OCR A Level Computer Science

Your notes

4.3 Boolean Algebra

Contents

Boolean Logic

Your notes

Logic Gates

What is Boolean Logic?

- Boolean logic is used in computer science and electronics to make **logical decisions**
- Boolean operators are either **TRUE** or **FALSE**, often represented as **1** or **0**
- Inputs and outputs are given **letters to represent them**
- To define Boolean logic we use **special symbols** to make **writing expressions** much **easier**

Combination of Boolean operators

- Can be **combined** to form more **complex** expressions
- Use **parentheses** to clarify the **order** of operations
  Example: NOT (TRUE AND FALSE) = TRUE

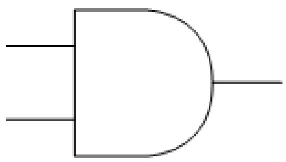
Evaluating Boolean expressions

- There is a specific sequence for evaluating expressions with **multiple operators** just like in normal maths where **BIDMAS** applies
- **Brackets come first** then **NOT** then **AND** then **OR**
- Using Brackets can **alter the standard order** of operations
- Expressions within **parentheses** are **evaluated first**, following the same **NOT**, **AND**, **OR** precedence inside the parentheses
- **Example:** NOT (TRUE AND FALSE) equals NOT FALSE, which equals TRUE

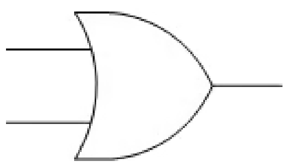
Logic Gates

- Logic gates are a visual way of representing a Boolean expression
- The logic gates covered in this course are:
  - **Conjunction (AND)**
  - **Disjunction (OR)**
  - **Negation (NOT)**
  - **Exclusive disjunction (XOR)**

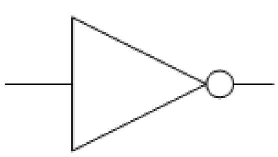
Conjunction (AND)

Operation	Circuit symbol	Notes
A ∧ B A . B		Returns TRUE only if **both** inputs are TRUE **TRUE** AND **TRUE** = **TRUE** Otherwise = FALSE Next **highest precedence** after NOT **Executes before OR operations**

Disjunction (OR)

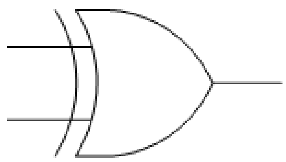
Operation	Circuit symbol	Explanation
A∨B A+B		Returns TRUE if **either** input is TRUE **TRUE** OR **FALSE** = **TRUE** **FALSE** OR **FALSE** = **FALSE** **Lowest precedence** in Boolean expressions Executes **after NOT and AND** operations

Negation (NOT)

Symbol	Circuit symbol	Notes
¬A $\overline{A}$		**Inverts** the input value NOT **TRUE** = **FALSE** NOT **FALSE** = **TRUE** **Highest precedence** in Boolean expressions Executes **before AND and OR** operations

Exclusive Disjunction (XOR)

Your notes

<table>
<tr><th>Operation</th><th>Circuit symbol</th><th>Notes</th></tr>
<tr><td>A⊻B
A
⊕
B</td><td></td><td>Outputs TRUE if the inputs are different

Outputs FALSE if they are the same</td></tr>
</table>

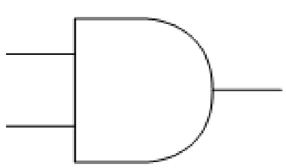
Examiner Tip

- Understanding the order of operations is crucial for correctly interpreting complex Boolean expressions
- Misunderstanding the order can lead to incorrect results
- Always use parentheses for clarity when combining multiple Boolean operations

Truth Tables

- A tool used in logic and computer science to **visualise** the **results** of **Boolean expressions**
- They represent **all possible inputs** and the **associated outputs** for a **given Boolean expression**

Conjunction (AND)

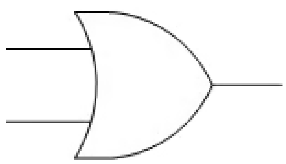
<table>
<tr><th>Circuit symbol</th><th>Truth Table</th></tr>
<tr><td></td><td>
<table>
<tr><th>A</th><th>B</th><th>A ^ B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
</table>

Your notes

Your notes

1	0	0
1	1	1

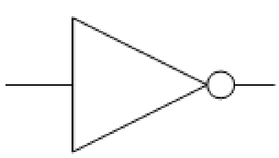
Disjunction (OR)



Circuit symbol	Truth Table

A	B	A V B
0	0	0
0	1	1
1	0	1
1	1	1

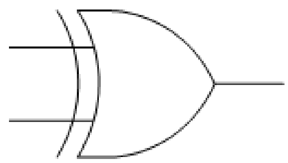
Negation (NOT)



Circuit symbol	Truth Table

A	¬A
0	1
1	0

Exclusive Disjunction (XOR)

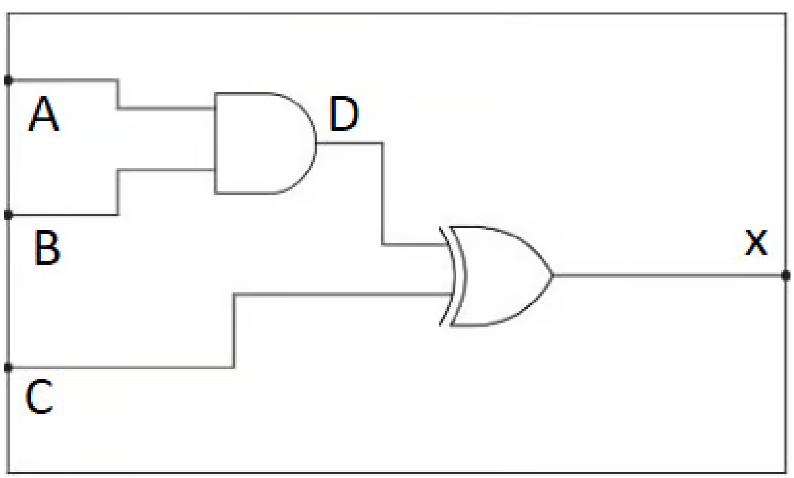
Circuit symbol	Truth Table
	

A	B	A V B
0	0	0
0	1	1
1	0	1
1	1	0

Your notes

Worked Example

Daniel is an engineer. He has created the following logic circuit as shown



Complete the truth table below for the logic circuit shown

A	B	C	D	X

Your notes

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

4 marks

Answer:

Example answer that gets full marks:

A	B	C	Calculating D	D	Calculating X	X	Mark
0	0	0	D is the result of A **AND** B	0	X is the result of D **XOR** C	0	1 Mark
0	0	1		0		1	
0	1	0		0		0	1 Mark
0	1	1		0		1	
1	0	0		0		0	1 Mark
1	0	1		0		1	
1	1	0		1		1	1 Mark
1	1	1		1		0	

Karnaugh Maps

Your notes

Karnaugh Maps

What is a Karnaugh Map?

- This is a tool that is used for **simplifying Boolean algebra expressions**
- It offers a **visual method** of **grouping together expressions** with **common factors**
- The format of the map makes it **easy to identify** and **eliminate redundant terms**
- They are used in digital logic design, such as simplifying the logic of digital circuits

Steps:

1. **Create the Map**: Each cell in the grid corresponds to a term in the Boolean expression. Fill cells with 1s and 0s corresponding to the output of that term
2. **Grouping**: Group the 1s in the grid. Each group must be a rectangle and the size of the group must be a power of 2. A cell can be part of multiple groups
3. **Simplifying**: Write down a simplified Boolean expression for each group. The simplified expression for a group consists of the variables that remain constant in all terms in the group
4. **Final Expression**: Combine the simplified expressions from each group using OR operations to get the final simplified Boolean expression

Creating Karnaugh Maps

- A Karnaugh Map (**KMap**) can be used to **simplify a Boolean expression** with 2 inputs
- Here is an example for the expression **A V B** (**A OR B**)

Step 1

- **Add each variable** starting with **A** at the top and **B** down the side
- **Add each possible state** for **A** and **B**

		A	A
		0	1
B	0		
B	1		

Step 2

- Take each expression in turn **separated** by the **V** (**OR**)
- First look at **A** on it's own
- Find **all cells where A is 1**
- **Add 1 to the cell**

		A	A
		0	1
B	0		1
B	1		1

Step 3

- **Repeat for B**

Your notes

Your notes

		A	A
		0	1
B	0		1
B	1	1	1

Step 4

- This is now a **completed KMap** for the expression **A V B** (**A OR B**)

		A	A
		0	1
B	0		1
B	1	1	1

Simplifying Expressions Using Karnaugh Maps

Simplify **¬A^¬B^C v ¬A^B^¬C v A^¬B^C v A^B** using a KMap.

In this example there will be **3 variables** A,B and C so the empty KMap will look like this:

		AB			
		00	01	11	10
C	0				
	1				

Your notes

Step 1:

Split this long term at each **OR** giving 4 smaller expressions (subterms) to add to the table:

- ¬A^¬B^C
- ¬A^B^¬C
- A^¬B^C
- A^B

Step 2:

- Take the first subterm **¬A^¬B^C**
- Put a **1** in the map for **every cell where this term would be TRUE (1)**
- So if **A** and **B** were **0** and **C** was **1** this subterm would be **1**
- So put a **1** in every cell in the KMap where **A is 0**, **B is 0** and **C is 1**

		AB			
		00	01	11	10
C	0				
	1	1			

Step 3:

- The next subterm is **¬A^B^¬C**
- Put a **1** in the KMap where **A is 0**, **B is 1** and **C is 0**

Your notes

		AB			
		00	01	11	10
C	0		1		
	1	1			

Step 4:

- The next subterm is **A^¬B^C**
- Put a **1** in the KMap where **A is 1**, **B is 0** and **C is 1**

		AB			
		00	01	11	10
C	0		1		
	1	1			1

Step 5:

- The final subterm is **A^B**
- Put a **1** in the KMap where **A is 1** and **B is 1** (2 cells this time)

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

Making the groups

Your notes

- Once you have written the 1s and 0s into your KMap, you can then use this to **simply the expression**
- In order to do this, you need to **identify the groups**
- This is the **key** to using the Karnaugh map to derive the simplified expression
- The aim of making the groups is to
  - **Make rectangular groups**
  - **Make groups that are as large as possible**
  - **Make groups that contain either 8,4,2 or 1 ones**
  - **Groups can overlap** (i.e. some ones can be in multiple groups)
  - **The Karnaugh map 'grid' wraps round in all directions so the groups can wrap round**

Example grouping

Group 1:

So this would be one group.

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

This group would represent **A** since within this group both C and B change (have zeros and ones) whereas for A all the cells are one.

Group 2:

This would be another group (**note the wrapping around**).

Your notes

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

This group would represent **¬B** since within this group both A and C change (have zeros and ones) whereas for B all the cells are zero

Final simplified expression

- To get the final simplified expression we **OR** the terms representing the two groups together.
- So the simplified expression is **A v ¬B**

Examiner Tip

In questions where you have to use a Karnaugh Map always show the groups by drawing a box/circle round them.

Worked Example

A Boolean expression is entered into a Karnaugh map.

Your notes

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

Give a simplified version of the expression using the Karnaugh map.

You must show your working

3 marks

How to answer this question:

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

- Looking at the group of 8 in the middle of the map, for all the cells in the group the variable B is always a 1
- The 3 other variables change across the group (i.e. for some cells they are 0 and for others they are 1
  So this group is B

- Looking at the other group of 8, for all the cells in this group, C is always 0 but the other 3 variables can be 1 or 0 in this group So this group simplifies to ¬C

Answer:

Answer that gets full marks:

B v ¬C

Your notes

Simplifying Boolean Algebra

Your notes

De Morgan's Law

- **Complex expressions** can be made **simpler** using the rules of **Boolean algebra**
- This is a **more powerful simplification** method than Karnaugh maps and can simplify expressions that Karnaugh maps cannot
- There are various different **rules** that you need to learn and that can then be applied to certain expressions to simplify them
- **Combining these rules** can mean that c**omplex expressions** can be **reduced to much simpler ones**

General rules

General AND rules

- X AND 0 = 0
- X AND 1 = X
- X AND A = X
- NOT X AND X = 0

Note, the value ox X is unknown and it is used as a placeholder. Therefore **X AND 1 = X** means that the output will be whatever the value of X is.

General OR rules

- X OR 0 = X
  X OR 1 = 1
  X OR A = X
  NOT X OR X = 1

DeMorgan's Law

- This provides a strategy for simplifying expressions that include a negation of a conjunction or disjunction (simplifying by inverting all variable)
- **NOT(A AND B)** is the same as (**NOT A) OR (NOT B)**

NOT (A AND B) ¬(A ^ B)				is the same as	(NOT A) OR (NOT B) ¬A V ¬B				
A	B	A AND B	NOT (A AND B)		A	B	NOT A	NOT B	(NOT A) OR (NOT B)
0	0	0	1	≡	0	0	1	1	1
0	1	0	1		0	1	1	0	1
1	0	0	1		1	0	0	1	1
1	1	1	0		1	1	0	0	0

- **Step 1**
  - Change AND to OR (or vice versa) - **¬(A V B)**
- **Step 2**
  - NOT the terms either side of the operator - **¬(¬A V ¬B)**
- **Step 3**
  - NOT everything that has changed - **¬¬(¬A V ¬B)**
- **Step 4**
  - Get rid of any double negation - **(¬A V ¬B)**
- **Step 5**
  - Remove any unnecessary brackets - **¬A V ¬B**
- **NOT(A OR B)** is the same as **(NOT A) AND (NOT B)**

NOT (A OR B) ¬(A V B)				is the same as	(NOT A) AND (NOT B) ¬A ^ ¬B				
A	B	A OR B	NOT (A OR B)		A	B	NOT A	NOT B	(NOT A) AND (NOT B)
0	0	0	1	≡	0	0	1	1	1
0	1	1	0		0	1	1	0	0
1	0	1	0		1	0	0	1	0
1	1	1	0		1	1	0	0	0

- **Step 1**
  - Change AND to OR (or vice versa) - **¬(A ^ B)**

Your notes

- **Step 2**
  - NOT the terms either side of the operator - **¬(¬A ^ ¬B)**
- **Step 3**
  - NOT everything that has changed - **¬¬(¬A ^ ¬B)**
- **Step 4**
  - Get rid of any double negation - **(¬A ^ ¬B)**
- **Step 5**
  - Remove any unnecessary brackets - **¬A ^ ¬B**
- Simplifying using this law allows the use of **only NAND or NOR gates** which makes **building microprocessors** much **easier** (i.e. Flash drives)

Distribution

Distributive Law

- This explains how AND and OR interact with each other
- This is a bit like factorising in normal maths
- **A AND (B OR C)** is the same as **(A AND B) OR (A AND C)**
- **A OR (B AND C)** is the same as **(A OR B) AND (A OR C**

Real-life example

- "*You can pick one subject from group A and either one from group B or group C*"
- is the same as
- "*You can pick one subject from group A and one from group B or one subject from group A and one from group C*"

Association

Associative Law

- This explains how variables associate in expressions of more than two variables
- Allows us to remove brackets and regroup variables
- **(A AND B) AND C** is the same as **A AND (B AND C)** is the same as **A AND B AND C**
- **(A OR B) OR C** is the same as **A OR (B OR C)** is the same as **A OR B OR C**

Your notes

Your notes

Real-life example

- *"Zarmeen and her friends, Zahra and Ella have been chosen to represent the school"*
- is the same as
- *"Zarmeen and Zahra, and their friend Ella have been chosen to represent the school"*
- is the same as
- *"Zarmeen, Zahra and Ella have been chosen to the represent the school"*

Commutation

Commutative Law

- States that the order of the variables does not change the truth value of the expression
- **A AND B** is the same as **B AND A**
- **A OR B** is the same as **B OR A**

Real-life example

- *"Fynn and George won gold medals"*
- is the same as
- *George and Fynn won gold medals"*

Double Negation

Double Negation Law

- States that the double negation of a variable results in the original variable
- NOT(NOT(A)) = A

Real-life example

- *"I don't not want to visit the castle"*
- is the same as
- *"I do want to visit the castle"*

Worked Example

SIMPLIFYING EXPRESSION EXAMPLE

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)

Your notes

Your notes

Flip Flop Circuits

D Type Flip Flops

What is a D Type Flip Flop?

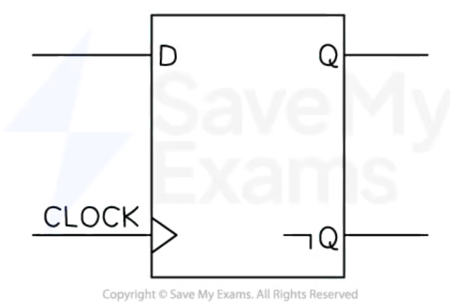
- A **fundamental component in digital circuits** and **computer memory**
- Can be referred to as **Positive Edge Triggered**

Key Features

- Contains **two stable states**, making it a type of bistable circuit
- Used to **store** the **state of 1 bit of data**
- Changes state on the **edge of the clock pulse**

Components

- Data input (**D**)
- Clock input (**CLK**)
- Two outputs: **Q** and **NOT(Q)**



D Type Flip Flip Components

Operation

- On the rising edge of the clock pulse:
  - If D is **high** (1), Q goes **high** and NOT(Q) goes **low**
  - If D is **low** (0), Q goes **low** and NOT(Q) goes **high**

- The state of Q holds or "**remembers**" its value until the **next** rising clock edge

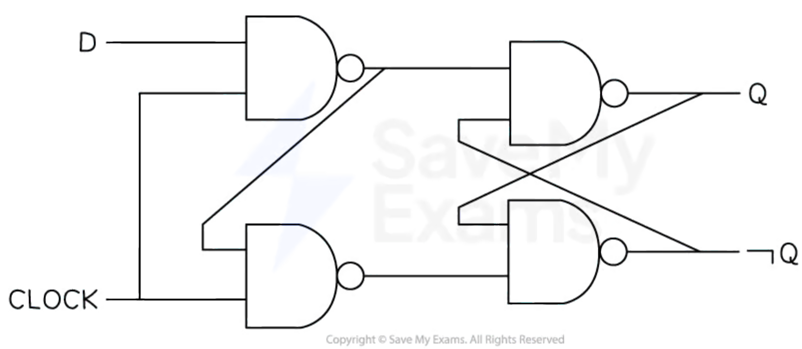
Use Cases

- Forms the basis of most types of flip flops and latches
- Used in **shift registers**, **counters**, and **memory units**
- Helpful in **edge-triggered devices**, **synchronous circuits**, and **data storage**

Your notes

Examiner Tip

You will not be asked to recall the logic gates that make up a D-type flip flop and they can be made from various different gates
They are often built using NAND gates which are AND gates that have their output inverted:

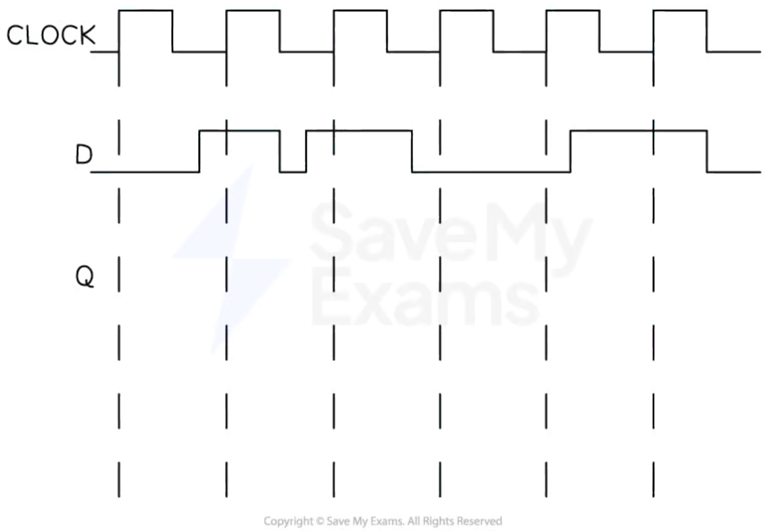


D Type Flip Flip Logic Gates

Worked Example

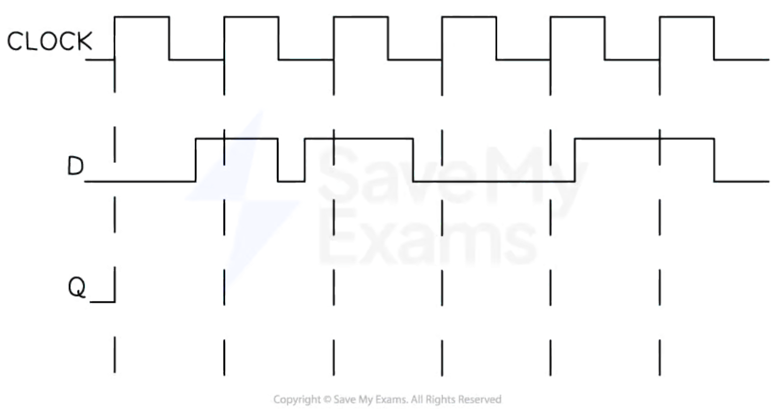
- **Draw the output of the Flip Flop on the diagram below**

Your notes



D Type Flip Flop Question

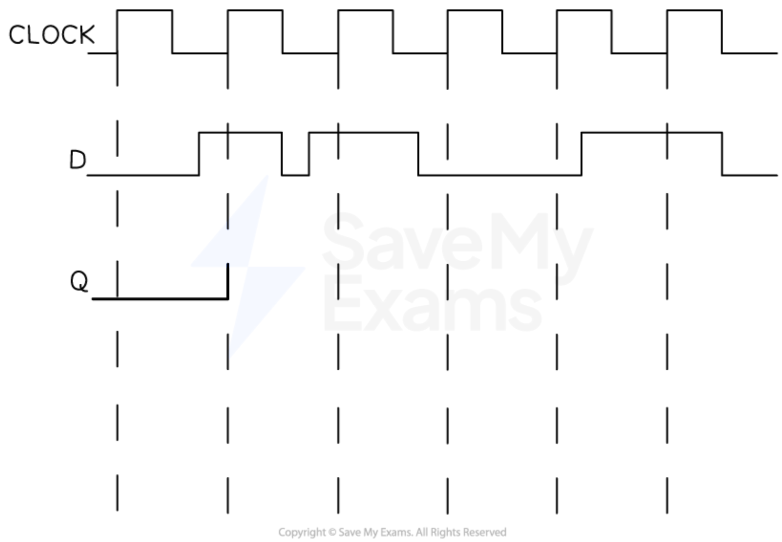
- For a question like this remember that the output line Q that you are asked to draw wants to be the same as the input D BUT it can only change when the clock signal changes from low to high (i.e. at the dotted lines)
- At the start assume Q starts the same as D (low in this case)
- Then draw it along from left to right until you get to a vertical dotted line (in an exam you would draw these lines on to help you if they weren't there
- At each dotted line, Q has the chance to change to whatever D is



D Type Flip Flop Working 1

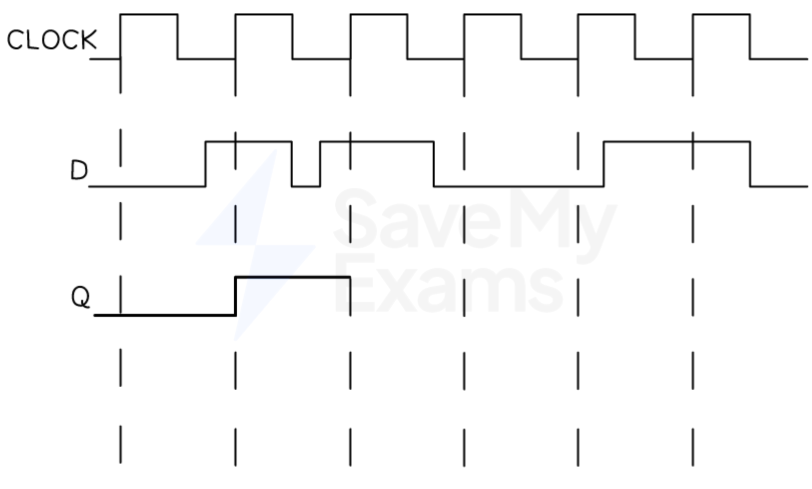
- So at the first line, D is low so Q stays low
- At the second dotted line D has changed to high so now Q can become high

Your notes



D Type Flip Flop Working 2

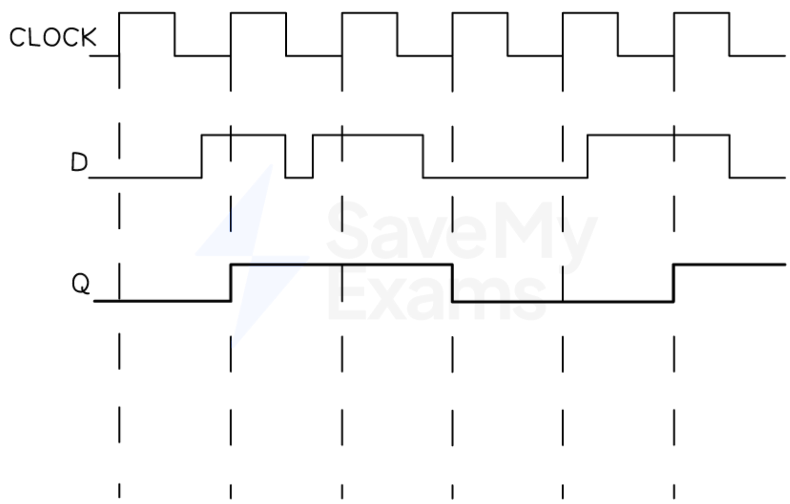
- Q now stays high until the next rising edge of the clock (the next dotted line) where it gets another chance to change, but here D is still high so Q stays high



D Type Flip Flop Working 3

- And you continue to do this until you reach the right hand side of the diagram

Your notes



D Type Flip Flop Working 4

Your notes

Adder Circuits

Half Adders

What is a Half Adder Circuit?

- Basic digital circuit used in computation to perform the **addition** of **two single bit numbers.**
- Has **two inputs**, usually labelled as **A** and **B**
- Produces **two outputs** labelled **Carry out** (C_{out}) and **Sum**(s)

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
		A AND B	**A XOR B**

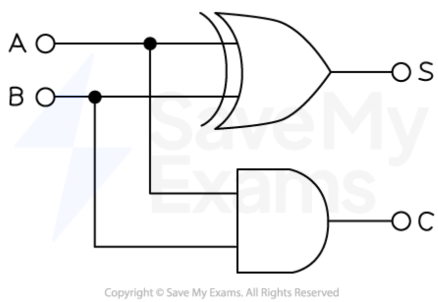
- Remember that you are **adding together** the **binary numbers** represented by **A and B**
- Create the **C_{out} column first** then for each row you can just **add A and B together** and write the **answer in 2 bits** in the **C_{out}** and **S** columns
  - For example in row 2, A is 0 and B is 1 and 0+1=1, which is 01 in 2 bits (C_{out} 0 and Sum 1)
  - In the last row, A is 1 and B is 1 and 1+1 = 2 which is 10 in 2 bit binary (C_{out} 1 and Sum 0)

Drawing a Half Adder Circuit

- A half adder circuit has **two inputs**, typically labelled as **A** and **B**, and **two outputs**: the **Sum** (S) and **Carry** (C_{out}). This circuit can be created using an **XOR gate** for the **Sum output** and an **AND gate** for the **Carry output**
- **Label Inputs**:
  - Begin by drawing two lines on the left side of your paper or drawing space. Label the top line as '**A**' and the bottom line as '**B**'. These represent your **inputs**

Your notes

- **XOR Gate (Sum)**:
  - **Draw an XOR gate** (often a shape like a curved 'D' or a shape similar to an OR gate but with an additional curved line on the input side) in the middle of the paper or drawing space. Connect the A and B lines to the two inputs of the **XOR gate**
  - The output from the XOR gate is the '**Sum**'. Draw a line from the output of the **XOR** gate to the right side of your paper and label it as '**S**'
- **AND Gate (Carry)**:
  - **Draw an AND gate** (typically a D-shaped symbol) above the **XOR gate**. Again, connect the A and B lines to the two inputs of the **AND gate**.
  - The output from the **AND gate** is the '**Carry**'. Draw a line from the output of the **AND** gate to the right side of your paper and label it as 'C_{out}'



Half Adder Logic Gates

Full Adders

- **Extends** the half adder to handle the **addition of three bits**
- Has **three inputs**: **A**, **B**, and an input carry (C_{in})
- Produces **two outputs**: carry (C_{out}) and sum (**S**)

A	B	C_{in}	C_{out}	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

Your notes

- To easily reproduce this **Truth Table**, remember:
  - The full adder **adds up three binary inputs** A,B and C
  - So the answer can be **0,1,2 or 3**
  - For each row, add up A, B and C and the **write the answer as a 2 bit binary number** in the last 2 columns (C_{out} and Sum)
  - For example in row 4, A=0, B=1 and C=1 - **0+1+1=2** which is **10 in binary**, so **C_{out} is 0 and Sum is 1**
  - In the last row, A=1, B=1 and C=1, **1+1+1=3** which is **11 in binary** so **C_{out} is 1 and Sum is 1**

Operation

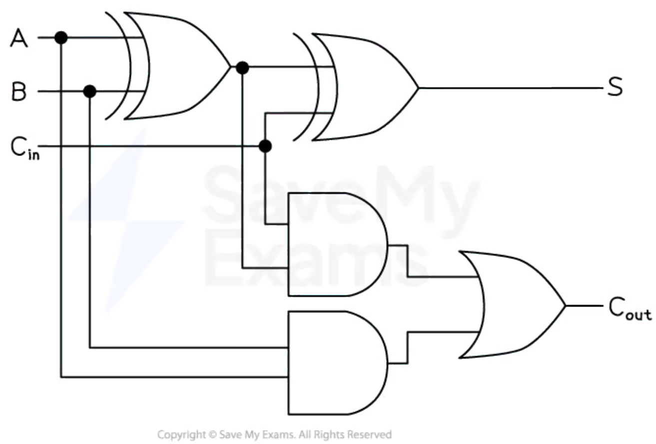
- The "Sum" output **provides the XOR** of the inputs A, B, and C_{in}
- The "Carry" output is **TRUE** if at **least two of the inputs** A, B, and C_{in} are **TRUE**

Drawing a Full Adder Circuit

- A full adder circuit consists of **three inputs**: **A**, **B**, and Carry (**C_{in}**), and **two outputs**: Sum (**S**) and Carry (**C_{out}**)
- It can be designed using **two half adders** and an **OR gate**.
- **Label Inputs**:
  - Start by **drawing three lines** on the left side of your paper or drawing space. Label the top line as '**A**', the middle line as '**B**', and the bottom line as '**C_{in}**'. These represent your inputs
- **First Half Adder**:
  - **Draw a half adder** with A and B as inputs. This consists of an **XOR gate** (for the Sum) and an **AND gate** (for the Carry). Label the output of the XOR gate as '**Sum1**' and the output of the AND gate as '**Carry1**'

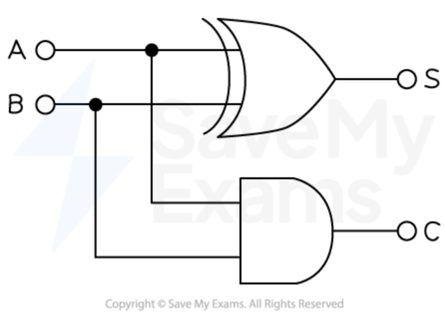
Your notes

- **Second Half Adder**:
  - **Draw a second half adder** underneath the first, using **Sum1** and **C_{in}** as inputs. Again, it consists of an **XOR gate** (for the Sum) and an **AND gate** (for the Carry). Label the output of the XOR gate as '**S**' (final Sum) and the output of the **AND gate** as '**Carry2**'
- **OR Gate**:
  - **Draw an OR gate** to the right of the half adders. Connect Carry1 and Carry2 to the inputs of the OR gate. The output of the OR gate is the final Carry (**C_{out}**)



Full Adder Logic Gates

Worked Example



Describe how this logic circuit can be adapted to add together two 4-bit binary numbers.

4 marks

Answer:

Logic circuit adds together 2 binary digits / half adder

S gives sum, C gives carry

Two half adders can be joined together...

...with an OR gate

to form full adder

4 full adders can be used to add two four bit numbers

Carry out on one joined to carry in on next

Your notes