gate has a name, a graphical symbol, a Boolean logic function and a truth table.

Gate name	Symbol	Logic function	Truth table															
NOT (aka 'Inverter')		$Y = \bar{A}$	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>A</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	Y	0	1	1	0									
A	Y																	
0	1																	
1	0																	
AND		$Y = A*B$	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																

OR		$Y = A + B$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NAND (NOT AND)		$Y = \overline{A * B}$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR (NOT OR)		$Y = \overline{A + B}$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive OR (XOR) (‘Difference’)		$Y = A \oplus B$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive NOR (XNOR) (‘Equality’)		$Y = \overline{A \oplus B}$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

## Basic identities and laws of Boolean algebra

Boolean algebra facilitates the analysis and design of digital circuits. It provides a convenient tool to:

- express in algebraic form a truth table relationship between binary variables;
- express in algebraic form the input-output relationship of logic diagrams;
- find simpler circuits for the same function.

By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates for its implementation. The basic identities and laws of Boolean algebra by which these manipulations can be performed are presented below:

### Commutative laws

$$A + B = B + A$$

$$A * B = B * A$$

### Associative laws

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

$$A * (B * C) = (A * B) * C = A * B * C$$

**Distributive laws**

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

**Other laws**

$$A + A = A$$

$$A * A = A$$

$$A + 1 = 1$$

$$A * 1 = A$$

$$A + 0 = A$$

$$A * 0 = 0$$

$$A + \bar{A} = 1$$

$$A * \bar{A} = 0$$

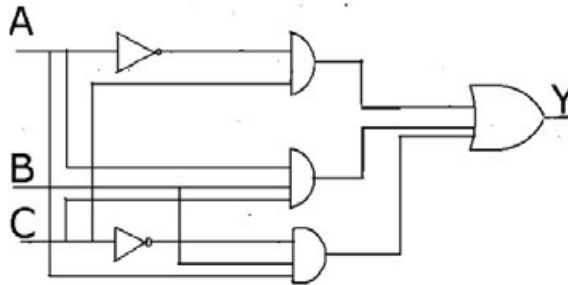
$$\bar{\bar{A}} = A$$

**de Morgan's law**

$$\overline{(A + B)} = \bar{A} * \bar{B}$$

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

The following example shows how Boolean algebra manipulation can be used to simplify digital circuits. Consider the circuit below.



The output of the circuit can be expressed algebraically as follows:

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

The expression can be simplified using the identities of Boolean algebra:

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

Note that  $C + \bar{C} = 1$  and  $A * B * 1 = A * B$ , from the identities above. As an exercise, draw the logic diagram of the simplified circuit.

**Canonical forms and circuit synthesis**

A canonical form is a standard way of writing a mathematical equation. Any combinatorial circuit can be expressed in one of two canonical forms: sum-of-products ('or of ands') and product-of-sums ('and of ors'). Of the two, only the sum-of-products form is discussed here as it is the most commonly used. This canonical form is a set of Boolean equations, one for each output, in which each equation is a Boolean sum of minterms, Boolean product terms in which each input appears exactly once, either unmodified or inverted. For each input combination leading to an output of 1, the corresponding minterm must appear in the canonical expression of the Boolean function.

Consider the following truth table for a majority voter, where A, B, C are the inputs and Y is the circuit output. The output is 1 if and only if two or more of its inputs are 1.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1 (minterm)
1	0	0	0
1	0	1	1 (minterm)
1	1	0	1 (minterm)
1	1	1	1 (minterm)

The canonical form for the circuit described in the above table is:

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

You can see here why the canonical form deduced is called the 'sum-of-products' form, as it reflects that the circuit function is described as a Boolean sum of the minterms, which in turn are Boolean products of the inputs. The canonical form allows the development of simple algorithms for synthesising any circuit automatically, starting from a set of Boolean equations and finishing with an integrated circuit, called a programmable logic array.

## Section 3

# Simple combinatorial circuits

## Combinatorial circuits

### CRUCIAL CONCEPT

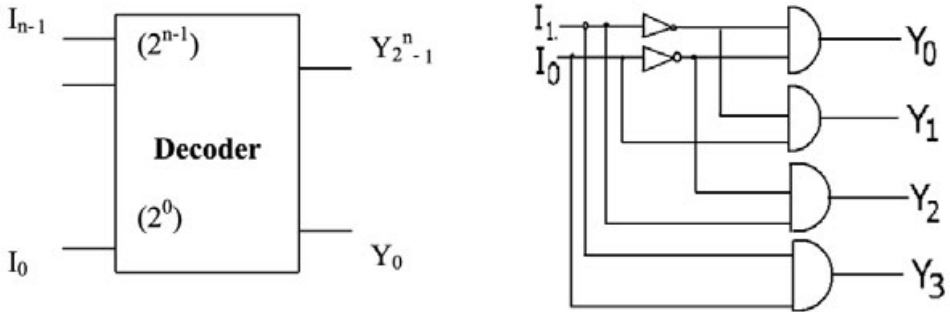
**Combinatorial circuits** are logic circuits in which the output immediately reflects the state of the inputs.

For a combinatorial circuit of  $n$  input variables, there are  $2^n$  possible input combinations. A combinatorial circuit can be specified by a truth table that lists the output values for each input variables combination, which in turn can be simply converted to a 'sum-of-products' canonical form.

From the large class of combinatorial circuits, the focus will be set in this section on several 'reusable' blocks (blocks which are used in more than one place within the hardware computer structure), which provide functions that are broadly useful. These blocks are sometimes called **functional blocks**.

## Decoders

A decoder is a combinatorial circuit that converts binary information from the  $n$  coded inputs to a maximum of  $2^n$  unique outputs (figure below). If the  $n$ -bit coded information has unused bit combinations, the decoder may have fewer than  $2^n$  outputs.



The logic diagram of a 2-to-4 decoder is shown. The two inputs are decoded into four outputs. The truth table is shown below.

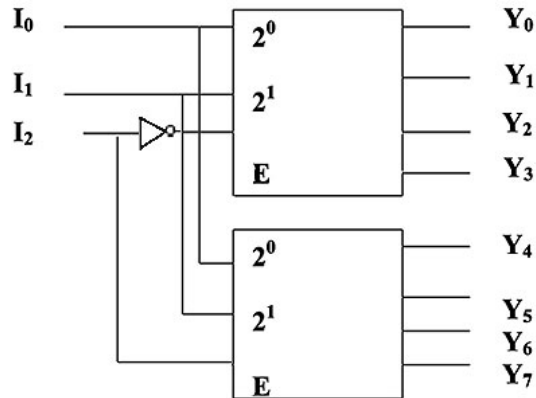
$I_1$	$I_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

### Exercise

Redesign the decoder just given to include an 'enable' input,  $E$ , such that when the enable line is 0, all outputs are 0, when  $E$  is 1, the selected output is 1, as shown in the truth table below.

Inputs			Outputs			
$E$	$I_1$	$I_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Obtaining higher order decoders from low order ones is possible by connecting them through their enable line. For example, a 3-to-8 decoder, useful for binary-to-octal transformations, can be obtained by connecting two 2-to-4 decoders as shown below. As an exercise, check the functionality of this decoder using a truth table approach.

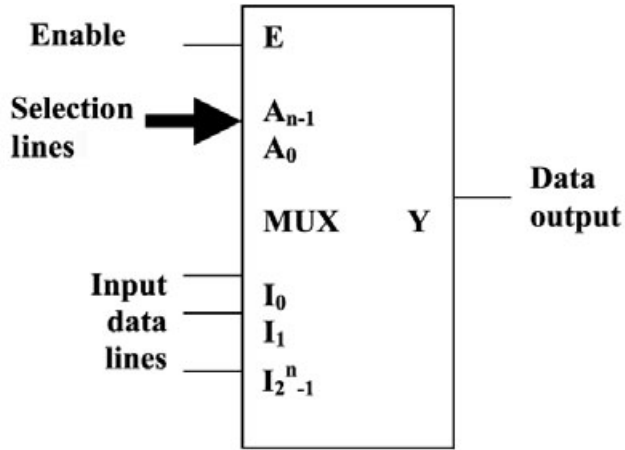


## Encoders and multiplexers

An encoder is a digital circuit that performs the inverse operation of a decoder. It has  $2^n$  (or less) input lines and  $n$  output lines. An example of an encoder is the octal-to-binary encoder, whose truth table is given in the table below. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

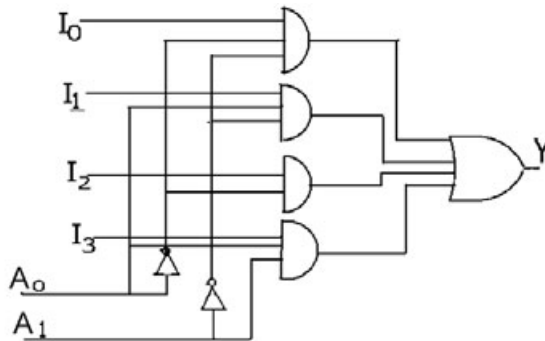
Inputs								Outputs		
$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	$A_2$	$A_1$	$A_0$
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

A multiplexer (MUX) is a combinatorial circuit that receives binary information from one of  $2^n$  input data lines and directs it to a single output line, determined by a set of  $n$  selection inputs.



The multiplexer is also called a data selector, since it selects one of many data inputs and steers the binary information to the output.

The gates implementation and function of a 4-to-1-line multiplexer is shown below.



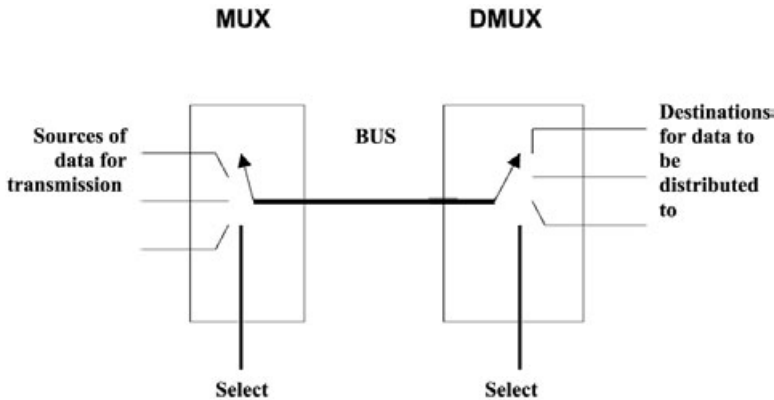
SELECT		OUTPUT
$A_1$	$A_0$	$Y$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in an inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as normal.

## Demultiplexer

A demultiplexer is a digital circuit that performs the inverse of the multiplexing operation. A demultiplexer receives information from a single line and transmits it to one of  $2^n$  possible output lines. The selection of the specific output is controlled by the bit combination of  $n$  selection lines.

A MUX can be used to select one of  $n$ -sources of data to transmit on a bus. At the far end of the bus, a DMUX can be used to re-route the bus data to one of  $m$  destinations.



## Arithmetic circuits

An arithmetic circuit is a combinatorial circuit that performs arithmetic operations such as addition, subtraction, multiplication and division, with binary numbers or with decimal numbers in a binary code.

### The half adder

A combinatorial circuit that performs the addition of two bits is called a half adder. The simple addition of two bits is as follows:

$$\begin{aligned} 0 + 0 &= 00 \\ 0 + 1 &= 01 \\ 1 + 0 &= 01 \\ 1 + 1 &= 10 \end{aligned}$$

The first three operations produce a sum requiring only one bit to represent, but for the fourth operation two bits are required. For this reason, two bits are always used to represent the sum of two bits: a sum bit and a carry bit.

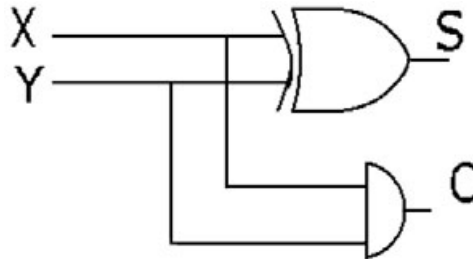
Below is the truth table of a half adder, where  $X$  and  $Y$  are the inputs (the bits to be added) and  $C$  and  $S$  are the outputs ( $C$  – carry,  $S$  – sum).

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The Boolean functions for the two outputs, obtained from the truth table, are:

$$S = (\bar{X} * Y + X * \bar{Y}) = X \oplus Y$$

$$C = X * Y$$



The half adder can be implemented with one XOR gate and one AND gate.

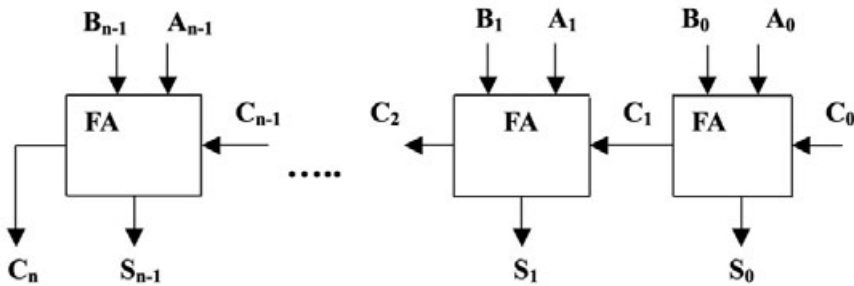
## The full adder

A full adder performs the arithmetic sum of three input bits and has three inputs and two outputs. The truth table of the full adder is shown in the section below. The third input,  $C_{in}$ , represents the carry from the previous lower significant position. The full adder can be built from two half adders and an OR gate.

## n-bit binary adder

A parallel binary adder uses  $n$  full adders in parallel, with all input bits applied simultaneously to produce the sum. The full adders are connected with the carry output from one full adder connected to the carry input of the next full adder. The more bits are there to be added, the longer it will take for the adder to complete the sum.

Inputs			Outputs	
X	Y	C <sub>in</sub>	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



## Subtractors

Subtraction circuitry can be designed by applying the same procedures as for the half and full adders. However, for the sake of modularity and blocks reuse, in practice subtraction is performed by using two's complements and full adders. Consider two integers, A and B. In order to subtract B from A, the following steps need to be performed:

- calculate  $(-B)$  by finding the two's complement of B;
- add this result to A.

The two's complement of a binary digit is found by complementing the individual bits of a number and adding one to the bottom digit.

## Multiplication and division

Multiplication of two binary numbers is carried out by multiplying all combinations of individual digits and then adding them up in the appropriate positions. An example of the procedure followed for decimal and binary multiplication is shown below. Suppose one has to multiply 21 and 43.

$$21 \times 43 = 2 \times 4 \times 10^2 + 2 \times 3 \times 10^1 + 1 \times 4 \times 10^1 + 1 \times 3 \times 10^0$$

Applying the same principle for the binary two digit numbers  $A = A_1A_0$  and  $B = B_1B_0$ , one obtains:

$$A_1A_0 \times B_1B_0 = A_1 \times B_1 \times 2^2 + A_1 \times B_0 \times 2^1 + A_0 \times B_1 \times 2^1 + A_0 \times B_0 \times 2^0$$

As all digits involved are binary ones, the multiplies can be replaced by ANDs and the multiplies by 2 can be replaced by shift right.

Division is not usually built as a combinatorial circuit. Instead, it is performed procedurally, with a sequential circuit, or in the machine code.

## Section 4

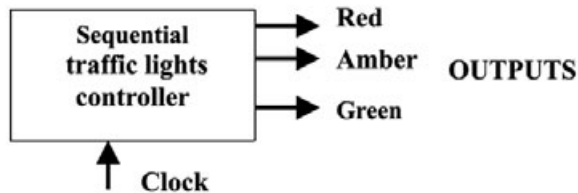
# Sequential logic and finite state machines (FSM)

This section discusses sequential logic circuits and their uses within the computer system. It shows how simple sequential circuits can be built from gates and gives some examples of sequential circuit design.

### CRUCIAL CONCEPT

In many digital designs there is a need for logic circuits whose outputs depend not only on the present inputs, but also on the past history, or sequence of actions of the circuit. This is achievable by building memory-type circuits, which are able to store information about the past history of the circuit. Such circuits are known as **sequential logic circuits**.

One simple example of usage for such circuits is the 'traffic lights controller'.

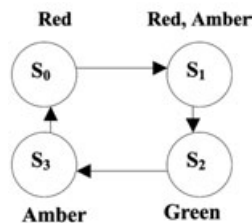


The traffic lights controller works as follows: if the colour displayed by the traffic lights is green, at the next 'clock' the lights will change to red and amber, then red and so on. The 'next' colour to be displayed depends on the colour displayed 'now'.

### CRUCIAL CONCEPT

The **state** of a system is its condition at a particular moment, described by the value of its outputs or 'variables'.

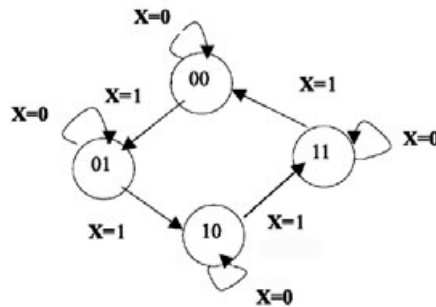
The traffic lights controller requires four states, named  $S_0$ ,  $S_1$ ,  $S_2$ ,  $S_3$ . The circuit evolves from one state to another under the control of a clock signal. Such circuits are called synchronous circuits.



In this state diagram the arrows signify transitions from one state to another which happen only on receipt of a clock 'tick'.

A circuit with  $n$  binary state variables has  $2^n$  possible states. As the number of possible states is always finite, sequential circuits are often called finite state machines (FSMs). The behaviour of FSMs is described by means of state tables, which contain the following columns: present state, input, next state and output. The lines of the state table contain all possible combinations of the present states and inputs.

Suppose the requirement is to design a binary counter, which counts in the sequence 0, 1, 2, 3. The first step in the design procedure is to translate the circuit specification into a state diagram. The state diagram is then converted to a state table, from which is designed the logic. As the circuit has to cope with four states, 0, 1, 2 and 3, two bits will suffice. The binary states, therefore, are: 00, 01, 10 and 11. The circuit has to change state when the clock 'ticks', and when an external input,  $X$ , equals 1 (true). When  $X$  is 0 (false), nothing will happen. These values 'label' the transition arrow to which they refer.



The state table can be drawn as below, with two state variables,  $A$  and  $B$  assigned to the two bits representing the states. In this example the ' $X$ ' input is the same as the clock.

Present state		Input $X = Ck$	Next state	
$A$	$B$		$A^+$	$B^+$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

We now look at some functional blocks which will allow us to store these states, a memory device.

## Flip-flops and memory devices

### CRUCIAL CONCEPT

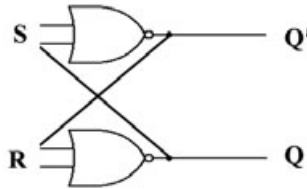
The simplest sequential circuit is the **flip-flop** — a memory/storage element for one bit.

The flip-flop represents the basic building block for sequential circuits. According to their internal structure, there are several types of flip-flops, of which two are discussed here:

- the set-reset latch;
- the D latch.

### The set–reset latch (SR)

The SR latch is a circuit constructed from two cross-coupled NOR gates. The inputs are labelled S for Set and R for Reset.



The outputs  $Q$  and  $\bar{Q}$  depend not only on the inputs  $S$  and  $R$ , but also on the previous values of the outputs. Analysis of the circuit results in:

$$Q = \overline{R + Q'}$$

$$Q' = \overline{S + Q}$$

Considering each possible combination of the two input variables:

1. For  $S = 0, R = 0$ :  $Q = \bar{Q}'$ ;  $Q' = \bar{Q}$

Therefore,  $Q$  and  $Q'$  could be either 0 or 1, depending on the previous values of  $S$  and  $R$  (i.e. just before they were taken to 0).

2. For  $S = 0, R = 1$

$$Q = \overline{1 + Q'} = \bar{1} = 0$$

$$Q' = \overline{0 + Q} = \bar{Q} = 1$$

This input condition, known as RESET, forces  $Q = 0$  and  $Q' = 1$ .

3. For  $S = 1, R = 0$

$$Q' = \overline{1 + Q} = \bar{1} = 0$$

$$Q = \overline{0 + Q'} = \bar{Q'} = 1$$

This is the case opposite to 2, forcing the output to the set condition  $Q = 1$  &  $Q' = 0$ . This input condition is known as SET.

4. For  $S = 1, R = 1$

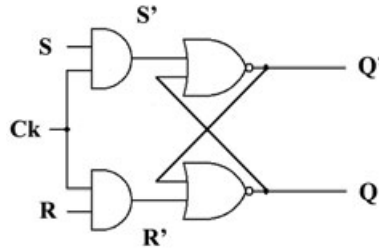
$$Q = \overline{1 + Q'} = \overline{1} = 0$$

$$Q' = \overline{1 + Q} = \overline{1} = 0$$

Although the outputs are stable for this input condition. If  $S$  and  $R$  are changed simultaneously to true, it is impossible to predict whether  $Q$  will remain at 0 or become 1, this is therefore an indeterminate state.

Input		Output
S	R	
1	0	Set state
0	0	
0	1	Reset state
0	0	
1	1	Indeterminate

The functionality of the S-R latch can be made synchronous by the introduction of a clock signal, to give a clocked SR flip-flop.



### Exercise

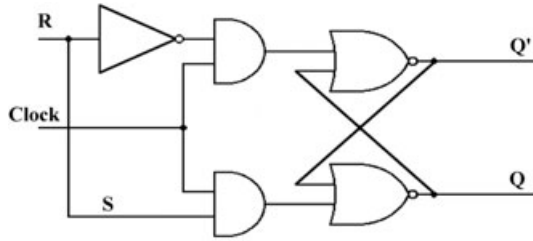
Produce a table showing the function of the the clocked S-R.

#### CRUCIAL CONCEPT

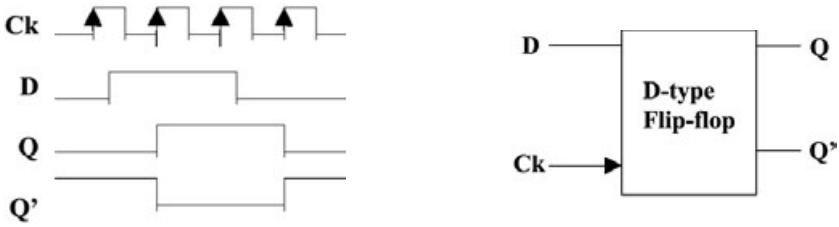
The **D-type latch**, described below, is a 1 bit memory device.

The D-type latch is a special case of clocked R-S latch with  $R = \bar{S}$  at all times. This new design eliminates therefore the indeterminate state ( $S = R = 1$ ), which represented the drawback of the S-R latch. The latch has two inputs: D-data and Ck-clock. The output  $Q$  follows the input  $D$ , as long as the clock is at logic level 1. The output  $Q$  is 'frozen' with the value of  $D$  at the instant the clock is taken at logic level 0, that is, the latch stores the value that was on the  $D$  input.

Ck	D	Next state of Q
0	X	No change
1	0	$Q = 0$ ; Reset state
1	1	$Q = 1$ ; Set state



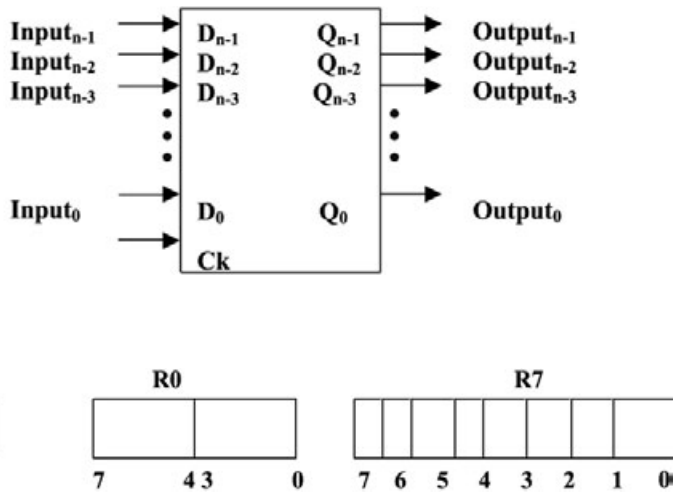
It is also possible to build flip-flops in which state changes are 'triggered' when the clock transits from 0 to 1, these are called edge-triggered devices. The output Q follows the input D, synchronising it with the transition of the clock from 0 to 1.



## Registers

Data processing in a computer system is usually done on fixed size binary 'words'. Hence, we need devices able to store a number of bits of information at a time. A register is a device in which a number of bits (binary digits) can be stored and retrieved.

A set of n flip-flops can be used to store n bits of information, to form a register. A schematic diagram of a n-bit register and three short-hand notations are shown below. The usual convention is to number the bits from 0 to (n-1).

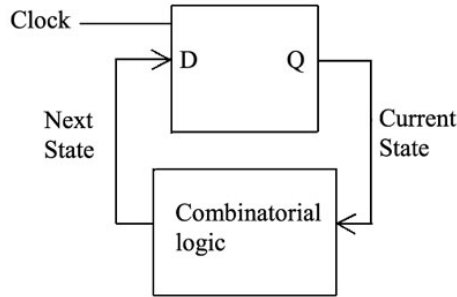


CRUCIAL CONCEPT

**Registers** are one of the fundamental building blocks of computer systems. They may contain data or control bits. One of the most common data manipulations involving registers is that of 'register transfer' through which the binary content of one register is copied into another register.

## Using registers in sequential logic

We now return to the counter example. We have already described the state diagram and the state table. We will use a D-type register to store the state of the machine. The next state will be generated using combinatorial logic from the current state and will form the inputs to the register, as shown below.



We add two more columns to the state table, for the inputs of the two flip-flops. The X input (the clock) has been implicitly considered here. The excitations ( $D_A$  and  $D_B$  respectively) are filled in by matching the required value for the input D of a flip-flop with the transition which has to take place. In this case, in order for the circuit to proceed from state  $AB = 00$  to state  $A + B + = 01$ ,  $D_A = 0 = A +$  and  $D_B = 1 = B +$ .

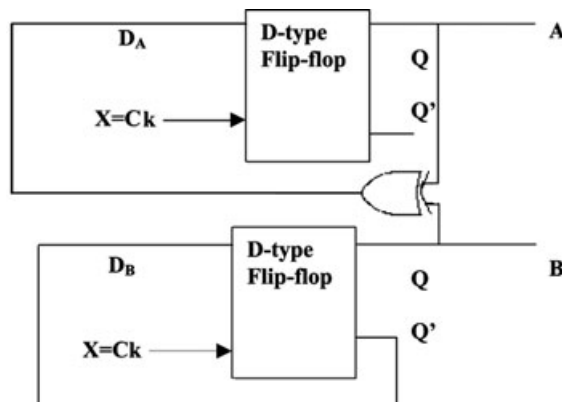
Present state		Next state		Flip-flop 1	Flip-flop 2
A	B	$A +$	$B +$	$D_A$	$D_B$
0	0	0	1	0	1
0	1	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0

The expressions for the 'next state'  $A + B + = D_A D_B$  can be obtained using sum-of-products procedures:

$$D_A = \bar{A} * B + A * \bar{B} = A \oplus B$$

$$D_B = \bar{A} * \bar{B} + A * \bar{B} = \bar{B}$$

The resulting circuit implementation is as follows.



## Section 5

# End of chapter assessment

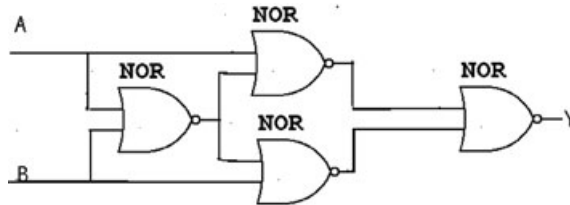
## Exercises

1. Simplify the following expressions using Boolean algebra identities and laws:

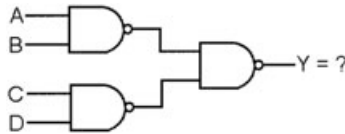
$$A * B * C * (A * B * \bar{C} + A * \bar{B} * C + \bar{A} * B * C)$$

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

2. The circuit below can be represented by a single gate. Which?



3. Determine the expression for Y in the figure below:

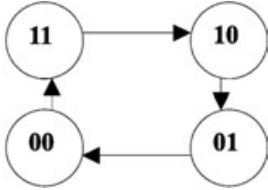


## Multiple-choice questions

- The denary number 172 is represented in binary as:
  - 10101100
  - 001110101
  - 10101010
  - 11001100
  - 11100111
- What is the main reason for the use of binary logic in computer systems?
  - robustness against noise
  - parallel processing of data
  - less wiring
  - smaller computers
  - convention
- What is the minimum number of bits required to represent all the characters on a keyboard that had nine keys?
  - 1
  - 2
  - 3
  - 4
  - 5

4. What is the minimum number of bits required to represent all the characters on a keyboard that has 22 keys?
- 1
  - 2
  - 3
  - 4
  - 5
5. The XOR function can be expressed in terms of AND, OR and NOT as:
- $(\bar{A} * \bar{B}) + (A * B)$
  - $(\bar{A} * B) + (A * \bar{B})$
  - $(\bar{A} * A) + (B * B)$
  - $(\bar{A} + B) * (A + \bar{B})$
  - $((A * \bar{B}) * A) + \bar{B}$
6. The result of regrouping the following expression,  
 $Y = ((A * B * C) * (A * \bar{B})) + ((A + B) + A) + (\bar{A} + (\bar{A} * B))$   
 is:
- $Y = A$
  - $Y = B$
  - $Y = 0$
  - $Y = 1$
  - $Y = \text{tri-state}$
7. A binary digit is called:
- byte
  - bit
  - word
  - and
  - or
8. A demultiplexer has:
- one input and multiple outputs
  - one output and multiple inputs
  - one input, one output
  - multiple inputs and multiple outputs
  - none of the above
9. A multiplexer has:
- one input and multiple outputs
  - one output and multiple inputs
  - one input, one output
  - multiple inputs and multiple outputs
  - none of the above
10. The number of state variables needed for the design of a FSM with six possible states is:
- 1
  - 2
  - 3
  - 4
  - 5

11. Consider the FSM in the figure below.



It represents the functionality of a:

- a) synchronous counter in the sequence 0-1-2-3
- b) asynchronous counter in the sequence 0-1-2-3
- c) synchronous counter in the sequence 3-2-1-0
- d) asynchronous counter in the sequence 3-2-1-0
- e) synchronous counter in the sequence 3-2-3-0

## Multiple-choice answers

1-a, 2-a, 3-d, 4-e, 5-b, 6-d, 7-d, 8-a, 9-b, 10-c, 11-c