Learning Aims

- Be familiar with drawing and interpreting logic gate circuit diagrams involving multiple gates
- Complete a truth table for a given logic gate circuit
- Write a Boolean expression for a given logic gate circuit
- Defining problems using Boolean logic

- Simplifying Boolean expressions using K maps
- Use Boolean identities and Laws to derive or simplify statements in Boolean algebra:

  - de Morgan's Laws
  - commutation
  - association
  - distribution
  - double negation
  - Absorption (not in spec)

Binary Logic

Boolean logic is **essential in computing**, particularly in:

- **Programming** (conditions like if A and B)
- **Search engines** (like using AND, OR, NOT in queries)
- **Digital circuits** (logic gates in a processor)

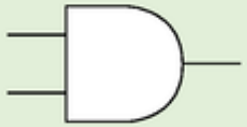
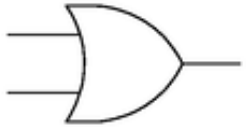
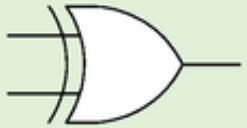
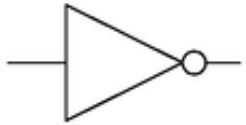
- There are a number of different logic gates that are each designed to perform a different operation in terms of output.

- **We will look at NOT, AND, OR and XOR gates.**

- Each of these gates may be represented by a **truth table** showing the output for each possible input or combination of inputs.

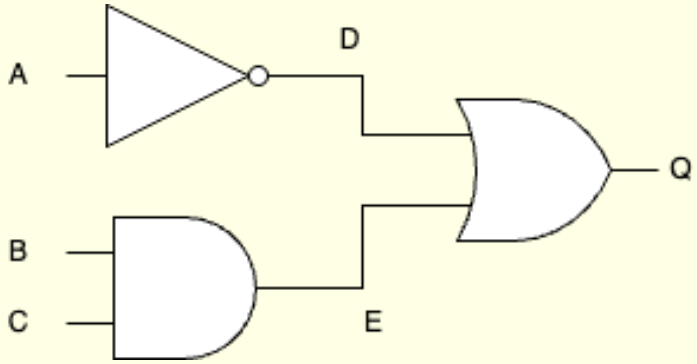
- Inputs are usually given algebraic letters such as A, B and C and output is usually represented by P or Q.

Logic Gates

Name	Logic	Logic gate	Notation	Alternative notation	Example	Truth table
Conjunction	AND		·	∗ ∧	A ∧ B A AND B	A B Output 0 0 0 0 1 0 1 0 0 1 1 1
Disjunction	OR		+	∨	A ∨ B A OR B	A B Output 0 0 0 0 1 1 1 0 1 1 1 1
Exclusive Disjunction	XOR		⊕	$\underline{\vee}$	A $\underline{\vee}$ B A XOR B	A B Output 0 0 0 0 1 1 1 0 1 1 1 0
Negation	NOT		−	¬ ~ !	¬A NOT A	A Output 0 1 1 0
Equivalence	The same as		≡	↔	A ≡ B A is the same as B	A Output 0 0 1 1

Creating logic gate circuits

- Multiple logic gates can be connected to produce an output based on multiple inputs.



This circuit can be represented by the expression Q = ¬A ∨ (B ∧ C)

or alternatively as Q = (NOT A) OR (B AND C)

Input A	Input B	Input C	D = ¬ A	E = B ∧ C	Output Q = D ∨ E
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

How to write Boolean expression represented in a logic diagram

Write the Boolean expression represented by the logic diagram below, using AND, OR and NOT instead of symbols. Then write the same expression using symbols.

First write (A AND B).

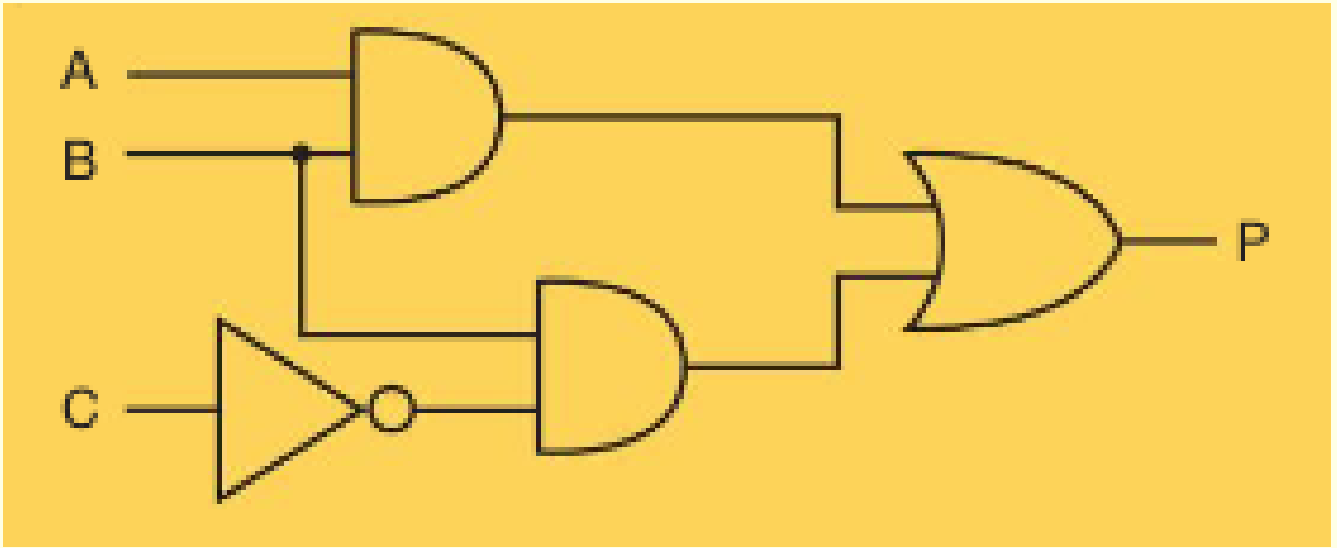
Then write (B AND NOT C)

These are the inputs to the OR gate

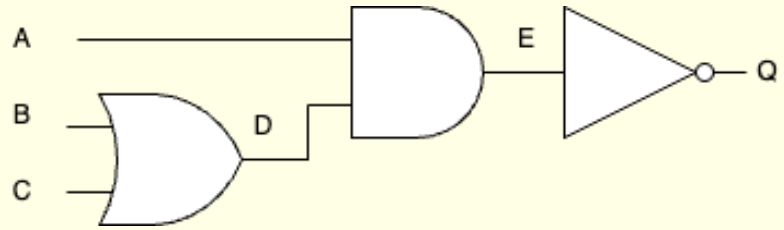
So the expression is:

P = (A AND B) OR (B AND NOT C)

$P = (A \wedge B) \vee (B \wedge \neg C)$



Q1 Write the expression and complete the truth table for the following circuit



Q = NOT (B OR C) AND A)

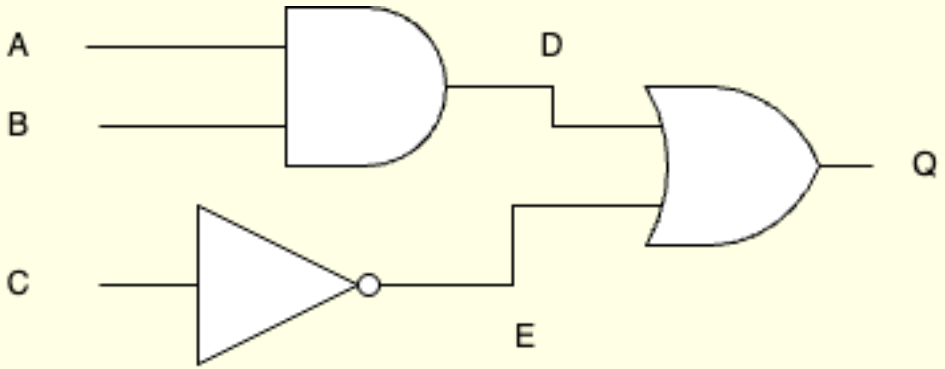
Q = ¬(A ∧ (B ∨ C))

Input A	Input B	Input C	D = B ∨ C	E = A ∧ D	Q = ¬ E
0	0	0	0	0	1
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Q2

Q = (A ^ B) v ¬ C

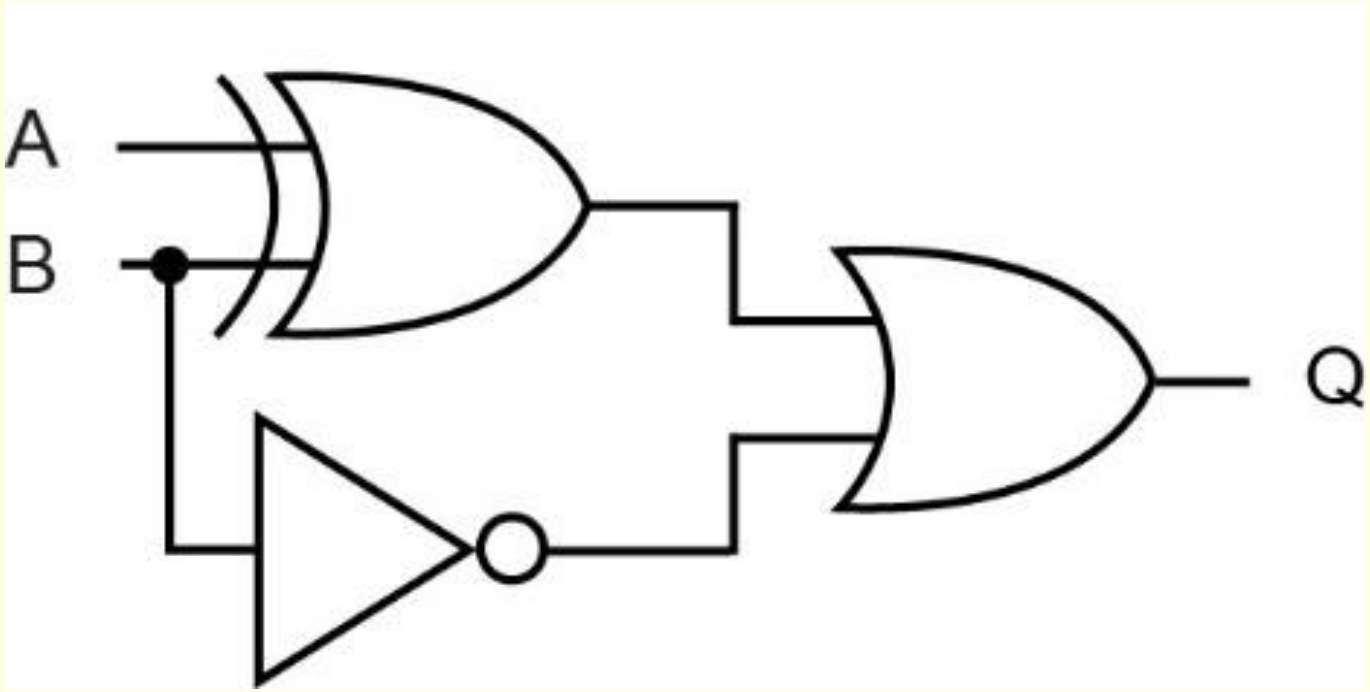
Draw the logic circuit



A	B	C	(A ^ B)	¬ C	Q
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

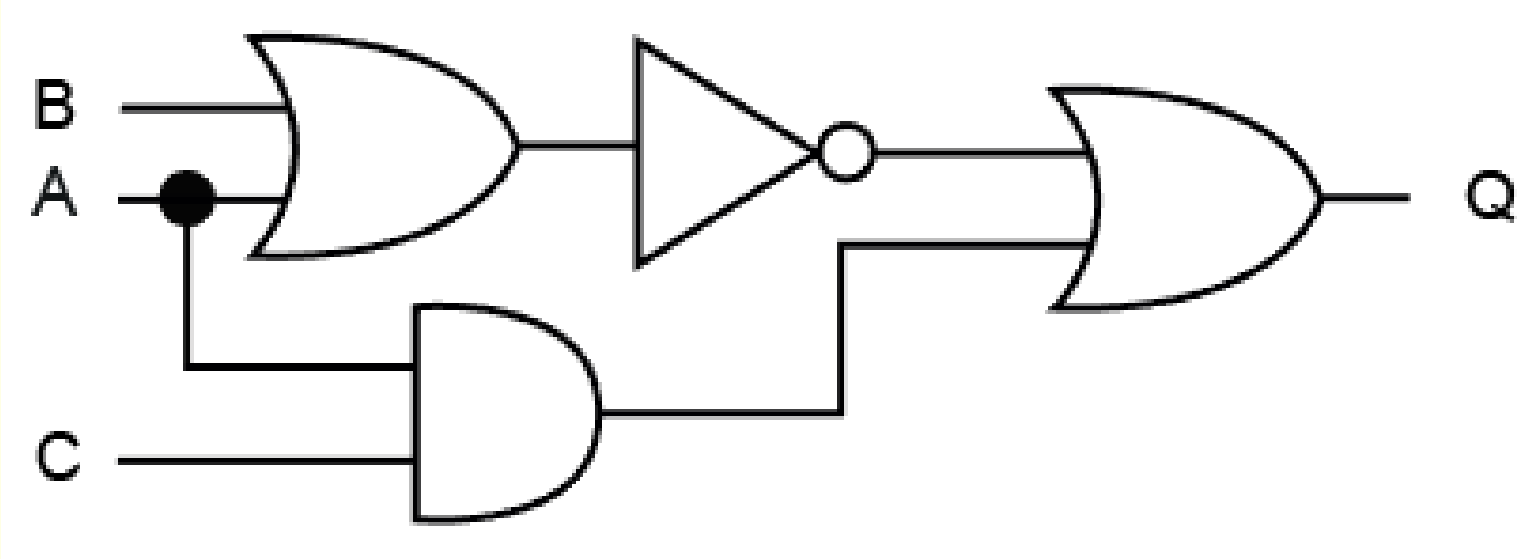
Draw logic circuit for the following Boolean expression:

Q = **A V B v ¬ C**

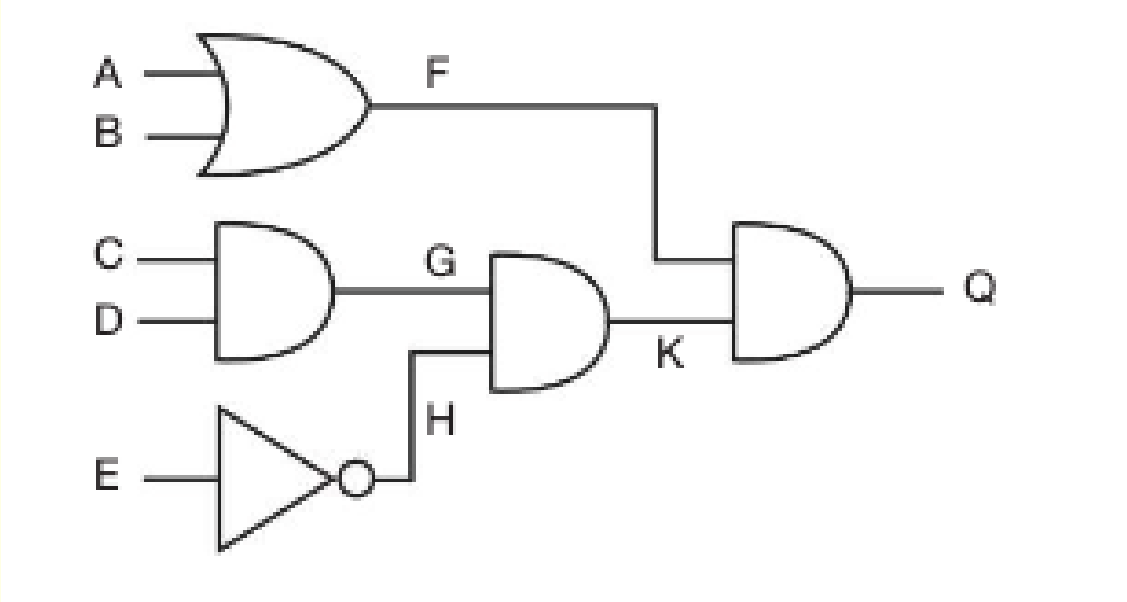


Draw logic circuit for the following Boolean expression:

Q = **¬(A v B) v (A ^ C)**



Write the Boolean Expression for this logic circuit [3]



$Q = (A \vee B) \wedge ((C \wedge D) \wedge \neg E)$

Defining problems using Boolean logic

- We can define problems in terms of Boolean logic.
- A boiler has two sensors, a pressure sensor and a temperature sensor. If either the temperature (T) or the pressure (P) is too high, a valve (V) will close.
- This can be expressed as V = T v P or alternatively as V = T OR P
- The table representing these conditions could be drawn as follows:

Input	Binary value	Condition
T	1	Temperature too high
	0	Temperature not too high
P	1	Pressure too high
	0	Pressure not too high

Worked Example

A chemical process has a sensor to detect a dangerous situation, in which case it sounds an alarm (A).

The alarm is sounded if:

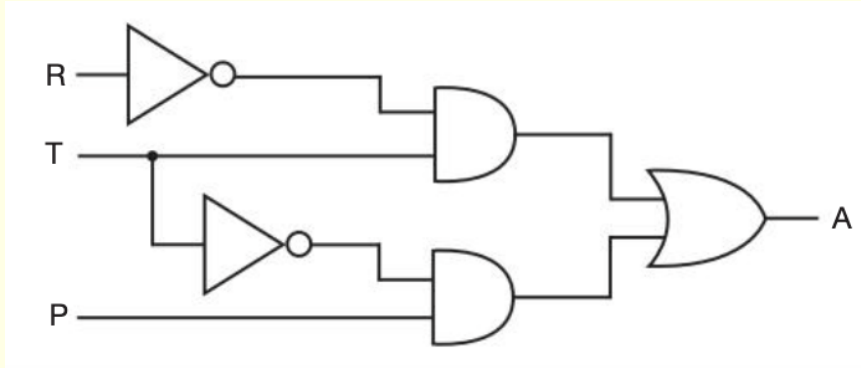
either temperature >= 100°C AND rotator is OFF

or

PH > 6 AND temperature < 100°C

A table can be drawn to represent these conditions as Boolean values.

Input	Binary value	Condition
T	1	Temperature >= 100°C
	0	Temperature < 100°C
R	1	Rotator ON
	0	Rotator OFF
P	1	PH > 6
	0	PH <= 6



The conditions can be written as

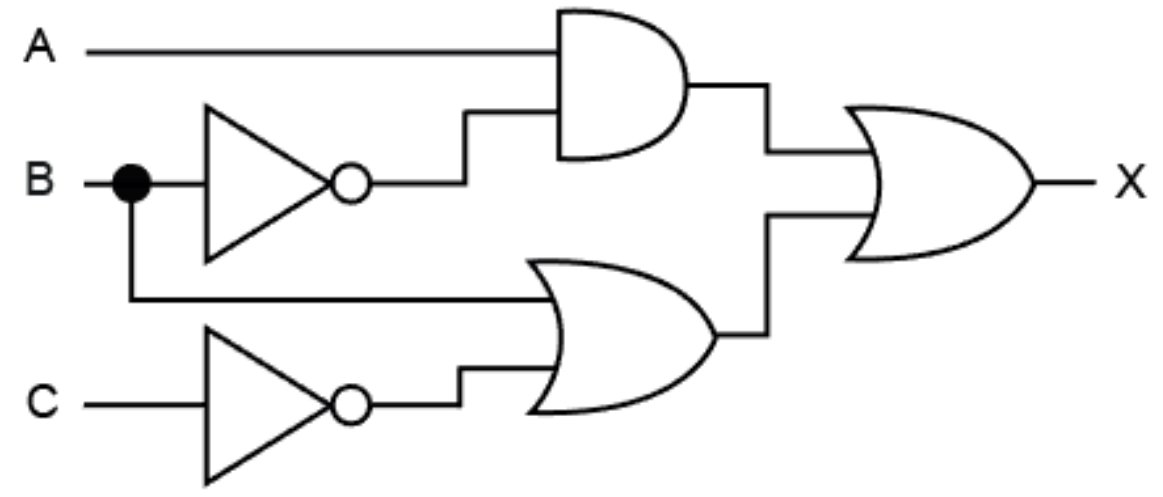
$A = (T \wedge \neg R) \vee (P \wedge \neg T)$ or alternatively as A = (T AND NOT R) OR (P AND NOT T)

Truth Table

Input R	Input T	Input P	$X = T \wedge \neg R$	$Y = P \wedge \neg T$	$A = X \vee Y$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	0	0	0
1	1	1	0	0	0

Question

Three sensors A, B and C are used to monitor a process. A signal X is output from the circuit.
X has the value 1 if either of the following conditions are met:
Sensor A outputs 1 AND sensor B outputs 0
Sensor B outputs 1 OR sensor C outputs 0
Draw a logic circuit to represent these conditions. [5]



Karnaugh maps
(K Maps)

Learning Objectives

- Simplify Boolean expressions using Karnaugh maps

A Karnaugh map provides an alternative way of simplifying Boolean expressions which is often easier than using Boolean algebra for those involving up to three or four variables. It is similar to a truth table and allows us to easily detect groupings of expressions with common factors.

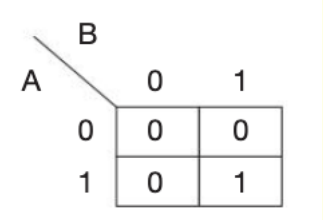
The two-variable problem

- The figure below shows the correspondence between a truth table and a K map.

The values inside the squares are copied from the output column of the truth table, so there is one square in the Karnaugh map for every row in the truth table. Suppose we have the following truth table:

Input A	Input B	Output Q
0	0	0
0	1	0
1	0	0
1	1	1

The corresponding Karnaugh map is:



For example, when A = 0 and B = 0, the output is 0. When A = 1 and B = 1, the output is 1.

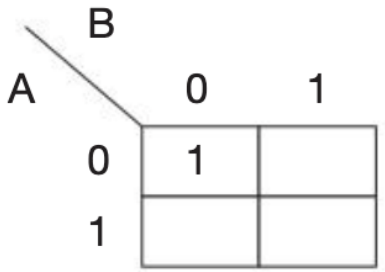
Worked Example

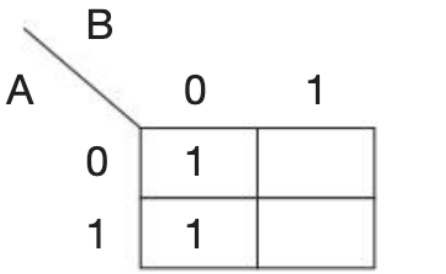
- Use a Karnaugh map to simplify the expression $Q = \neg A \wedge \neg B \vee A \wedge \neg B \vee \neg A \wedge B$

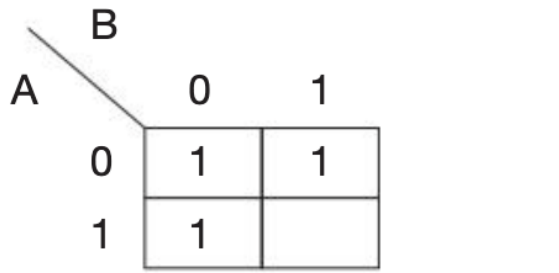
1. Group the expression into three sub-expressions separated by v

$Q = (\neg A \wedge \neg B) \vee (A \wedge \neg B) \vee (\neg A \wedge B)$

1. Draw a blank Karnaugh map and fill in a 1 for the first sub-expression **¬A ^ ¬B.**
2. Then insert a 1 for the second sub-expression **A ^ ¬B.**
3. Finally add a 1 for the sub-expression **¬A ^ B**



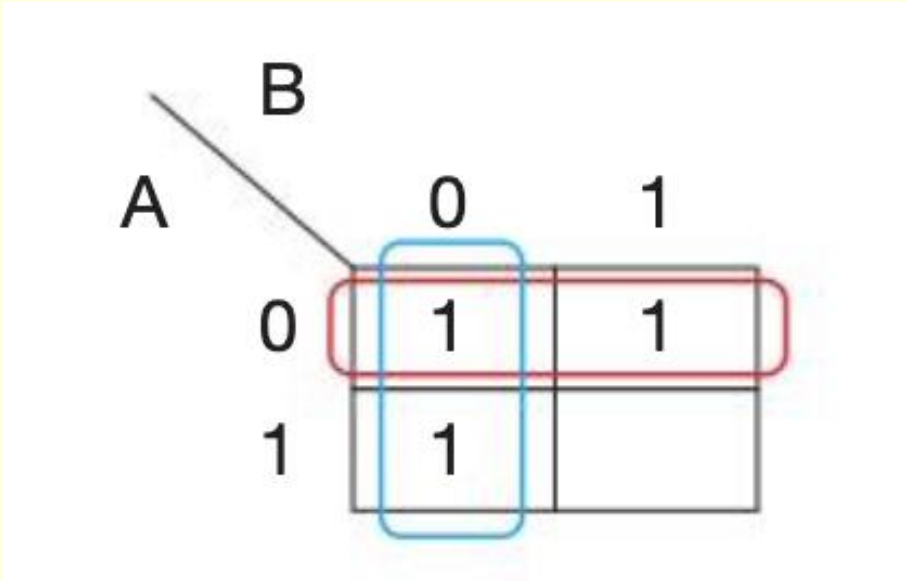




Worked Example

- Now make groupings of 1, 2, or 4 ones, which can be overlapping.
- Each grouping should be as large as possible – in this case, the two groupings each consist of two squares.
- The pink group represents NOT A, and the blue group represents NOT B.
- Therefore the whole expression
- represents **Q = NOT A OR NOT B**, or in alternative notation **¬A v ¬B.**
- This is the simplification of the expression

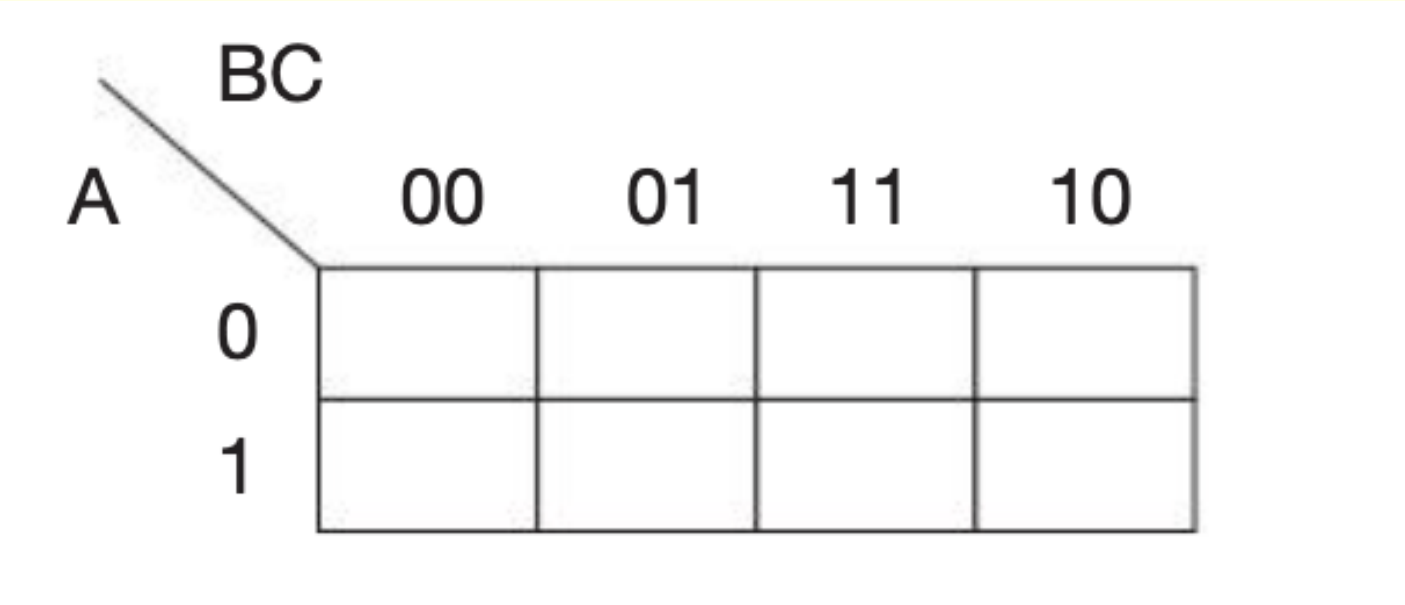
$$Q = \neg A \wedge \neg B \vee A \wedge \neg B \vee \neg A \wedge B$$



The three-variable problem

- With three variables, each column can represent a combination of two variables.
- Represent the expression

$$\neg A \vee \neg B \vee A \wedge B \wedge \neg C$$



The order of terms along the top is not random: they are arranged so that each subsequent term reflects a change in only one variable.
They are not in numerical sequence of 00, 01, 10, 11.

Worked Example

- The choice of whether to put A on its own, and group B and C together, or choose a different pair, and put for example C as the column heading and AB as the row heading, is not important, and will produce the same groupings.
- First, divide the expression into sub-expressions, bracketing between the v (OR)

$(\neg A) \vee (\neg B) \vee (A \wedge B \wedge \neg C)$

As before, we can now start filling in the table one step at a time, representing each sub-expression in turn.

A \ BC	00	01	11	10
0	1	1	1	1
1				

$(\neg A)$

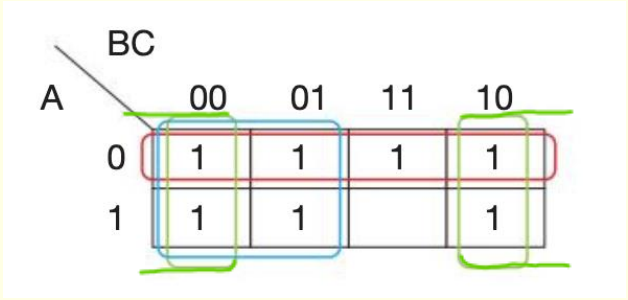
A \ BC	00	01	11	10
0	1	1	1	1
1	1	1		

$(\neg A) \vee (\neg B)$

A \ BC	00	01	11	10
0	1	1	1	1
1	1	1		1

$(\neg A) \vee (\neg B) \vee (A \wedge B \wedge \neg C)$

The next step is to identify the largest groups of 1, 2 4 or 8 ones.



Notice that the green group has "wrapped around" and is counted as one group representing ¬C.

These three groups together represent **¬A v ¬B v ¬C.**

The four-variable problem

With four variables, each row or column represents a combination of two variables.
Represent the expression and hence simplify the expression.

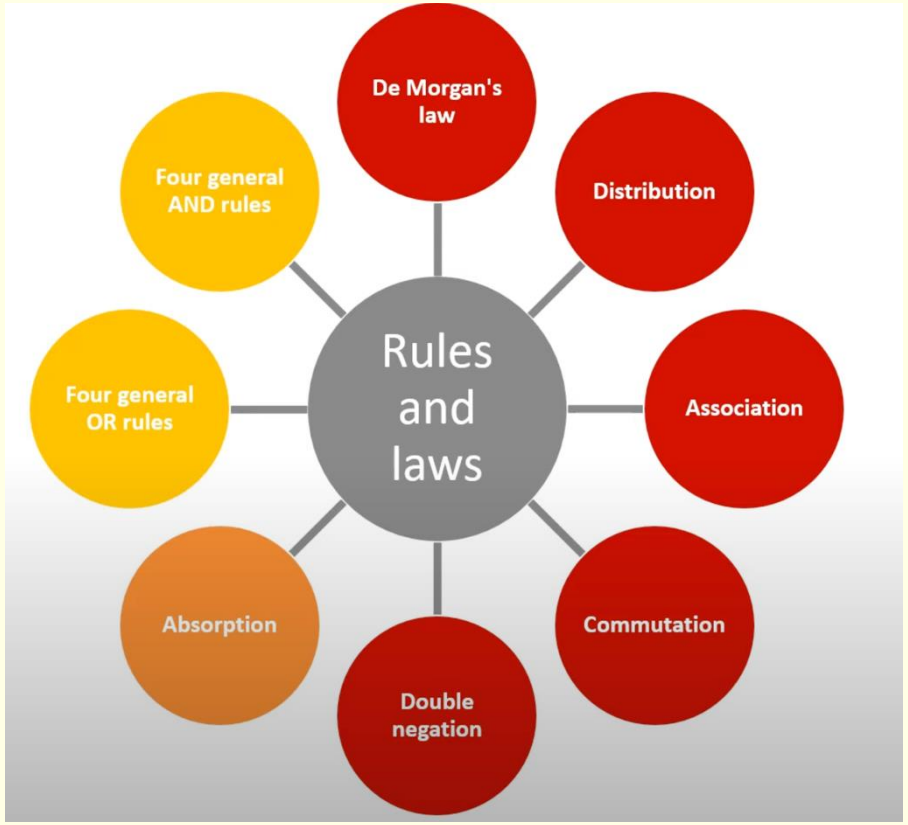
$$A \vee (A \wedge \neg B \wedge C \wedge D)$$

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

This simplifies to A.

Learning Objectives

- Use the following rules to derive or simplify statements in Boolean algebra:
  - De Morgan's Laws
  - Commutation
  - Association
  - Distribution
  - Double negation
  - Absorption
- Write a Boolean expression for a given logic gate circuit, and vice versa



Boolean Identities and Laws

X – is a variable (0 or 1)

General AND rules	General OR rules
X AND 0 = 0	X OR 0 = X
X AND 1 = X	X OR 1 = 1
X AND X = X	X OR X = X
NOT X AND X = 0	NOT X OR X = 1

De Morgan's Laws - Breaking a negation and changing the operator between two inputs.

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

Distribution - Expanding brackets
Reverse Distribution – Taking out the common factor from multiple terms and expressing them in bracketed form.
This is also known as **factoring** or **factoring out the common factor**.

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge (A \wedge C)$$
$$A \vee (B \vee C) \equiv (A \vee B) \vee (A \vee C)$$

Commutation – order not important

$$A \vee B \equiv B \vee A$$
$$A \wedge B \equiv B \wedge A$$

Double Negation

$$\neg\neg A \equiv A$$

Association – allows for the removal of brackets and regrouping of variables

$$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C) \equiv A \wedge B \wedge C$$
$$(A \vee B) \vee C \equiv A \vee (B \vee C) \equiv A \vee B \vee C$$

Absorption This means that if a is true, the entire expression is true regardless of b.

$$a \wedge (a \vee b) = a \vee (a \wedge b) = a$$

Distribution

What does it mean?

- This rule allows us to multiply or factor out an expression.

In Boolean algebra

The **OR** distribution rule:

A ∧ (B V C) ≡ (A ∧ B) V (A ∧ C)

A **AND** (B **OR** C) *is the same as*
(A **AND** B) **OR** (A **AND** C)

The **AND** distribution rule:

A V (B ∧ C) ≡ (A V B) ∧ (A V C)

A **OR** (B **AND** C) *is the same as*
(A **OR** B) **AND** (A **OR** C)

Real-life analogy

"You can choose one main course and either a starter or dessert."
is the same as,
"You can choose one main and one starter or one main and one dessert."

Simplify (A v B)^ (A v C)

How to answer this question:

Step one - Distribution

This is a bit like multiplying out the brackets in an expression in regular maths. Think of OR being like ADD and AND being like MULTIPLY.

(A v B) ^ (A v C)

becomes

(A ^ A) v (B ^ A) v (A ^ C) v (B ^ C)

Step two - General rules

Since (A ^ A) is just A we can replace this term in the expression with a simpler one.

(A ^ A) v (B ^ A) v (A ^ C) v (B ^ C)

becomes

A v (B ^ A) v (A ^ C) v (B ^ C)

Step three - Commutation

This means the order of the logical operators does not matter so can change (B ^ A) into (A ^ B).

A v (B ^ A) v (A ^ C) v (B ^ C)

becomes

A v (A ^ B) v (A ^ C) v (B ^ C)

Step four - Absorption

This rule says that A AND (A OR B) = A.

A v (A ^ B) v (A ^ C) v (B ^ C)

becomes

A v (A ^ C) v (B ^ C)

Step five - Another absorption

Again this rule says that A AND (A OR C) = A so

A v (A ^ C) v (B ^ C)

becomes

A v (B ^ C)

Example answer that gets full marks:

A v (B ^ C)

OR Just Reverse Distrubution

Adders and D-type flip-flops

Learning Objectives

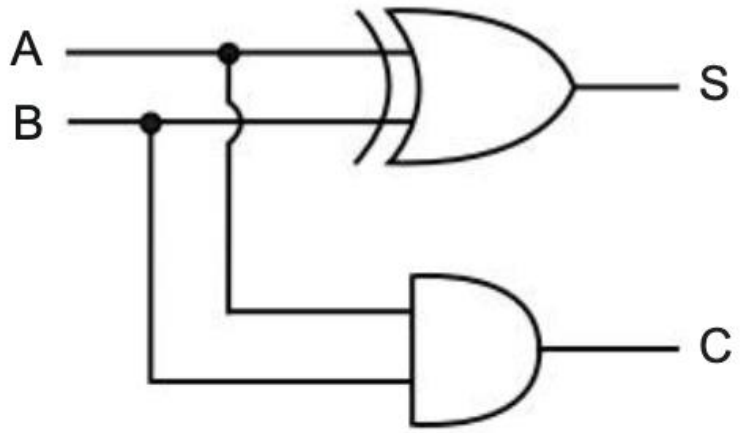
- Recognise and trace the logic of the circuits of a half adder and a full adder
- Construct the circuit for a flip-flop
- Be familiar with the use of the edge-triggered D-type flip-flop as a memory unit

Performing calculations using gates

- With the right combination of gates, it is possible to output the result of a binary addition or subtraction including the value of any carry bit as a second output.

Half adders

A half adder can take an input of two bits and give a two-bit output as the correct result of an addition of the two inputs.



A		B		S	C
0	+	0	=	0	0
0	+	1	=	1	0
1	+	0	=	1	0
1	+	1	=	0	1

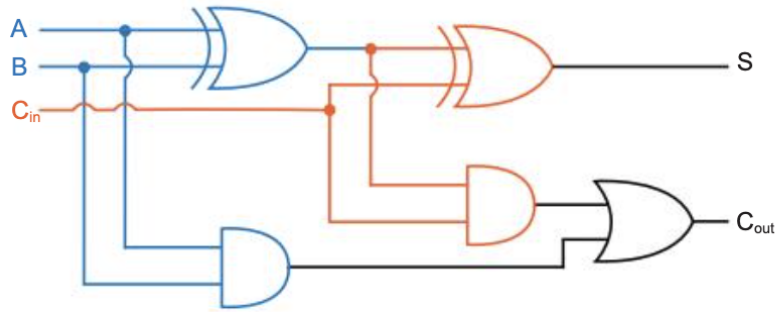
This is shown by the diagram above and represented by the truth table where S represents the sum and C represents the carry bit. S can be given as

$$S = A \veebar B, \text{ and } C \text{ as } C = A \wedge B.$$

Although a flip-flop can output the value of a carry bit, it only has two inputs so it cannot use the carry from a previous addition as a third input to a subsequent addition in order to add n-bit numbers.

Full adders

- A full adder combines two half adders to add three bits together including the two inputs A and B, and a carry bit C. The logic gate circuit below illustrates how two half adders have been connected with an additional OR gate to output the carry bit.



A		B		C_{in}		S	C_{out}
0	+	0	+	0	=	0	0
0	+	0	+	1	=	1	0
0	+	1	+	0	=	1	0
0	+	1	+	1	=	0	1
1	+	0	+	0	=	1	0
1	+	0	+	1	=	0	1
1	+	1	+	0	=	0	1
1	+	1	+	1	=	1	1

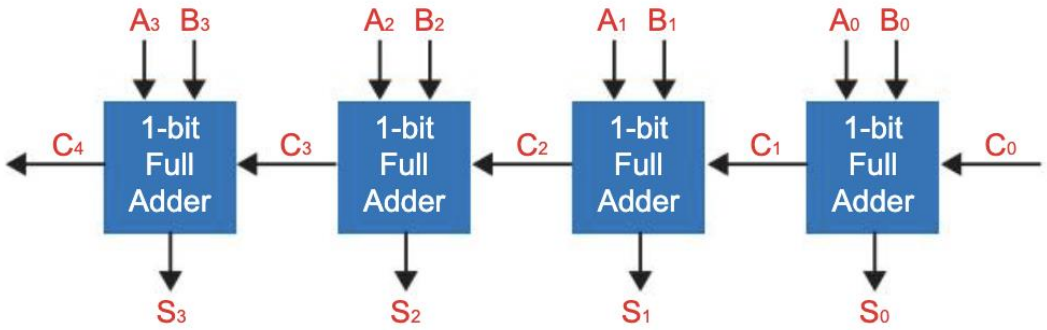
Now the Boolean logic becomes

$$S = A \veebar B \veebar C_{in}, \text{ and } C_{out} = (A \wedge B) \vee (C_{in} \wedge (A \veebar B))$$

Concatenating full adders

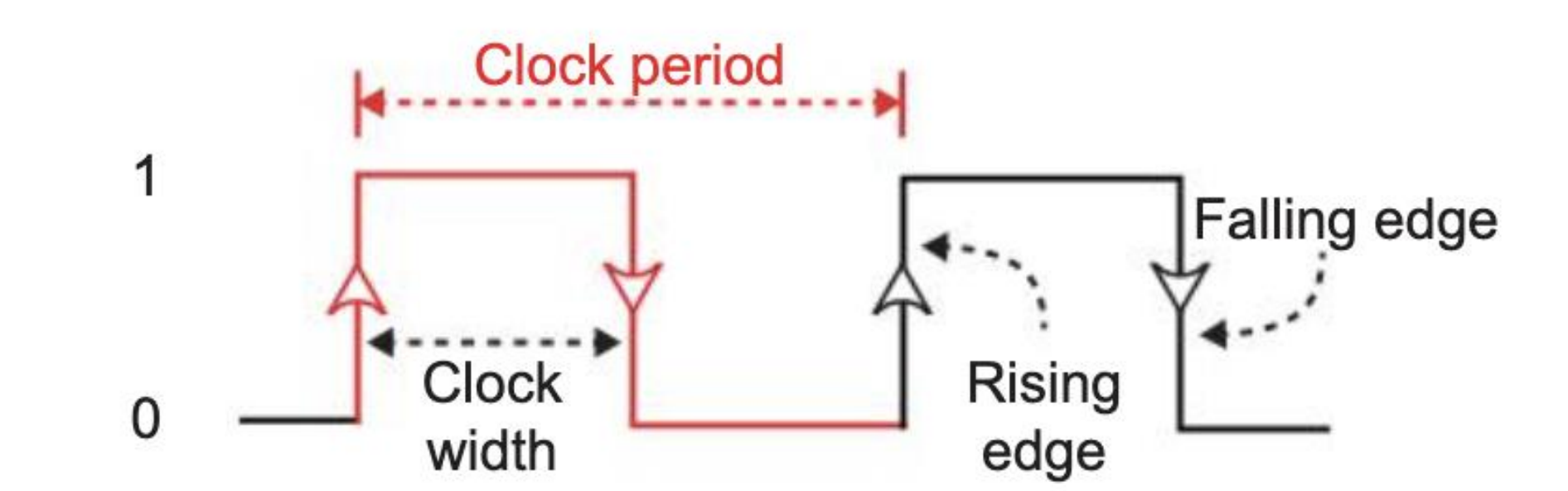
- Multiple full adders can be connected together.
- Using this construct, n full adders can be connected together in order to input the carry bit into a subsequent adder along with two new inputs to create a concatenated adder capable of adding a binary number of n bits.
- The four-bit adder is an example of a standard component that can be used in many applications involving arithmetic operations.

	0^{C_3}	1^{C_2}	1^{C_1}		
	0^{A_3}	0^{A_2}	1^{A_1}	1^{A_0}	3
+	1^{B_3}	0^{B_2}	1^{B_1}	1^{B_0}	11
=	1^{S_3}	1^{S_2}	1^{S_1}	0^{S_0}	14

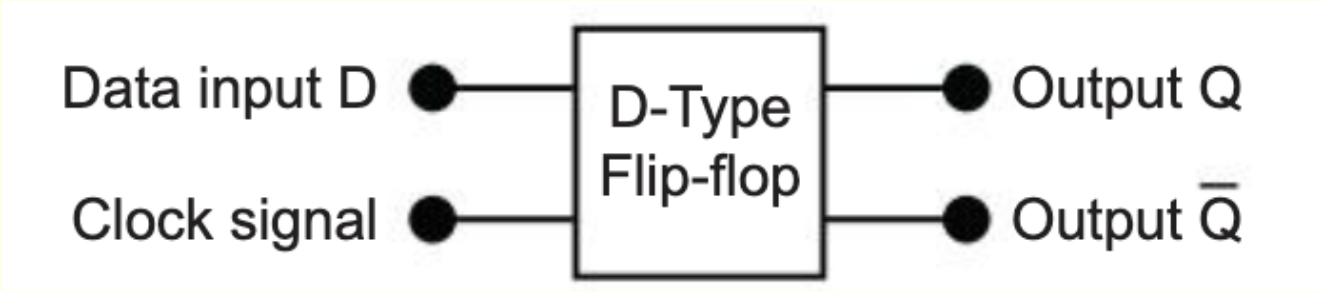


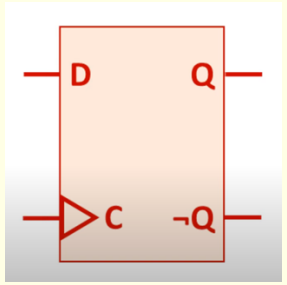
D-type flip-flops

- A flip flop is an elemental **sequential logic circuit** that can store one bit and flip between two states, 0 and 1.
- It has two inputs, a control input labelled D and a clock signal.
- The **clock** or **oscillator** is another type of sequential circuit that changes state at regular time intervals.
- Clocks are needed to synchronise the change of state of flip flop circuits.



- The **D-type flip-flop** (D stands for Data or Delay) is a positive **edge-triggered flip-flop**, meaning that it can only change the output value from 1 to 0 or vice versa when the clock is at a rising or positive edge, i.e. at the beginning of a clock period.





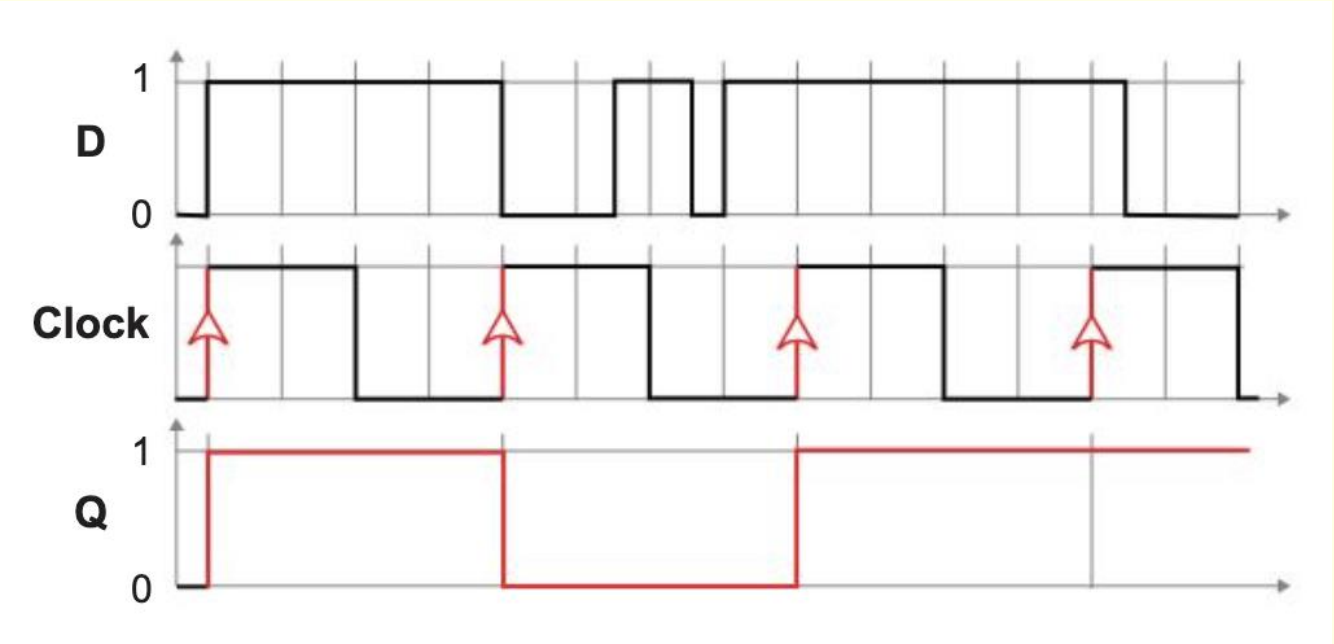
When the clock is not at a positive edge, the input value is held and does not change.

The flip-flop circuit is important because it can be used as a memory cell to store the state of a bit.

Output Q only takes on a new value if the value at D has changed at the point of a clock pulse. This means that the clock pulse will freeze or 'store' the input value at D until the next clock pulse. If D remains the same on the next clock pulse, the flip-flop will hold the same value.

The use of a D type flip flop as a memory unit

- A flip flop comprises several NAND (or AND and OR) gates and is effectively 1-bit memory.
- To store eight bits, eight flip-flops are required.
- **Register memories** are constructed by connecting a series of flip-flops in a row and are typically used for the intermediate storage needed during arithmetic operations.
- Static RAM is also created using D-type flip-flops.
- Imagine trying to assemble 16GB of memory in this way!
- The graph below illustrates how the output Q only changes to match the input D in response to the rising edge on the clock signal. Q therefore delays, or 'stores' the value of D by up to one clock cycle.



Bitwise manipulation and masks

Learning Aims

- Perform logical, arithmetic and circular shifts on binary data
- Perform bitwise operations AND, OR and XOR
- Use masks to manipulate bits

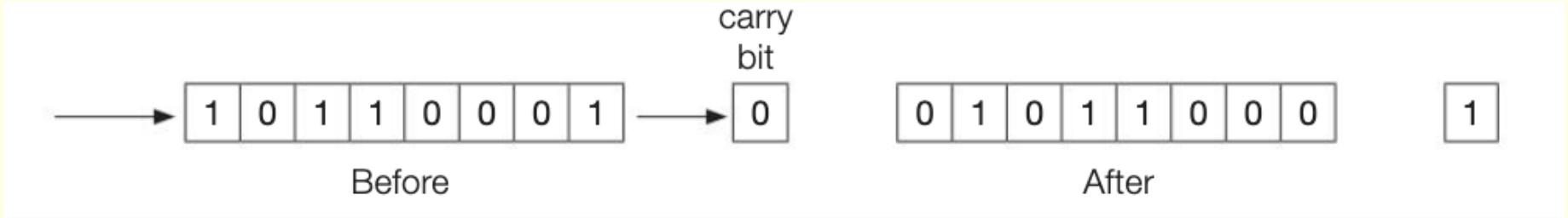
XOR

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Logical shift instructions

- All the bits move right or left.

- A logical **shift right** causes the least significant bit (lsb) to be shifted into the carry bit, and a zero moves into the most significant bit (msb) to occupy the vacated space.



It is useful for examining the least significant bit of a number.
After the operation, the carry bit can be tested and a conditional branch executed.

Logical shift instructions

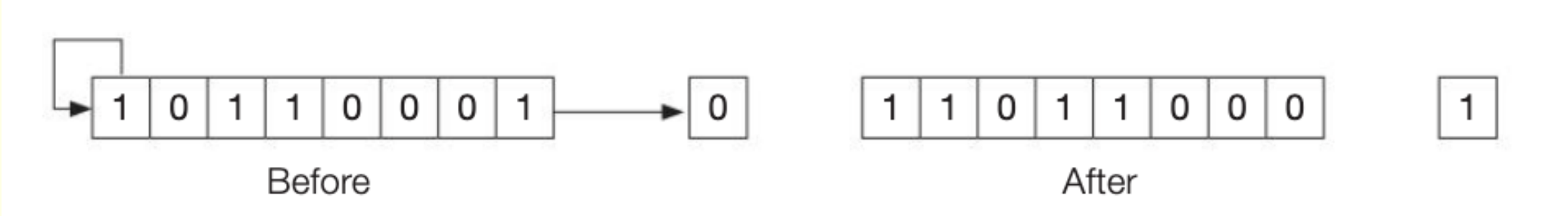
- A logical **shift left** works in the same way, but the bits move left.
- The most significant bit (msb) moves into the carry bit and a zero moves into the lsb.
- You can visualise the carry bit as being on the left of the byte.

Q1 Shift the binary pattern 0100 0111 right twice and then left once.
What are the contents of the byte and the carry bit after these shifts?

Arithmetic shift instructions

- An arithmetic shift is similar, but it takes into account the sign bit, which always remains the same.

- **Shifting right** has the effect of **dividing by 2.**
- If the sign bit is 1, 1 is moved in from the left instead of 0.



Q2 Convert the number -16 to binary, and divide by 8 using arithmetic shifts.

Arithmetic shift instructions

Shifting left multiplies by 2.

The shift bypasses the sign bit, leaving the msb the same whatever the value of the other bits.

However, this may result in arithmetic overflow, as shown below.

Q3 Convert the number 14 to binary and then multiply it by 4 using arithmetic shifts. Convert the result back to denary.

Multiplying two numbers using arithmetic shifts

- Using a combination of shifts and addition, two binary numbers may be multiplied together.

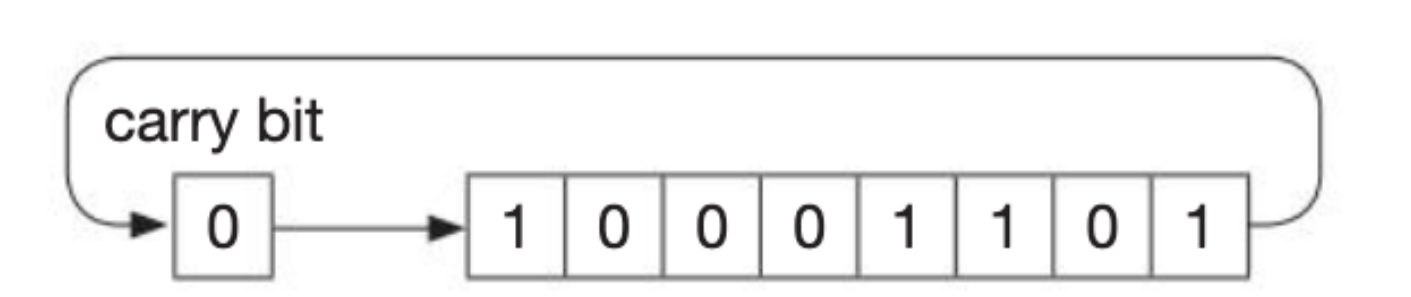
- Multiply 9 by 5 using shifts and addition:

Multiply 9 x 1	0000 1001	
Multiply 9 by 4 with 2 left shifts:	0010 0100	
Add together:	0010 1101	= 45

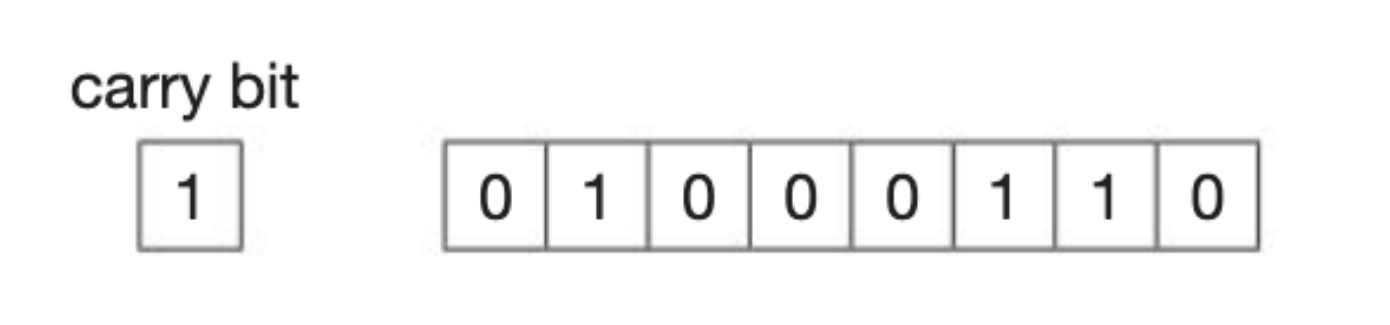
Q4: Multiply 12 by 6 using shifts and addition.

Circular shift instructions

- A rotate or circular shift is useful for performing shifts in multiple bytes. In a circular shift right, the value in the least significant bit (lsb) is moved into the carry bit, and the carry bit is moved into the most significant bit (msb).

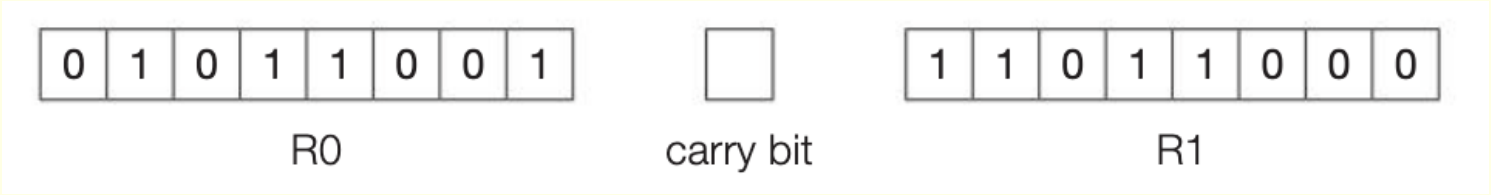


A circular shift right of the bit pattern shown above will result in the following:



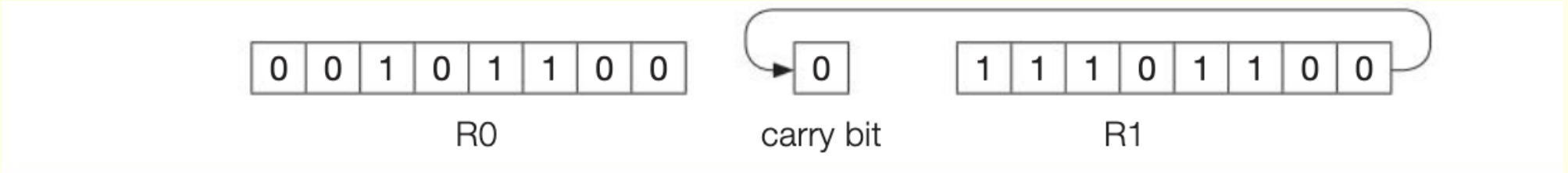
Worked Example

Assume that R0 and R1, shown below, are two 8-bit registers being used as a double register to hold a 16-bit binary integer, with R0 holding the high half of the number. Show how a combination of shift instructions may be used to divide the 16-bit integer by 2.



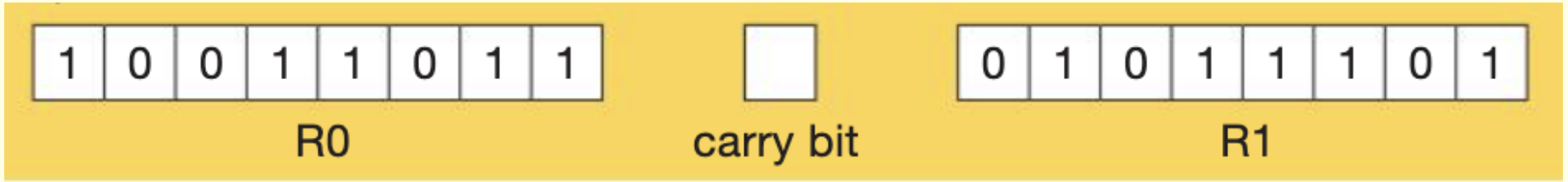
Answer: First perform an arithmetic shift right on R0. 1 is shifted into the carry bit.

Then perform a circular shift on R1. This places the carry bit into the msb of R1, and the carry bit is replaced with the lsb of R1.



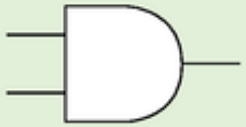
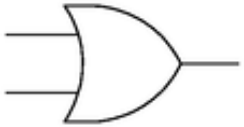
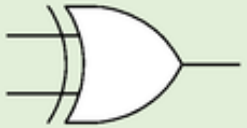
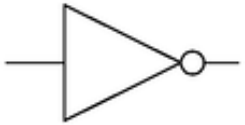
Q5

A 16-bit integer is held in R0 and R1 which are being used as a double register. Show the effect on the registers of doing an arithmetic shift right of 1 place in R0, followed by a circular shift right of 1 place in R1.



R0: 0100 1101 carry: 1 R1: 10101110

Recap Logic Gates

Name	Logic	Logic gate	Notation	Alternative notation	Example	Truth table
Conjunction	**AND**		·	$*$ $\wedge$	$A \wedge B$ A AND B	A B Output 0 0 0 0 1 0 1 0 0 1 1 1
Disjunction	**OR**		+	$\vee$	$A \vee B$ A OR B	A B Output 0 0 0 0 1 1 1 0 1 1 1 1
Exclusive Disjunction	**XOR**		$\oplus$	$\underline{\vee}$	$A \underline{\vee} B$ A XOR B	A B Output 0 0 0 0 1 1 1 0 1 1 1 0
Negation	**NOT**		−	$\neg$ $\sim$!	$\neg A$ NOT A	A Output 0 1 1 0
Equivalence	**The same as**		$\equiv$	$\leftrightarrow$	$A \equiv B$ A is the same as B	A Output 0 0 1 1

Masks

- The OR function may be used to set selected bits to 1 without affecting the other bits.

- A system has 8 lights that can be turned ON (output 1) of OFF (output 0), controlled by an 8-bit binary code.
- At present, lights 1 to 4 are ON, lights 5 to 8 are OFF.
- Lights 5 and 6 are to be turned ON.

Light number	1 2 3 4 5 6 7 8
Present state	1 1 1 1 0 0 0 0
OR with	0 0 0 0 1 1 0 0
Result	1 1 1 1 1 1 0 0

The AND function may be used to mask particular bits, by setting them to zero.

Worked Example

- The ASCII bit pattern for the number "5" is 0011 0101. Convert this to a pure binary number using a mask.
- We need to mask out the first four bits. This can be done with an AND operation.

ASCII "5"	0 0 1 1 0 1 0 1
AND with	0 0 0 0 1 1 1 1
Result	0 0 0 0 0 1 0 1 = 5 in binary

The XOR function may be used to invert chosen bits.

Worked Example

- Convert an uppercase letter represented in ASCII to its lowercase equivalent.
- The letter “C”, for example, is 0100 0011 in ASCII. The lowercase letter “c” is 0110 0011. We want to change the third bit (counting from the left) from 0 to 1.

ASCII “C”	0 1 0 0 0 0 1 1
XOR with	0 0 1 0 0 0 0 0
Result	0 1 1 0 0 0 1 1 = “c”