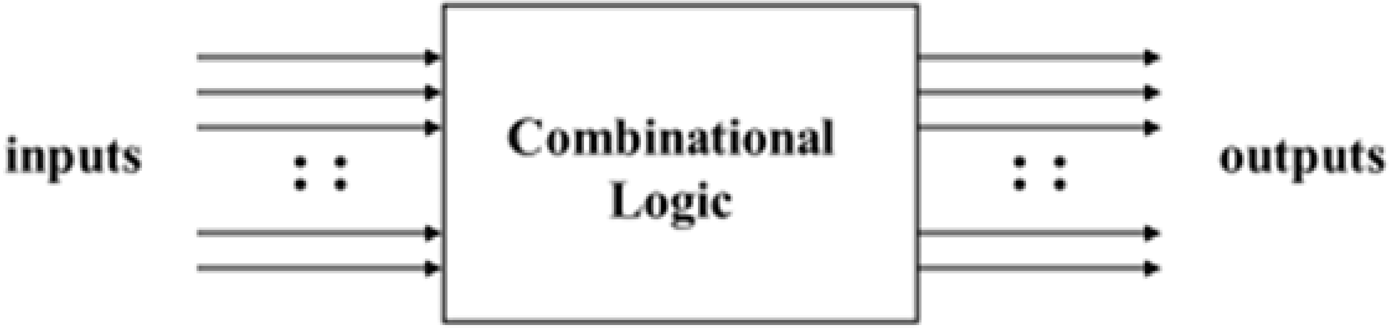
Module 1

Module I (11 Hours)

- Representation of signed numbers – 1's complement and 2’s complement , Logic gates – AND - OR – NOT - NAND- NOR - XOR , Boolean algebra - Basic laws and theorems , Boolean functions - truth table, Standard forms of Boolean Expressions – Sum of Products and Product of Sums - minimization of Boolean function using Karnaugh map method - Realization using logic gates, Floating point numbers.
- Combinational Circuits - Half adder - Full Adder- Decoder - Encoder- Multiplexer – Demultiplexer

Introduction

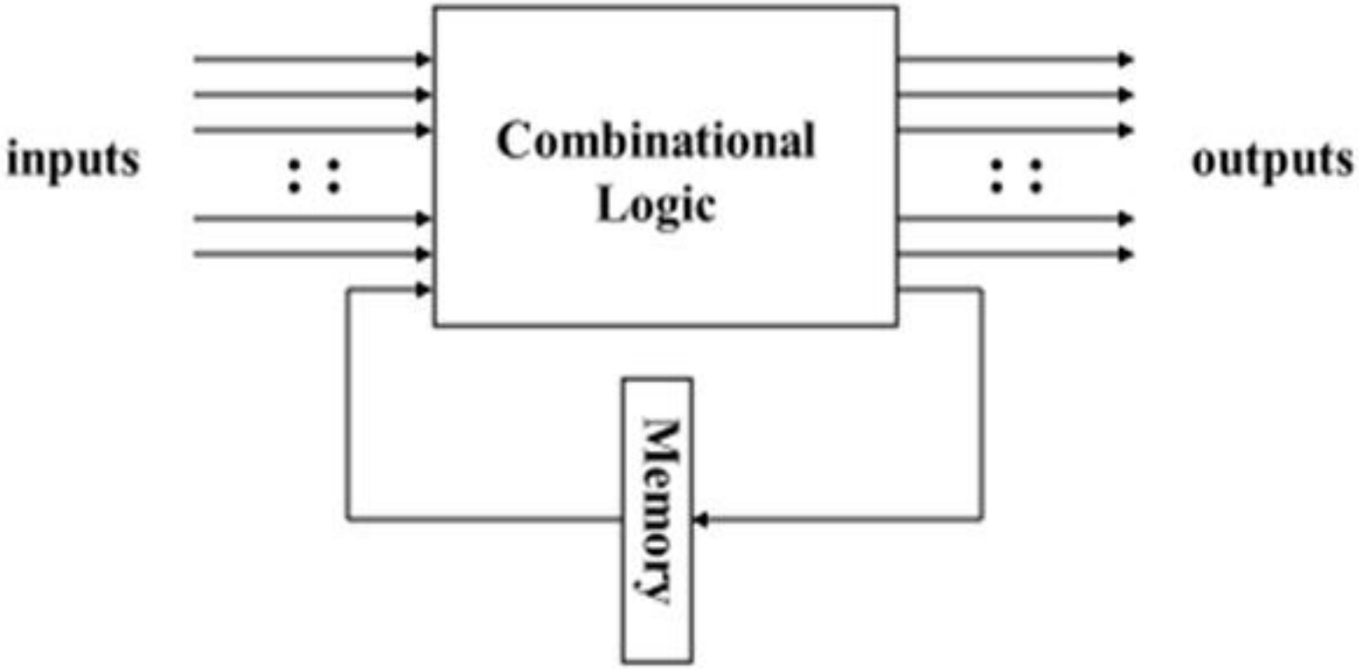
- Two classes of logic circuits:
  - combinational
  - sequential
- **Combinational Circuit**:



Each output depends entirely on the immediate (present) inputs.

Introduction

- **Sequential Circuit**: (not covered)



Output depends on both *present* and *past* inputs.
Memory (via feedback loop) contains past information.

Binary Numbers

For digital systems, the binary number system is used. Binary uses the digits 0 and 1 to represent quantities.

The column weights of binary numbers are powers of two that increase from right to left beginning with 2^0 =1:

$$\ldots 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0.$$

For fractional binary numbers, the column weights are negative powers of two that decrease from left to right:

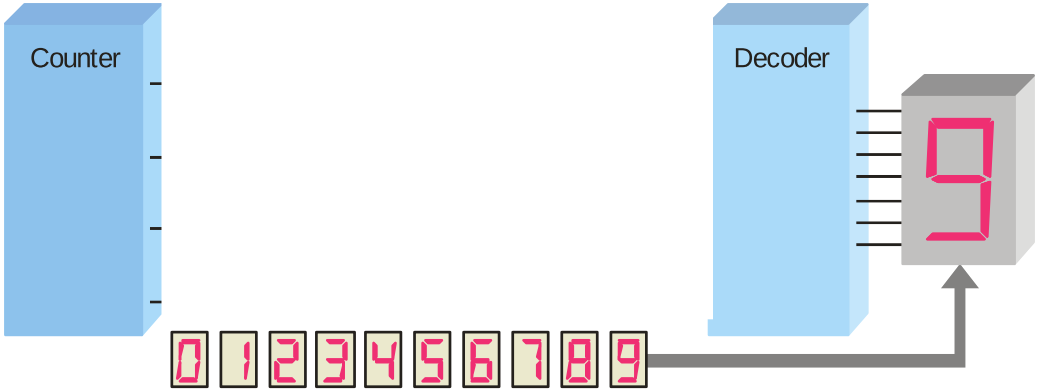
$$2^2\ 2^1\ 2^0.\ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ \ldots$$

Binary Numbers

A binary counting sequence for numbers from zero to fifteen is shown.

Notice the pattern of zeros and ones in each column.

Digital counters frequently have this same pattern of digits:



Decimal Number	Binary Number
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
10	1 0 1 0
11	1 0 1 1
12	1 1 0 0
13	1 1 0 1
14	1 1 1 0
15	1 1 1 1

Signed Magnitude Representation

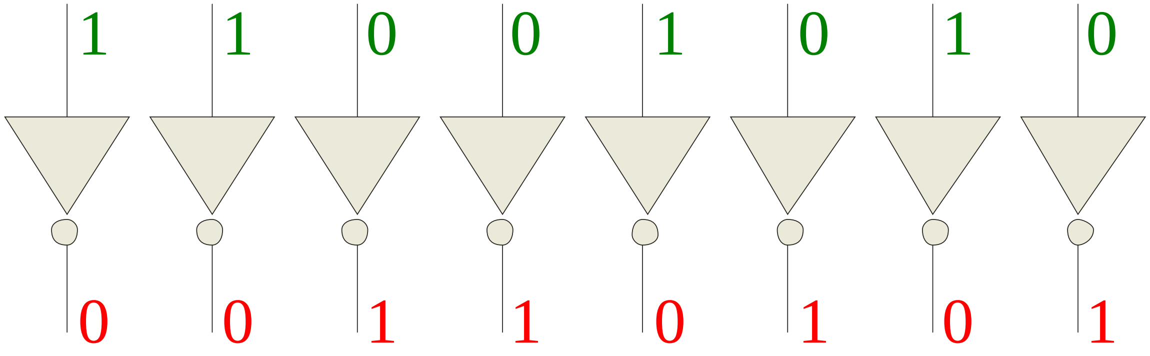
- A number's sign is represented with a **sign bit**:
  - setting SIGN bit (often the most significant bit) to 0 for a positive number and setting it to 1 for a negative number.
- The remaining bits in the number indicate the magnitude (or absolute value).
- For example, -43_{10} encoded in an eight-bit byte is **1**0101011 while 43_{10} is **0**0101011.

1’s Complement

The 1’s complement of a binary number is just the inverse of the digits. To form the 1’s complement, change all 0’s to 1’s and all 1’s to 0’s.

For example, the 1’s complement of 11001010 is
00110101

In digital circuits, the 1’s complement is formed by using inverters:



2's Complement

The 2's complement of a binary number is found by adding 1 to the LSB of the 1's complement.

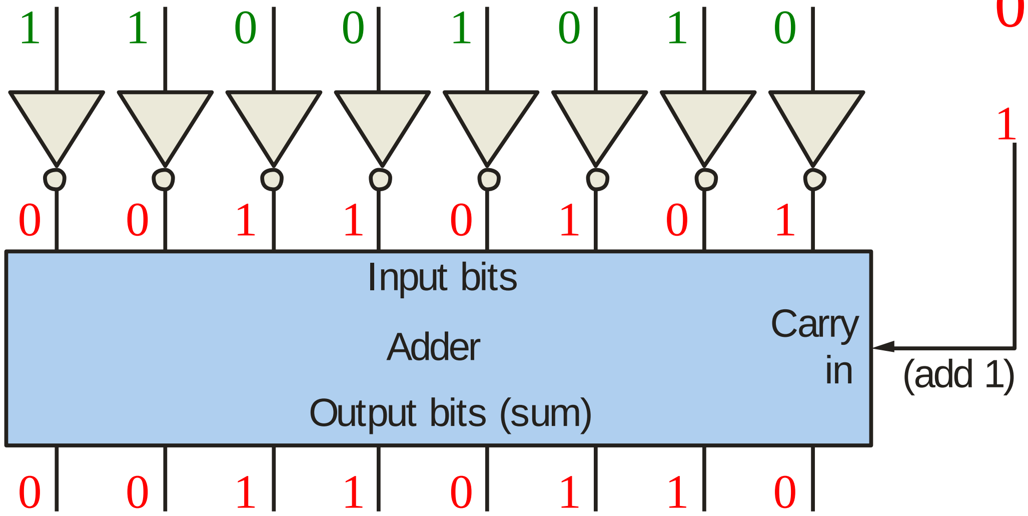
Recall that the 1's complement of 11001010 is

00110101 (1's complement)

To form the 2's complement, add 1:

$$\begin{array}{r} 00110101 \\ +1 \\ \hline 00110110 \end{array}$$

00110110 (2's complement)



Signed Binary Numbers

There are several ways to represent signed binary numbers.

In all cases, the MSB in a signed number is the sign bit, that tells you if the number is positive or negative.

Computers use a modified 2’s complement for signed numbers. Positive numbers are stored in *true* form (with a 0 for the sign bit) and negative numbers are stored in *complement* form (with a 1 for the sign bit).

For example, the positive number 58 is written using 8-bits as 00111010 (true form).

Sign bit

Magnitude bits

Signed Binary Numbers

Negative numbers are written as the 2’s complement of the corresponding positive number.

The negative number –58 is written as:

–58 = 11000110 (complement form)

Sign bit

Magnitude bits

An easy way to read a signed number that uses this notation is to assign the sign bit a column weight of –128 (for an 8-bit number). Then add the column weights for the 1’s.

Example

Assuming that the sign bit = –128, show that 11000110 = –58 as a 2’s complement signed number:

Solution

Column weights:	–128	64	32	16	8	4	2	1.	
	1	1	0	0	0	1	1	0	
	–128	+64				+4	+2		= –58

Arithmetic Operations with Signed Numbers

Using the signed number notation with negative numbers in 2’s complement form simplifies addition and subtraction of signed numbers.

Rules for **addition**: Add the two signed numbers. Discard any final carries. The result is in signed form.

Examples:

$$\begin{array}{r} 00011110 = +30 \\ 00001111 = +15 \\ \hline 00101101 = +45 \end{array}$$

$$\begin{array}{r} 00001110 = +14 \\ 11101111 = -17 \\ \hline 11111101 = \ -3 \end{array}$$

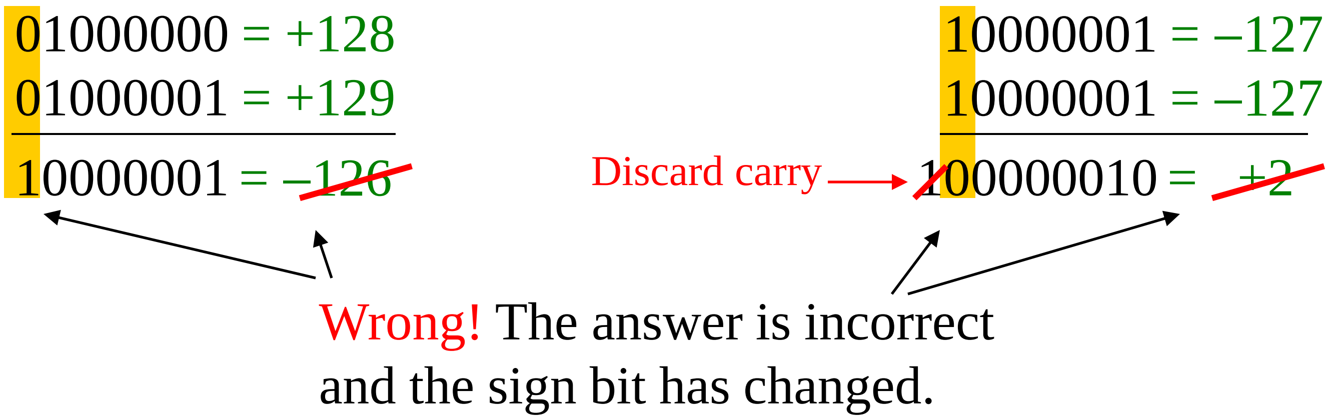
$$\begin{array}{r} 11111111 = \ -1 \\ 11111000 = \ -8 \\ \hline \cancel{1}\,11110111 = \ -9 \end{array}$$

Discard carry

Arithmetic Operations with Signed Numbers

Note that if the number of bits required for the answer is exceeded, overflow will occur. This occurs only if both numbers have the same sign. The overflow will be indicated by an incorrect sign bit.

Two examples are:



01000000 = +128
01000001 = +129
10000001 = ~~–126~~

10000001 = –127
10000001 = –127
Discard carry → ~~1~~00000010 = ~~+2~~

Wrong! The answer is incorrect
and the sign bit has changed.

Arithmetic Operations with Signed Numbers

Rules for **subtraction**: 2’s complement the subtrahend and add the numbers. Discard any final carries. The result is in signed form.

Repeat the examples done previously, but subtract:

$$\begin{array}{rll} & 00011110 & (+30) \\ - & 00001111 & -(+15) \\ \hline \end{array} \qquad \begin{array}{rll} & 00001110 & (+14) \\ - & 11101111 & -(-17) \\ \hline \end{array} \qquad \begin{array}{rll} & 11111111 & (-1) \\ - & 11111000 & -(-8) \\ \hline \end{array}$$

2’s complement subtrahend and add:

$$\begin{array}{r} 00011110 = +30 \\ 11110001 = -15 \\ \hline \cancel{1}00001111 = +15 \end{array} \qquad \begin{array}{r} 00001110 = +14 \\ 00010001 = +17 \\ \hline 00011111 = +31 \end{array} \qquad \begin{array}{r} 11111111 = -1 \\ 00001000 = +8 \\ \hline \cancel{1}00000111 = +7 \end{array}$$

Discard carry

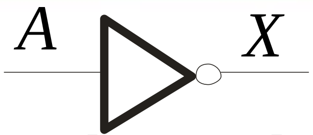
Discard carry

Syllabus

Module I (11 Hours)

- Representation of signed numbers – 1's complement and 2’s complement , Logic gates – AND - OR – NOT - NAND- NOR - XOR , Boolean algebra - Basic laws and theorems , Boolean functions - truth table, Standard forms of Boolean Expressions – Sum of Products and Product of Sums - minimization of Boolean function using Karnaugh map method - Realization using logic gates, Floating point numbers.
- Combinational Circuits - Half adder - Full Adder- Decoder - Encoder- Multiplexer – Demultiplexer

The Inverter

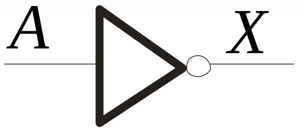


The inverter performs the Boolean **NOT** operation. When the input is LOW, the output is HIGH; when the input is HIGH, the output is LOW.

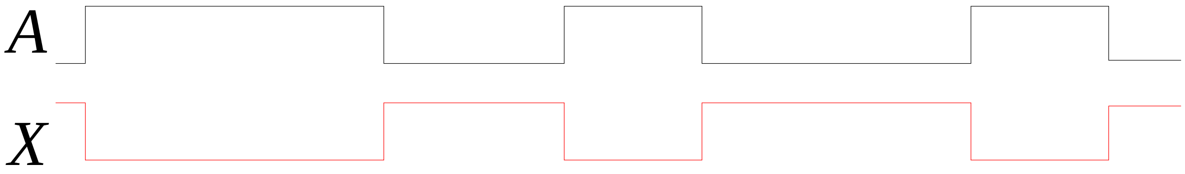
Input	Output
A	X
LOW (0) HIGH (1)	HIGH (1) LOW(0)

The **NOT** operation (complement) is shown with an overbar. Thus, the Boolean expression for an inverter is $X = \overline{A}$.

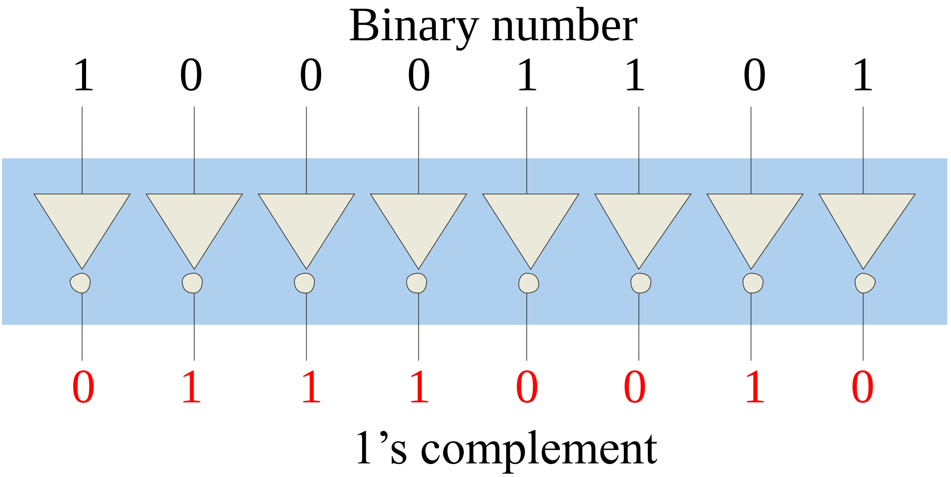
The Inverter



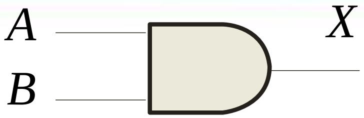
Example waveforms:

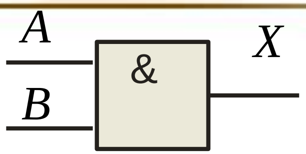


A group of inverters can be used to form the 1's complement of a binary number:



The AND Gate



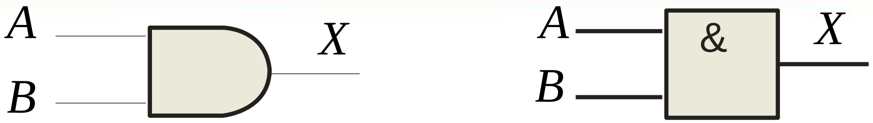


The **AND gate** produces a HIGH output when all inputs are HIGH; otherwise, the output is LOW. For a 2-input gate, the truth table is

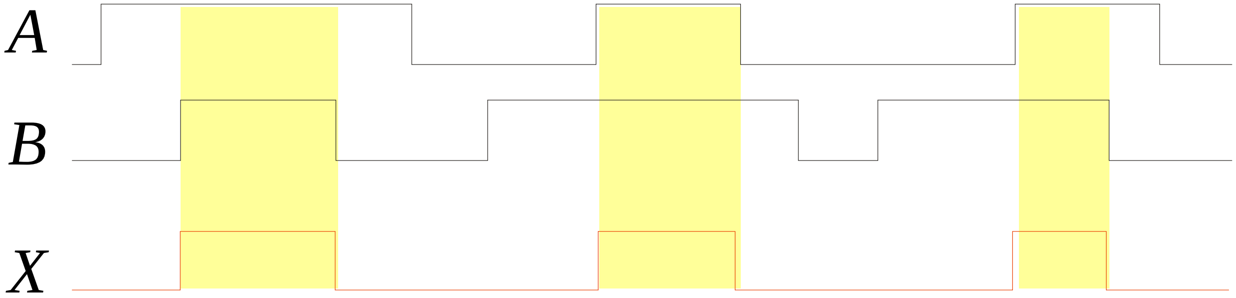
Inputs		Output
A	*B*	*X*
0	0	0
0	1	0
1	0	0
1	1	1

The **AND** operation is usually shown with a dot between the variables but it may be implied (no dot). Thus, the AND operation is written as $X = A \cdot B$ or $X = AB$

The AND Gate



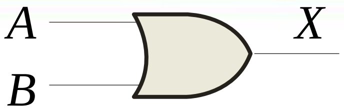
Example waveforms:



The AND operation is used in computer programming as a selective mask. If you want to retain certain bits of a binary number but reset the other bits to 0, you could set a mask with 1's in the position of the retained bits.

Example If the binary number 10100011 is ANDed with the mask 00001111, what is the result? 00000011

The OR Gate

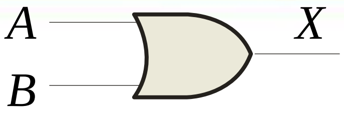


The **OR gate** produces a HIGH output if any input is HIGH; if all inputs are LOW, the output is LOW. For a 2-input gate, the truth table is

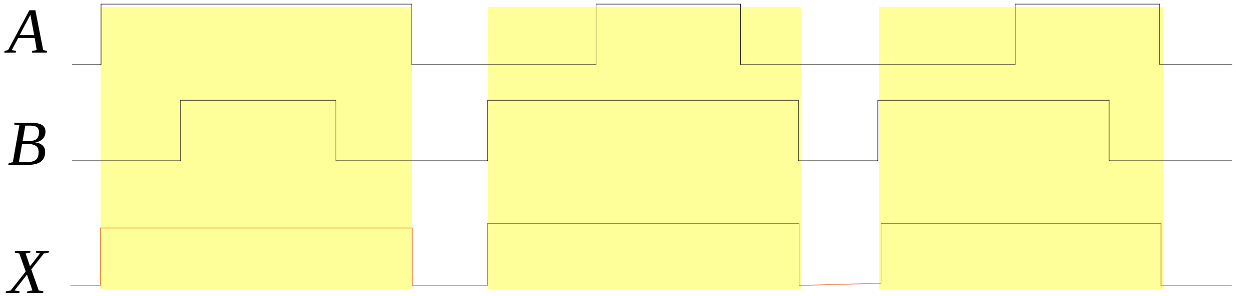
Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	1

The **OR** operation is shown with a plus sign (+) between the variables. Thus, the OR operation is written as $X = A + B$.

The OR Gate



Example waveforms:

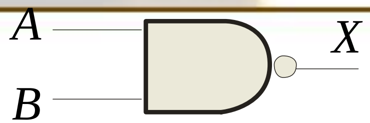


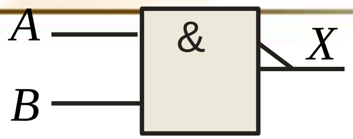
The OR operation can be used in computer programming to set certain bits of a binary number to 1.

Example ASCII letters have a 1 in the bit 5 position for lower case letters and a 0 in this position for capitals. (Bit positions are numbered from right to left starting with 0.) What will be the result if you OR an ASCII letter with the 8-bit mask 00100000?

Solution The resulting letter will be lower case.

The NAND Gate



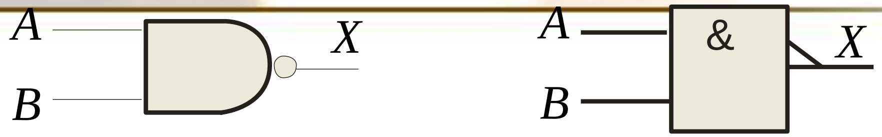


The **NAND gate** produces a LOW output when all inputs are HIGH; otherwise, the output is HIGH. For a 2-input gate, the truth table is

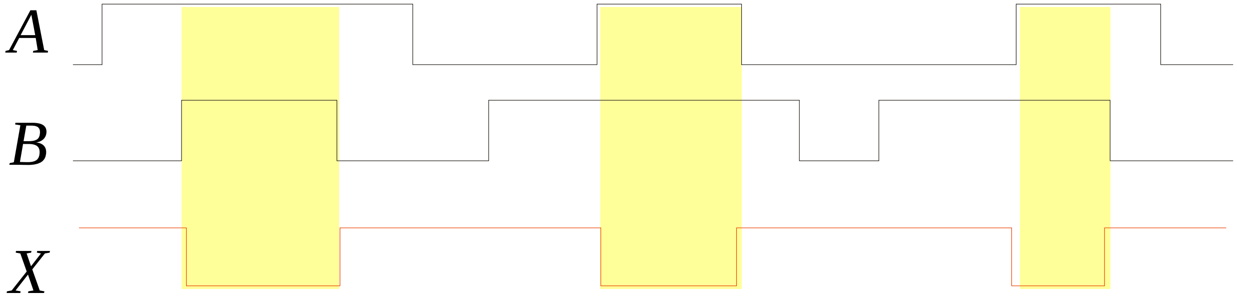
Inputs		Output
A	*B*	*X*
0	0	1
0	1	1
1	0	1
1	1	0

The **NAND** operation is shown with a dot between the variables and an overbar covering them. Thus, the NAND operation is written as $X = \overline{A \cdot B}$ (Alternatively, $X = \overline{AB}$.)

The NAND Gate

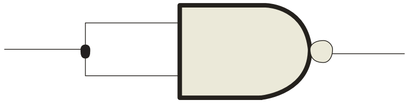


Example waveforms:

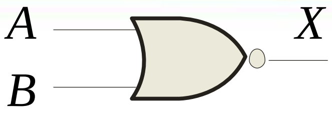


The NAND gate is particularly useful because it is a “universal” gate – all other basic gates can be constructed from NAND gates.

Question How would you connect a 2-input NAND gate to form a basic inverter?



The NOR Gate

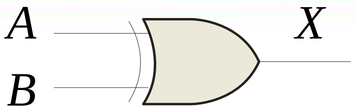


The **NOR gate** produces a LOW output if any input is HIGH; if all inputs are HIGH, the output is LOW. For a 2-input gate, the truth table is

Inputs		Output
A	*B*	*X*
0	0	1
0	1	0
1	0	0
1	1	0

The **NOR** operation is shown with a plus sign (+) between the variables and an overbar covering them. Thus, the NOR operation is written as $X = \overline{A + B}$.

The XOR Gate

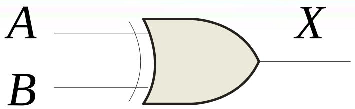


The **XOR gate** produces a HIGH output only when both inputs are at opposite logic levels. The truth table is

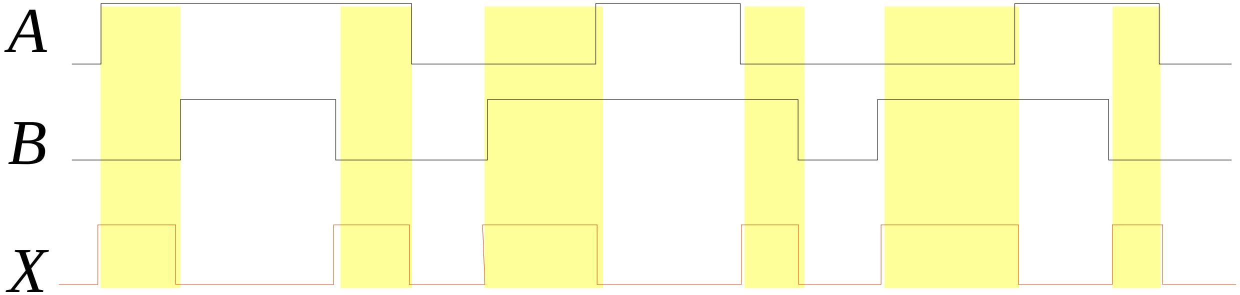
Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

The **XOR** operation is written as $X = \overline{A}B + A\overline{B}$. Alternatively, it can be written with a circled plus sign between the variables as $X = A \oplus B$.

The XOR Gate



Example waveforms:

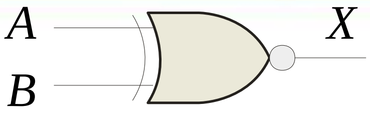


Note that the XOR gate will produce a HIGH only when exactly one input is HIGH.

Question If the *A* and *B* waveforms are both inverted for the above waveforms, how is the output affected?

There is no change in the output.

The XNOR Gate

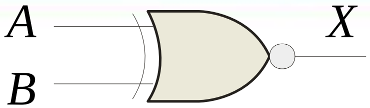


The **XNOR gate** produces a HIGH output only when both inputs are at the same logic level. The truth table is

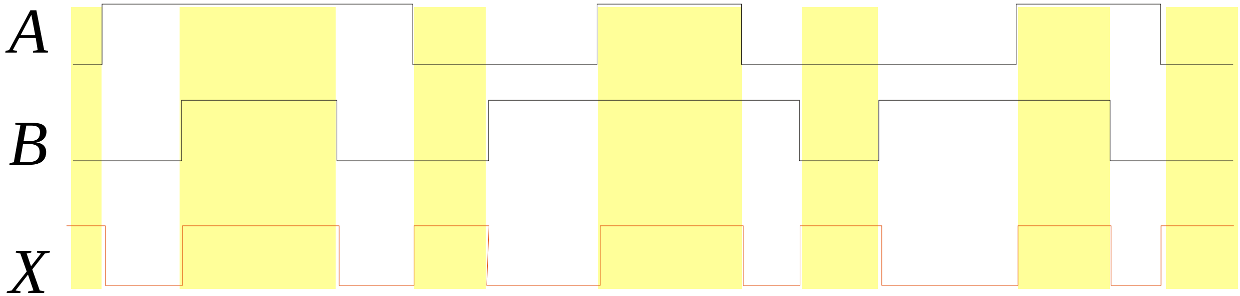
Inputs		Output
A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

The **XNOR** operation shown as $X = \overline{A}\overline{B} + AB$. Alternatively, the XNOR operation can be shown with a circled dot between the variables. Thus, it can be shown as $X = A \odot B$.

The XNOR Gate



Example waveforms:



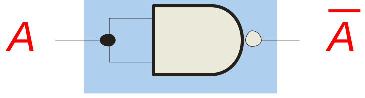
Note that the XNOR gate will produce a HIGH when both inputs are the same. This makes it useful for comparison functions.

Question If the *A* waveform is inverted but *B* remains the same, how is the output affected?

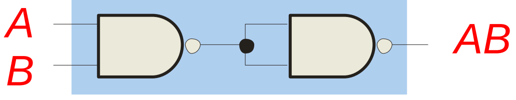
The output will be inverted.

Universal Gates

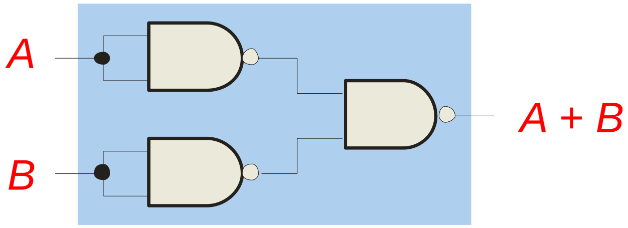
NAND gates are sometimes called **universal** gates because they can be used to produce the other basic Boolean functions.



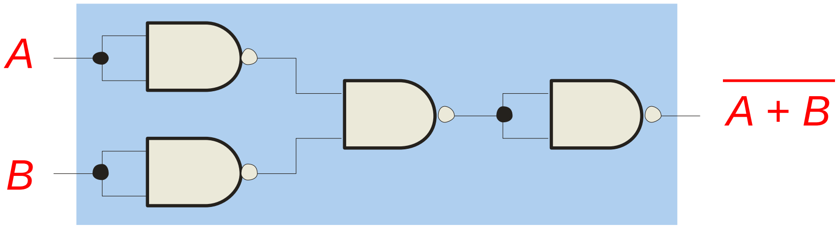
Inverter



AND gate



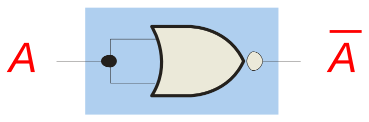
OR gate



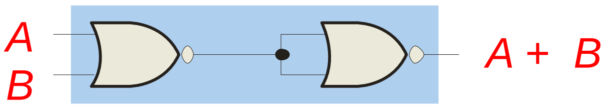
NOR gate

Universal Gates

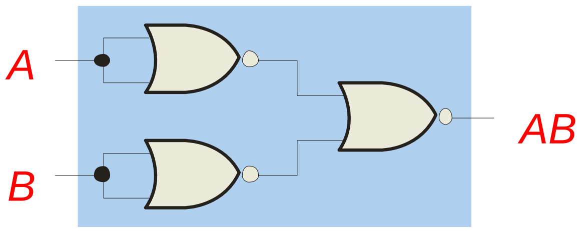
NOR gates are also **universal** gates and can form all of the basic gates.



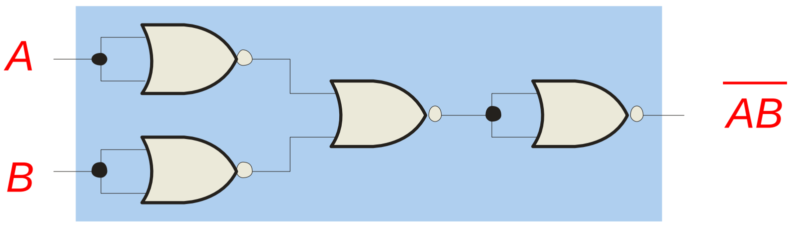
Inverter



OR gate



AND gate



NAND gate

Syllabus

Module I (11 Hours)

- Representation of signed numbers – 1's complement and 2’s complement , Logic gates – AND - OR – NOT - NAND- NOR - XOR , Boolean algebra - Basic laws and theorems , Boolean functions - truth table, Standard forms of Boolean Expressions – Sum of Products and Product of Sums - minimization of Boolean function using Karnaugh map method - Realization using logic gates, Floating point numbers.
- Combinational Circuits - Half adder - Full Adder- Decoder - Encoder- Multiplexer – Demultiplexer

Boolean Algebra

In Boolean algebra, a **variable** is a symbol used to represent an action, a condition, or data. A single variable can only have a value of 1 or 0.

The **complement** represents the inverse of a variable and is indicated with an overbar. Thus, the complement of A is $\overline{A}$.

A **literal** is a variable or its complement.

Addition is equivalent to the OR operation. The sum term is 1 if one or more of the literals are 1. The sum term is zero only if each literal is 0.

Boolean Addition

Addition is equivalent to the OR operation. The sum term is 1 if one or more of the literals are 1. The sum term is zero only if each literal is 0.

Example

Determine the values of *A*, *B*, and *C* that make the sum term of the expression $\overline{A} + B + \overline{C} = 0$?

Solution

Each literal must = 0; therefore *A* = 1, *B* = 0 and *C* = 1.

Boolean Multiplication

In Boolean algebra, multiplication is equivalent to the AND operation. The product of literals forms a product term. The product term will be 1 only if all of the literals are 1.

Example What are the values of the A, B and C if the product term of $A \cdot \overline{B} \cdot \overline{C} = 1$?

Solution Each literal must = 1; therefore $A = 1$, $B = 0$ and $C = 0$.

Commutative Laws

The **commutative laws** are applied to addition and multiplication. For addition, the commutative law states

In terms of the result, the order in which variables are ORed makes no difference.

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

For multiplication, the commutative law states

In terms of the result, the order in which variables are ANDed makes no difference.

$$\boxed{AB = BA}$$

Associative Laws

The **associative laws** are also applied to addition and multiplication. For addition, the associative law states

When ORing more than two variables, the result is the same regardless of the grouping of the variables.

$$\boxed{A + (B + C) = (A + B) + C}$$

For multiplication, the associative law states

When ANDing more than two variables, the result is the same regardless of the grouping of the variables.

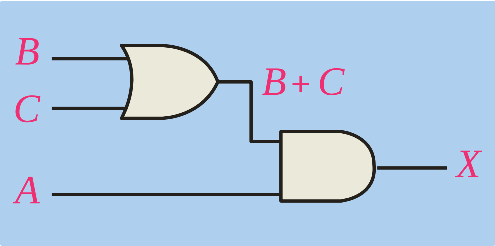
$$\boxed{A(BC) = (AB)C}$$

Distributive Law

The **distributive law** is the factoring law. A common variable can be factored from an expression just as in ordinary algebra. That is

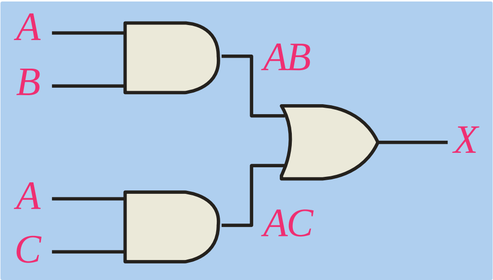
$$\boxed{A(B+C) = AB + AC}$$

The distributive law can be illustrated with equivalent circuits:



$A(B+C)$

$\equiv$



$AB + AC$

Rules of Boolean Algebra

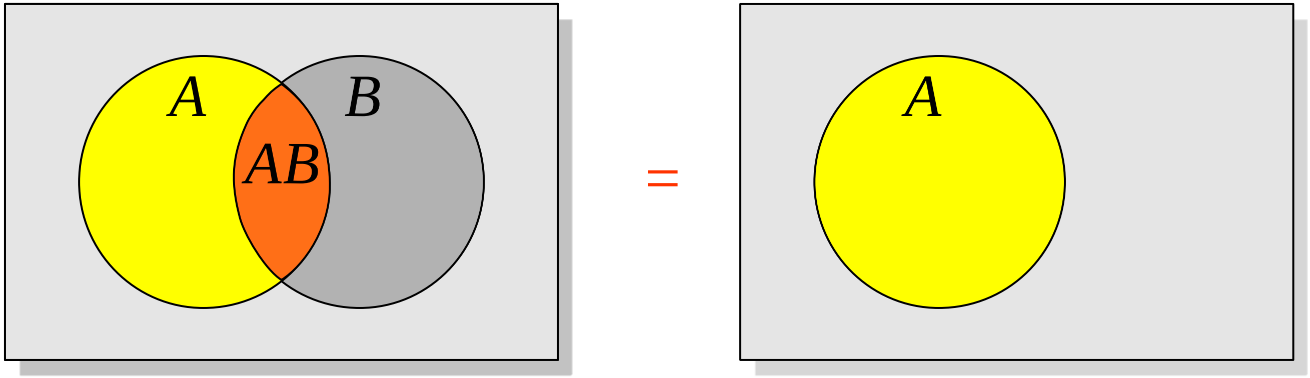
1. $A + 0 = A$
2. $A + 1 = 1$
3. $A \cdot 0 = 0$
4. $A \cdot 1 = A$
5. $A + A = A$
6. $A + \overline{A} = 1$
7. $A \cdot A = A$
8. $A \cdot \overline{A} = 0$
9. $\overline{\overline{A}} = A$
10. $A + AB = A$
11. $A + \overline{A}B = A + B$
12. $(A + B)(A + C) = A + BC$

Rules of Boolean Algebra

Rules of Boolean algebra can be illustrated with *Venn* diagrams. The variable *A* is shown as an area.

The rule $A + AB = A$ can be illustrated easily with a diagram. Add an overlapping area to represent the variable *B*.

The overlap region between A and B represents *AB*.



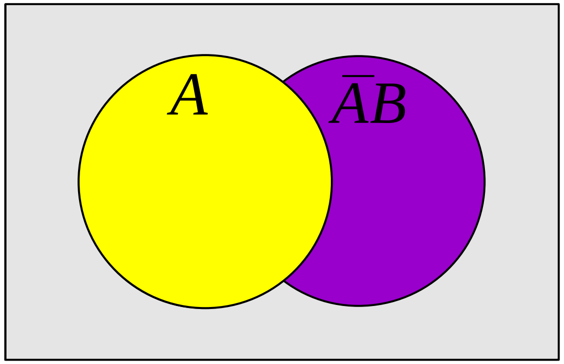
The diagram visually shows that $A + AB = A$. Other rules can be illustrated with the diagrams as well.

Rules of Boolean Algebra

Example Illustrate the rule $A + \overline{A}B = A + B$ with a Venn diagram.

Solution

This time, $\overline{A}$ is represented by the blue area and B again by the red circle. The intersection represents $\overline{A}B$. Notice that $A + \overline{A}B = A + B$



Rules of Boolean Algebra

Rule 12, which states that $(A + B)(A + C) = A + BC$, can be proven by applying earlier rules as follows:

$$
\begin{aligned}
(A + B)(A + C) &= AA + AC + AB + BC \\
&= A + AC + AB + BC \\
&= A(1 + C + B) + BC \\
&= A \cdot 1 + BC \\
&= A + BC
\end{aligned}
$$

This rule is a little more complicated, but it can also be shown with a Venn diagram, as given on the following slide…

Rules of Boolean Algebra

Three areas represent the variables *A*, *B*, and *C*.

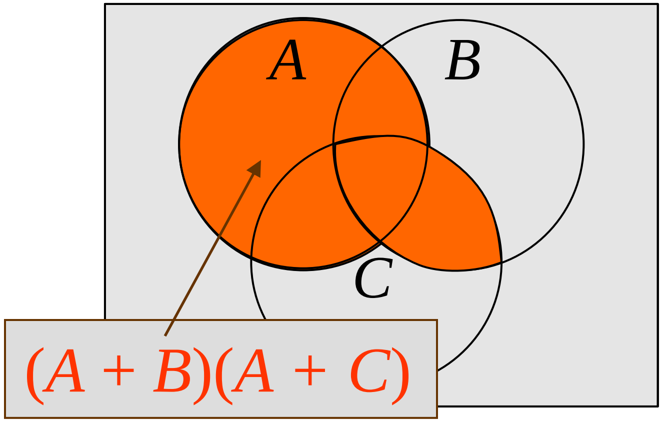
The area representing *A* + *B* is shown in yellow.

The area representing *A* + *C* is shown in red.

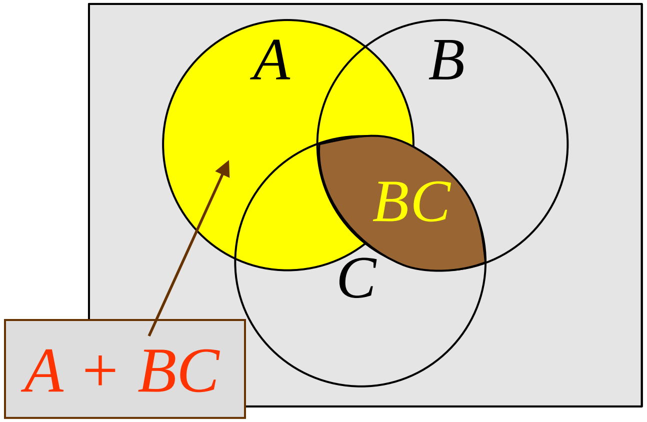
The overlap of red and yellow is shown in orange.

The overlapping area between *B* and C represents *BC*.

ORing with *A* gives the same area as before.



=



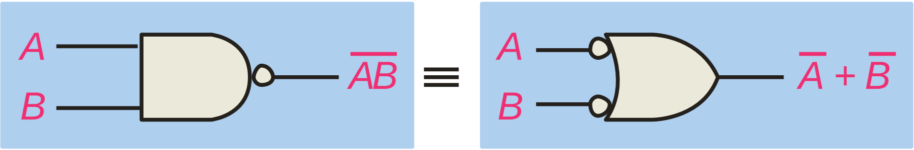
DeMorgan’s Theorem

DeMorgan’s 1^{st} Theorem

The complement of a product of variables is equal to the sum of the complemented variables.

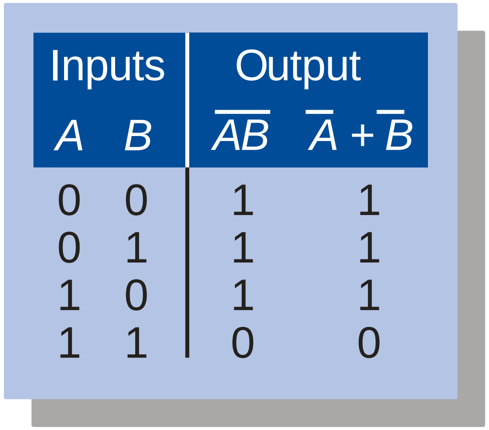
$$\overline{AB} = \overline{A} + \overline{B}$$

Applying DeMorgan’s first theorem to gates:



NAND

Negative-OR



Inputs		Output	
A	B	$\overline{AB}$	$\overline{A} + \overline{B}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

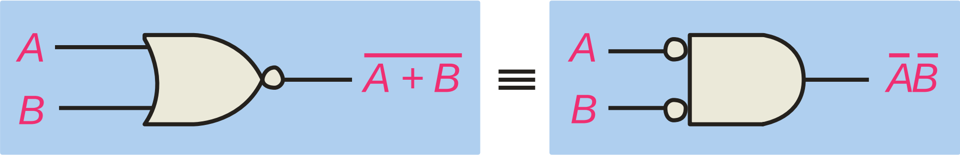
DeMorgan’s Theorem

DeMorgan’s 2nd Theorem

The complement of a sum of variables is equal to the product of the complemented variables.

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

Applying DeMorgan’s second theorem to gates:



NOR

Negative-AND

Inputs		Output	
A	B	$\overline{A + B}$	$\overline{A}\overline{B}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

DeMorgan’s Theorem

Example Apply DeMorgan’s theorem to remove the overbar covering both terms from the expression $X = \overline{\overline{C} + D}$.

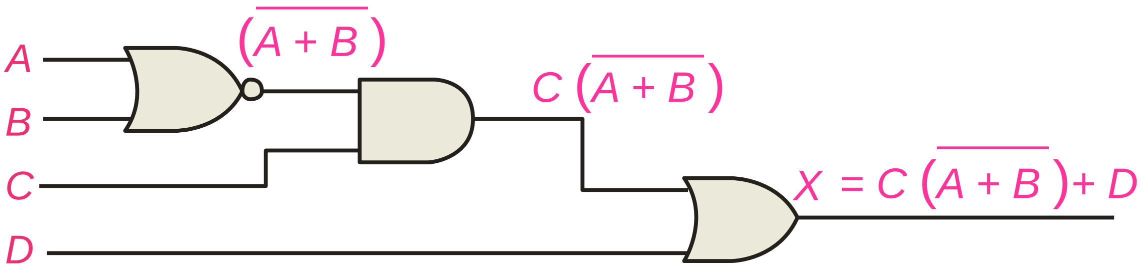
Solution To apply DeMorgan’s theorem to the expression, you can break the overbar covering both terms and change the sign between the terms. This results in $X = \overline{\overline{C}} \cdot \overline{D}$. Deleting the double bar gives $X = C \cdot \overline{D}$.

Boolean Analysis of Logic Circuits

Combinational logic circuits can be analyzed by writing the expression for each gate and combining the expressions according to the rules for Boolean algebra.

Example Apply Boolean algebra to derive the expression for *X*.

Solution Write the expression for each gate:



Applying DeMorgan’s theorem and the distribution law:

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

SOP and POS forms

Boolean expressions can be written in the **sum-of-products** form (**SOP**) or in the **product-of-sums** form (**POS**). These forms can simplify the implementation of combinational logic.

An expression is in SOP form when two or more product terms are summed as in the following examples:

$\overline{A}\,\overline{B}\,\overline{C} + A\,B$ $\quad$ $A\,B\,\overline{C} + \overline{C}\,\overline{D}$ $\quad$ $C\,D + \overline{E}$

An expression is in POS form when two or more sum terms are multiplied as in the following examples:

$(A + B)(\overline{A} + C)$ $\quad$ $(A + B + \overline{C})(B + D)$ $\quad$ $(\overline{A} + B)C$

SOP Standard form

In **SOP standard form**, every variable in the domain must appear in each term. This form is useful for constructing truth tables or for implementing logic in PLDs.

You can expand a nonstandard term to standard form by multiplying the term by a term consisting of the sum of the missing variable and its complement.

Example

Convert $X = \overline{A}\,\overline{B} + ABC$ to standard form.

Solution

The first term does not include the variable C. Therefore, multiply it by the $(C + \overline{C})$, which = 1:

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

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

POS Standard form

In **POS standard form**, every variable in the domain must appear in each sum term of the expression.

You can expand a nonstandard POS expression to standard form by adding the product of the missing variable and its complement and applying rule 12, which states that $(A + B)(A + C) = A + BC$.

Example

Convert $X = (\overline{A} + \overline{B})(A + B + C)$ to standard form.

Solution

The first sum term does not include the variable C. Therefore, add $C\,\overline{C}$ and expand the result by rule 12.

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

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

DNF, CNF

DNF = SOP
CNF = POS

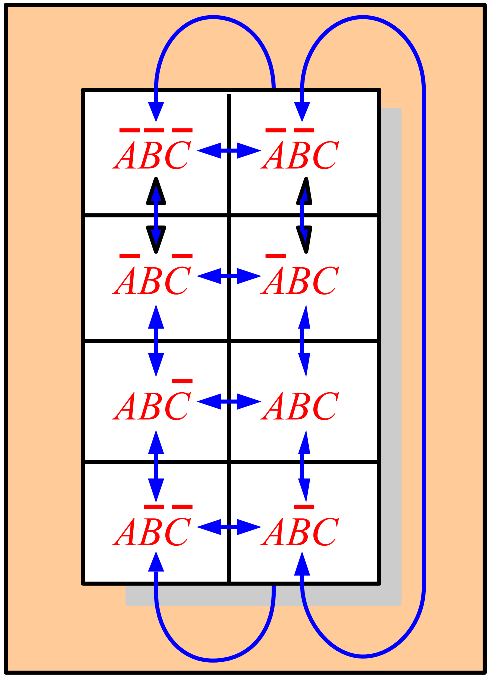
Karnaugh maps

The Karnaugh map (K-map) is a tool for simplifying combinational logic usually with 3 or 4 variables. For 3 variables, 8 cells are required (2^3).

The map shown is for three variables labeled *A*, *B*, and *C*.

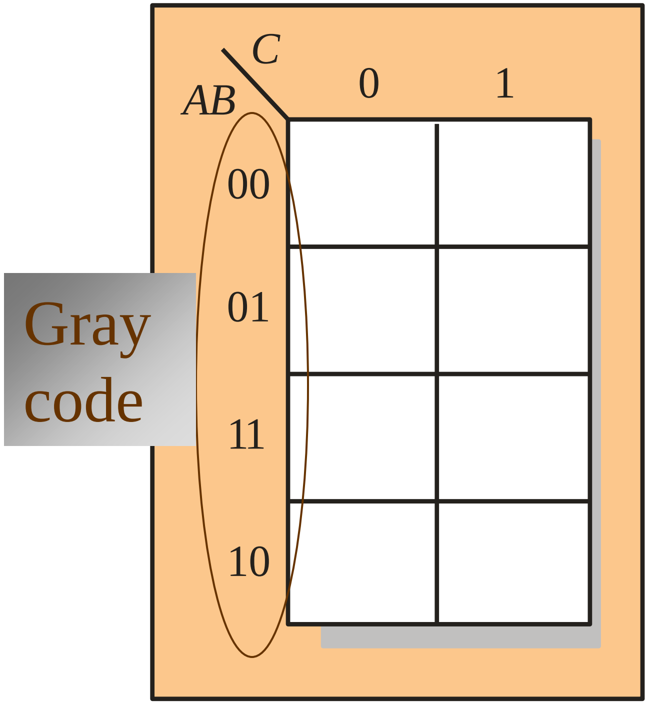
Each cell represents one possible product term.

Each cell differs from an adjacent cell by only one variable.


$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$AB\overline{C}$
ABC
$A\overline{B}\,\overline{C}$
$A\overline{B}C$

Karnaugh maps

Cells are usually labeled using 0's and 1's to represent the variable and its complement.



The numbers are entered in gray code, to force adjacent cells to be different by only one variable.

Ones are read as the true variable and zeros are read as the complemented variable.

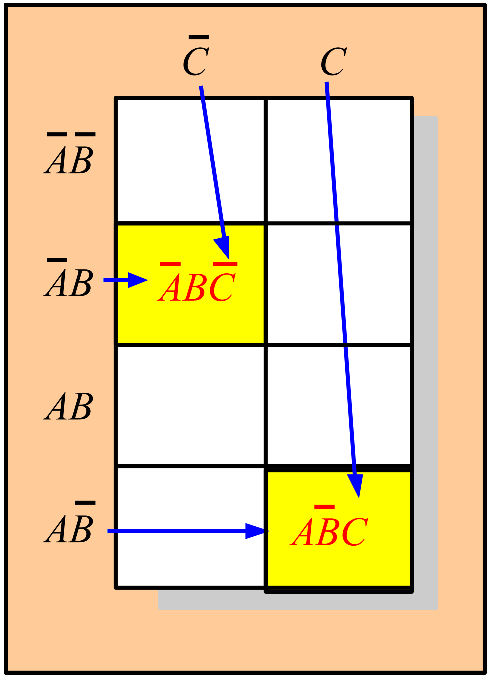
Karnaugh maps

Alternatively, cells can be labeled with the variable letters. This makes it simple to read, but it takes more time preparing the map.

Example Read the terms for the yellow cells.

Solution

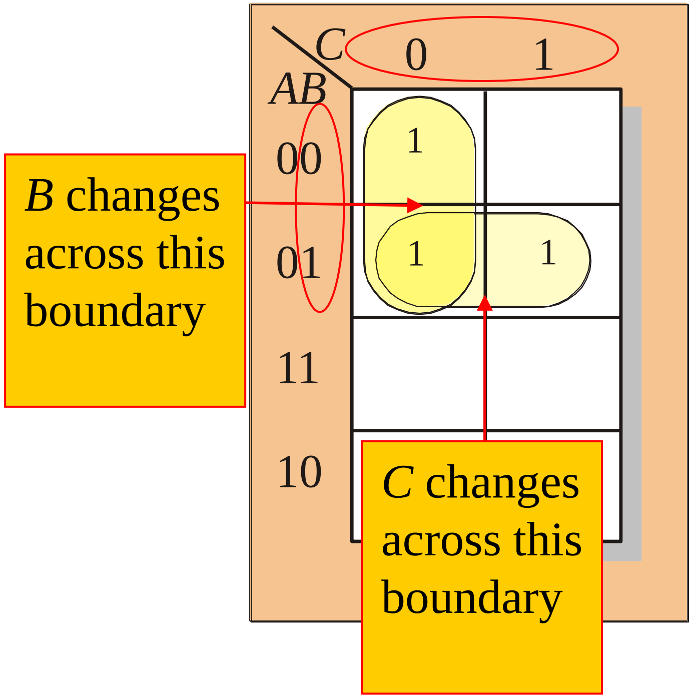
The cells are $\overline{A}B\overline{C}$ and $A\overline{B}C$.



Karnaugh maps

K-maps can simplify combinational logic by grouping cells(powers of 2) and eliminating variables that change.

Example Group the 1's on the map and get the minimum logic.



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

B changes across this boundary

C changes across this boundary

Solution

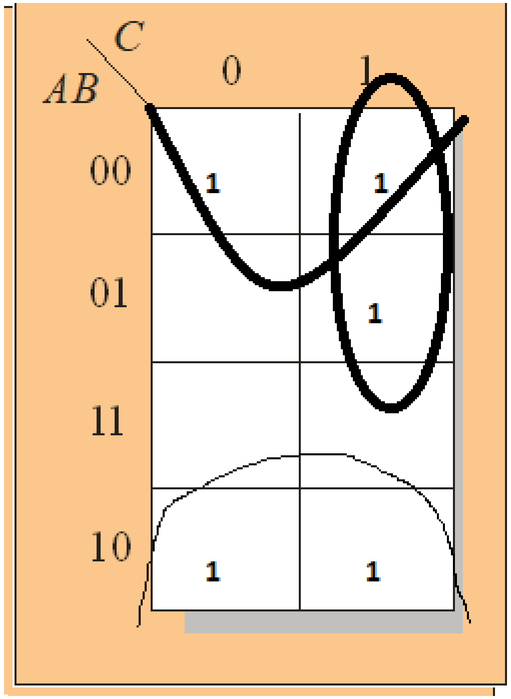
1. Group the 1's into two overlapping groups as indicated.
2. Read each group by eliminating any variable that changes across a boundary.
3. The vertical group is read $\overline{A}\,\overline{C}$.
4. The horizontal group is read $\overline{A}B$.

$$X = \overline{A}\,\overline{C} + \overline{A}B$$

Karnaugh maps

$$X = A\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,\overline{B}\,C + \overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C}$$

Example



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

Solution

1. Group the 1’s. (One with 4 cells, second with 2 cells)
2. Read each group by eliminating any variable that changes across a boundary.

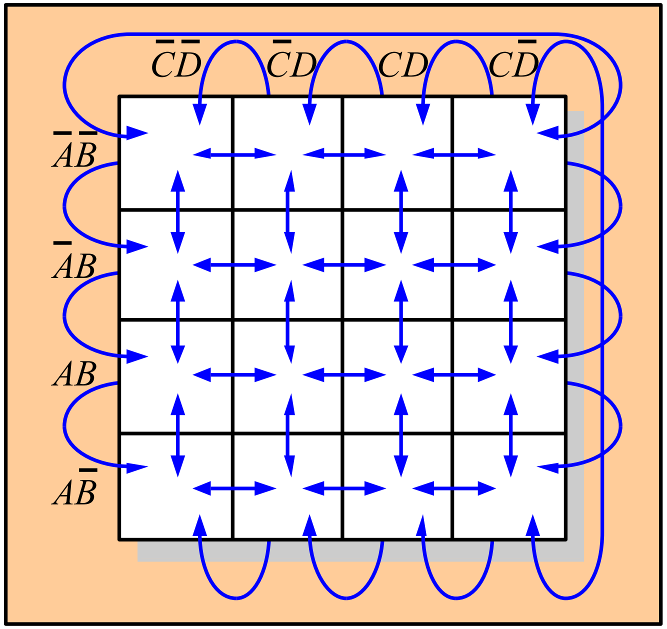
$$X = \overline{B} + \overline{A}\,C$$

3 variable K-map

yz *x*	00	01	11	10
0	0	1	3	2
1	4	5	7	6

Karnaugh maps

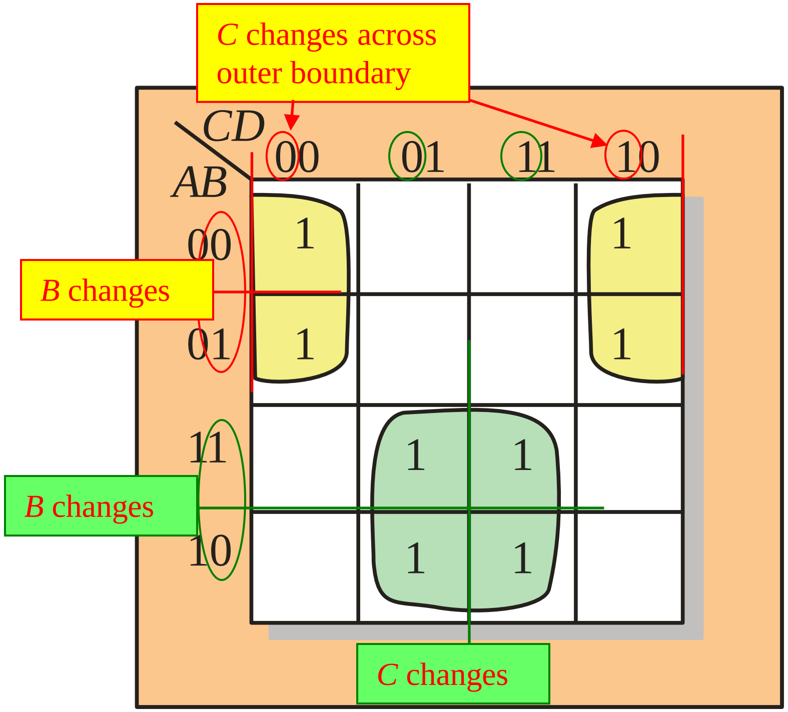
A 4-variable map has an adjacent cell on each of its four boundaries as shown.



Each cell is different only by one variable from an adjacent cell.

Karnaugh maps

Example Group the 1’s on the map and read the minimum logic.



C changes across outer boundary

B changes

B changes

C changes

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

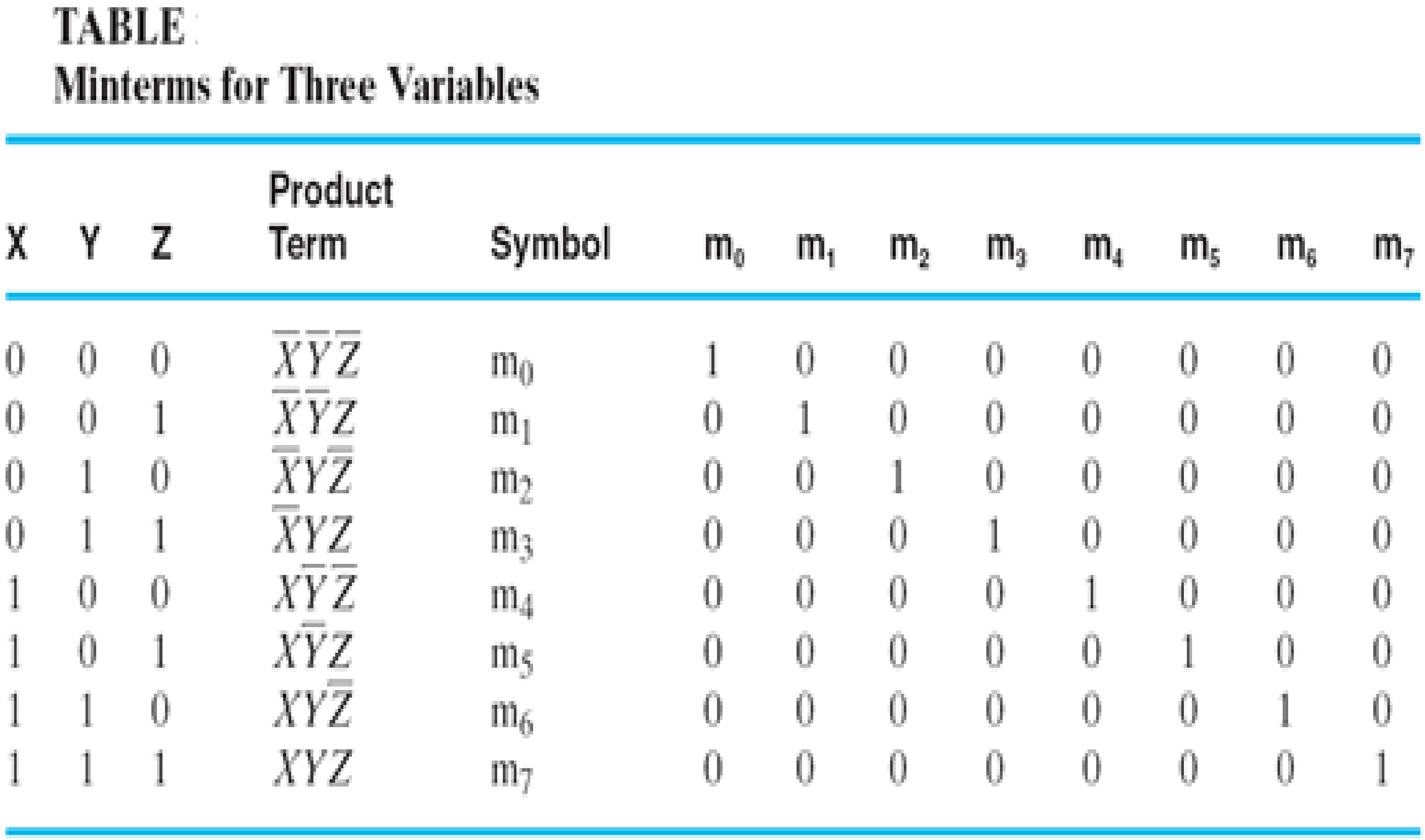
X

Solution

1. Group the 1’s into two separate groups as indicated.
2. Read each group by eliminating any variable that changes across a boundary.
3. The upper (yellow) group is read as $\overline{A}\overline{D}$.
4. The lower (green) group is read as AD.

$$X = \overline{A}\overline{D} + AD$$

Minterms for three variables



TABLE

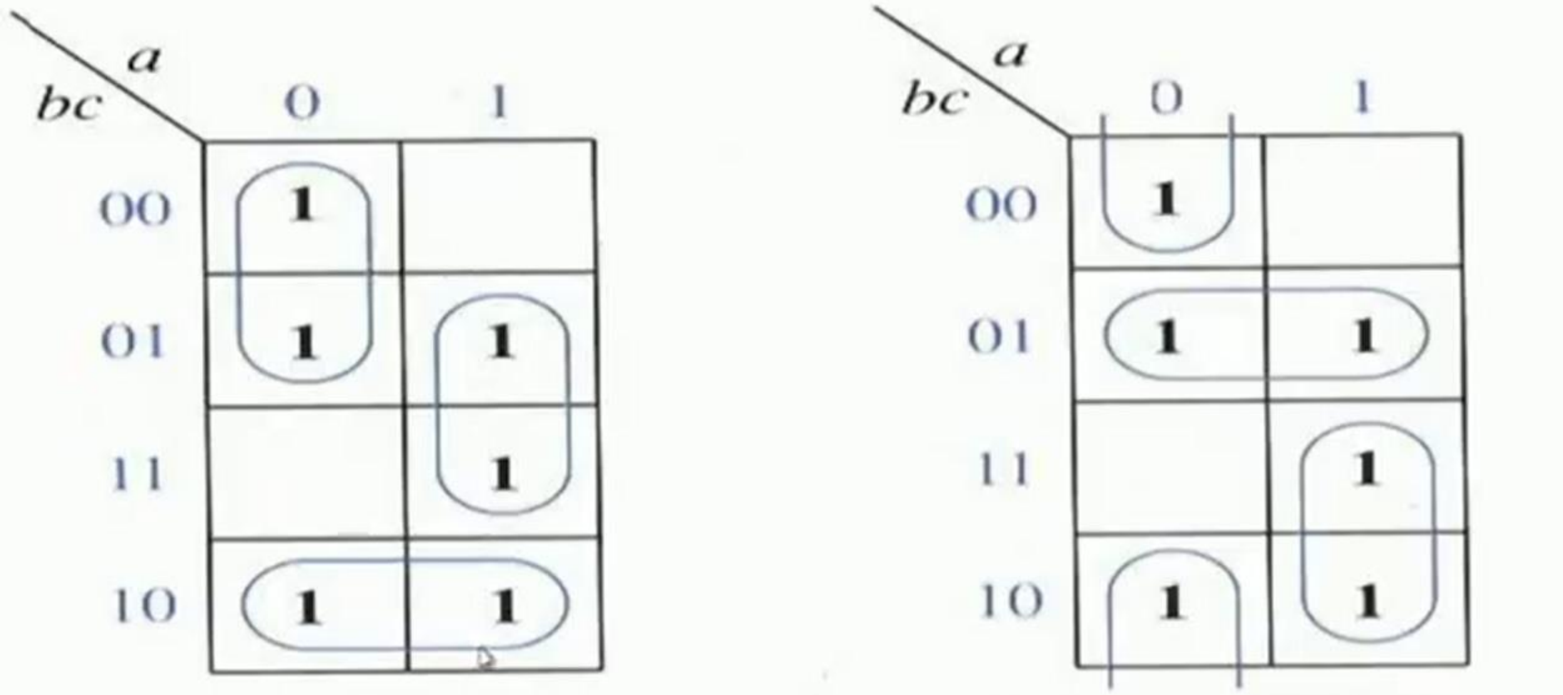
Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\,\overline{Y}\,\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\,\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}Y\overline{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\,\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

Boolean Expression in Minterms

- F= Σm(0,1,2,5,6,7)

$F = \sum m(0,1,2,5,6,7)$



bc \ a	0	1
00	1	
01	1	1
11		1
10	1	1

$F = a'b' + bc' + ac$

bc \ a	0	1
00	1	
01	1	1
11		1
10	1	1

$F = a'c' + b'c + ab$

Maxterms for three variables

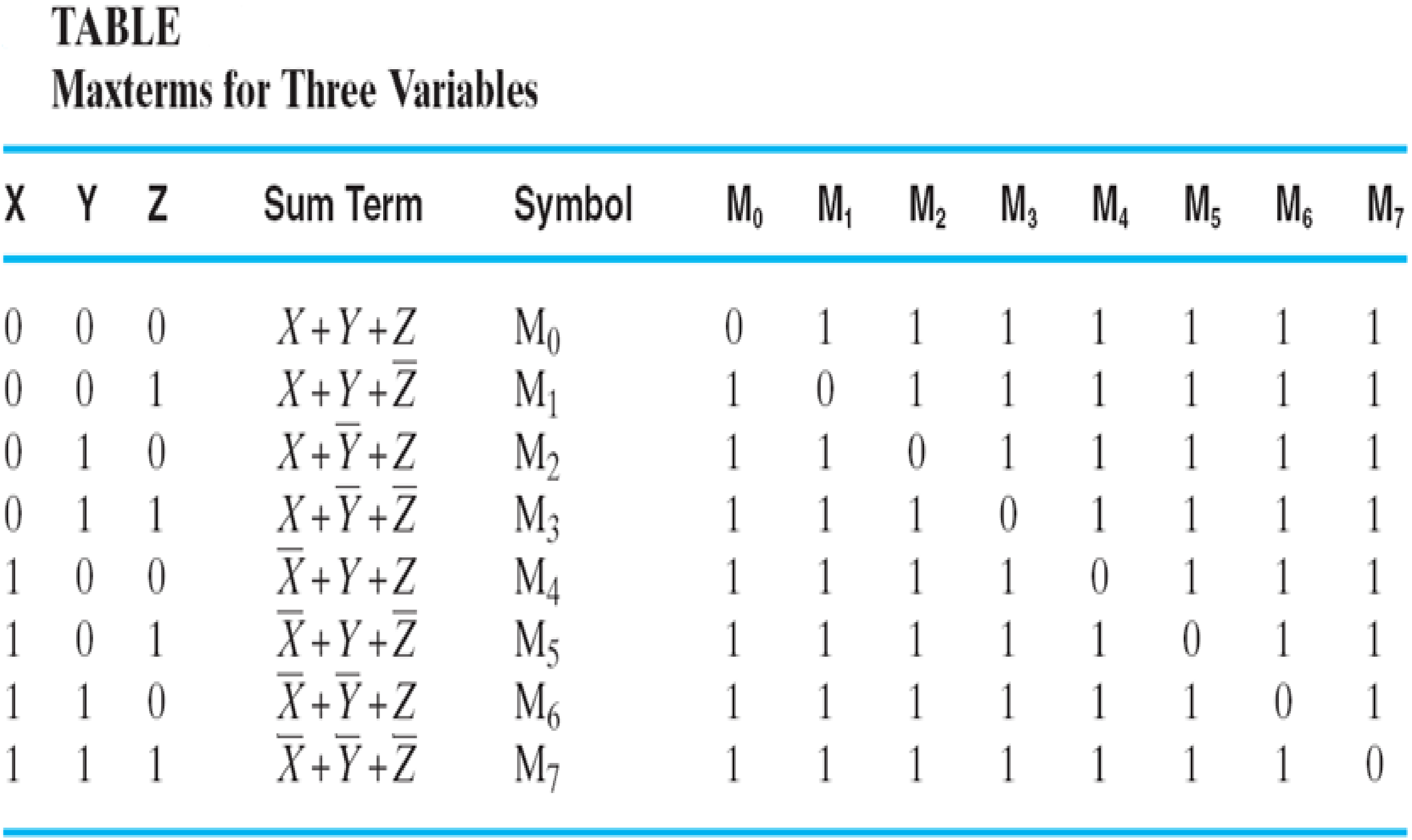


TABLE
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X+Y+Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X+Y+\overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X+\overline{Y}+Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X+\overline{Y}+\overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X}+Y+Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X}+Y+\overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X}+\overline{Y}+Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X}+\overline{Y}+\overline{Z}$	M_7	1	1	1	1	1	1	1	0

Fixed Point Representation-8 bit

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
				.				

If 16 ---- overflow (10000)

If 2^{-5} ----overflow

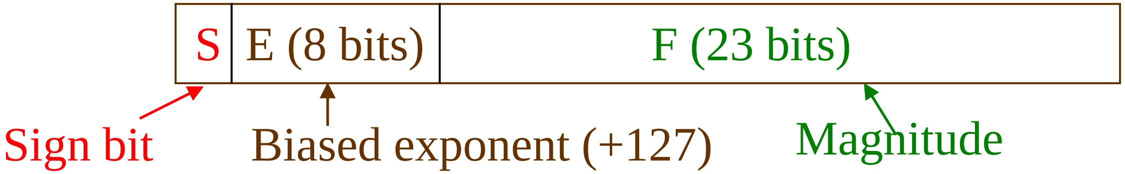
Floating Point Numbers

Floating point notation is capable of representing very large or small numbers by using a form of scientific notation.

Single : 32 bits [1 sign , 8 exponent, 23 mantissa]

Double : 64 bits [1 sign , 11 exponent, 52 mantissa]

A 32-bit single precision number is illustrated.



S

E (8 bits)

F (23 bits)

Sign bit

Biased exponent (+127)

Magnitude

Floating Point Numbers

Sign (1+mantissa) x $2^{\text{exponent-127}}$

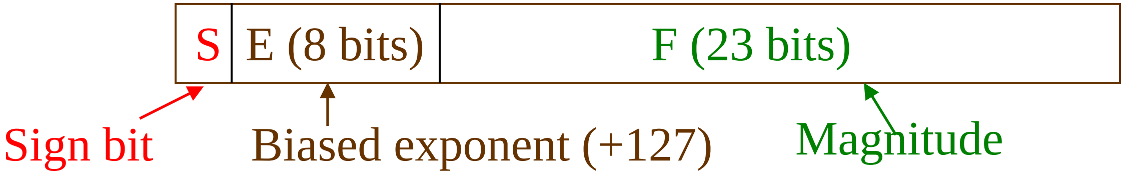
Eg:- 0 10000011 00010010000000000000000

Sign: 0 , + ve

Exponent: 10000011 $= 2^7 + 2^1 + 2^0 = 131$

Mantissa: $2^{-4} + 2^{-7} = (1/16) + (1/128) = 9/128$

Decimal: $+ (1+(9/128)) \times 2^{131-127} = 2192/128 = 17.125$



S | E (8 bits) | F (23 bits)

Sign bit

Biased exponent (+127)

Magnitude

Floating Point Numbers

Sign $(1+\text{mantissa}) \times 2^{\text{exponent}-127}$

Eg:- 0 10000011 00010010000000000000000

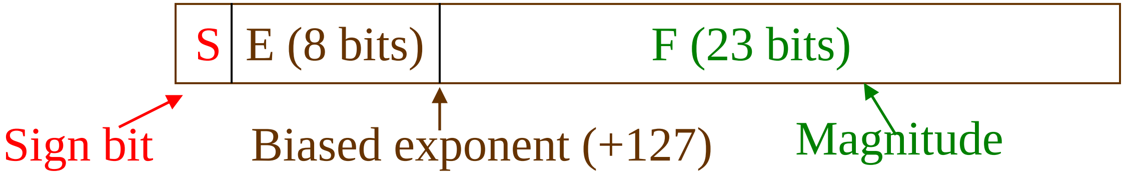
Final value : 17.125

10001 . 001

10001.001×2^{0}

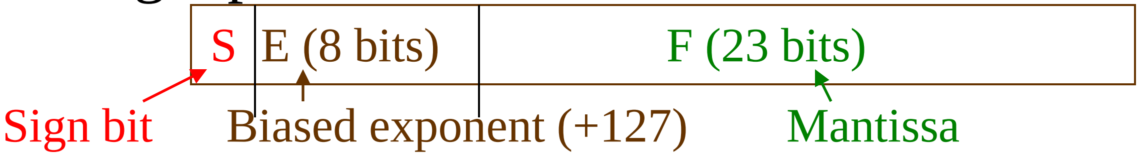
Binary: $+\ 1.0001001 \times 2^{4}$

exponent = 4+127 =131 = 10000011



Floating Point Numbers

Floating point notation is capable of representing very large or small numbers by using a form of scientific notation. A 32-bit single precision number is illustrated.



S | E (8 bits) | F (23 bits)

Sign bit

Biased exponent (+127)

Mantissa

Sign(1+Mantissa) x $2^{\text{exponent-127}}$

It doesn't have a fixed decimal point somewhere within the bit string (use exponent)

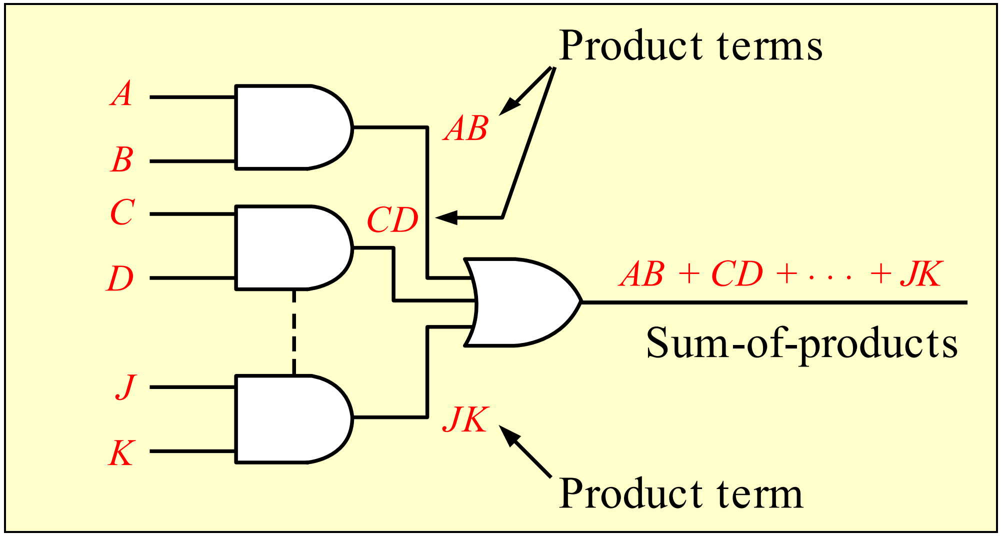
Syllabus

Module I (11 Hours)

- Representation of signed numbers – 1's complement and 2’s complement , Logic gates – AND - OR – NOT - NAND- NOR - XOR , Boolean algebra - Basic laws and theorems , Boolean functions - truth table, Standard forms of Boolean Expressions – Sum of Products and Product of Sums - minimization of Boolean function using Karnaugh map method - Realization using logic gates, Floating point numbers.
- Combinational Circuits - Half adder - Full Adder- Decoder - Encoder- Multiplexer – Demultiplexer

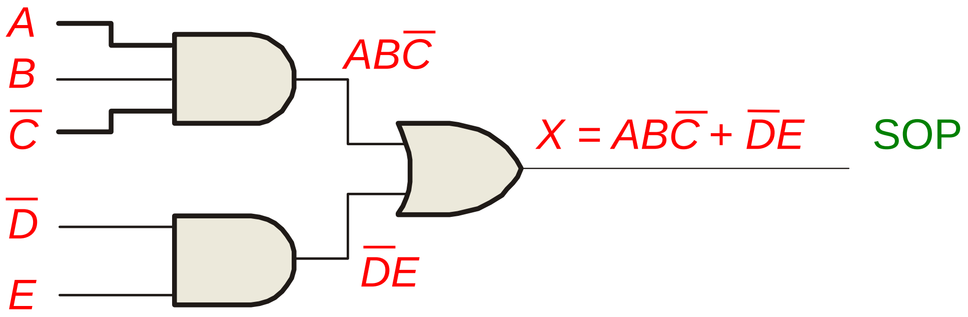
Combinational Logic Circuits

In Sum-of-Products (SOP) form, basic combinational circuits can be directly implemented with AND-OR combinations if the necessary complement terms are available.



Combinational Logic Circuits

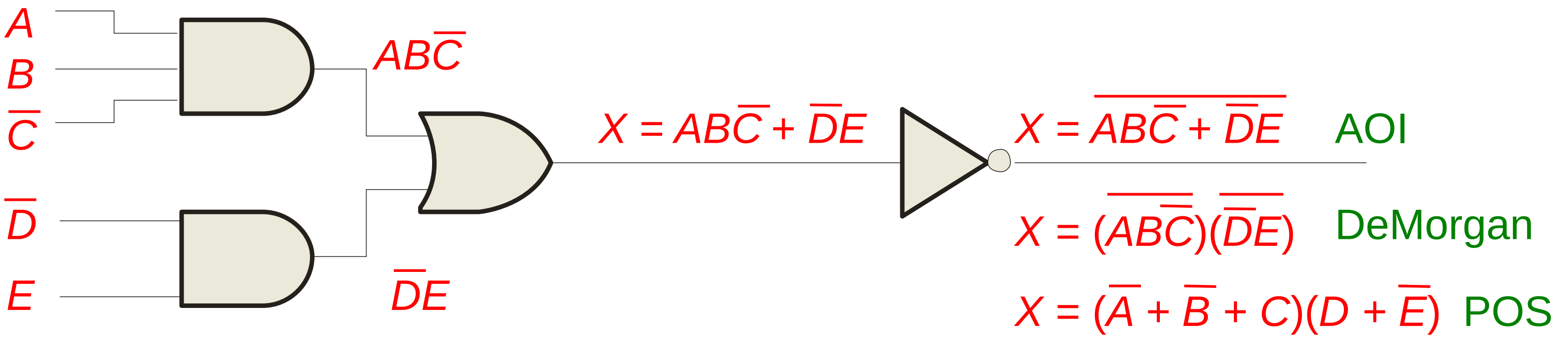
An example of an SOP implementation is shown. The SOP expression is an AND-OR combination of the input variables and the appropriate complements.



Combinational Logic Circuits

When the output of a SOP form is inverted, the circuit is called an AND-OR-Invert(AOI) circuit. The AOI configuration lends itself to product-of-sums (POS) implementation.

An example of an AOI implementation is shown. The output expression can be changed to a POS expression by applying DeMorgan’s theorem.



$X = AB\overline{C} + \overline{D}E$

$X = \overline{AB\overline{C} + \overline{D}E}$ AOI

$X = (\overline{AB\overline{C}})(\overline{\overline{D}E})$ DeMorgan

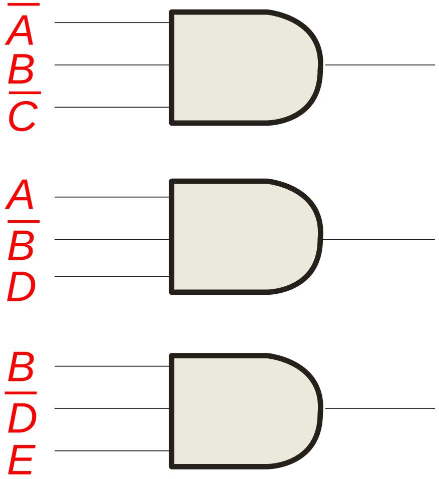
$X = (\overline{A} + \overline{B} + C)(D + \overline{E})$ POS

Implementing Combinational Logic

Implementing a SOP expression is done by first forming the AND terms; then the terms are ORed together.

Example Show the circuit that will implement the Boolean expression $X = \overline{A}B\overline{C} + A\overline{B}D + B\overline{D}E$. (Assume that the variables and their complements are available.)

Solution Start by forming the terms using three 3-input AND gates. Then combine the three terms using a 3-input OR gate.



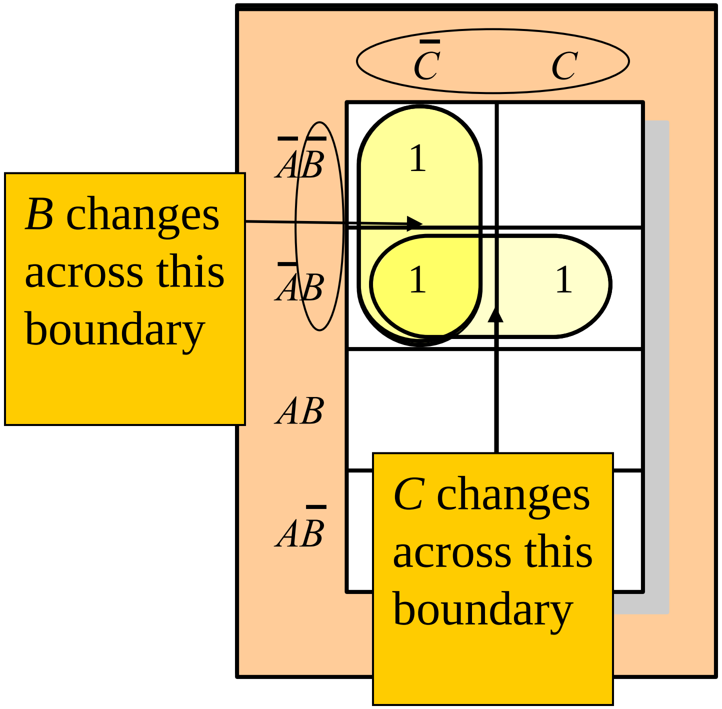
$X = \overline{A}B\overline{C} + A\overline{B}D + B\overline{D}E$

Karnaugh Map Implementation

For basic combinational logic circuits, the Karnaugh map can be read and the circuit drawn as a minimum SOP.

Example

A Karnaugh map is drawn from a truth table. Read the minimum SOP expression and draw the circuit.



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

B changes across this boundary

C changes across this boundary

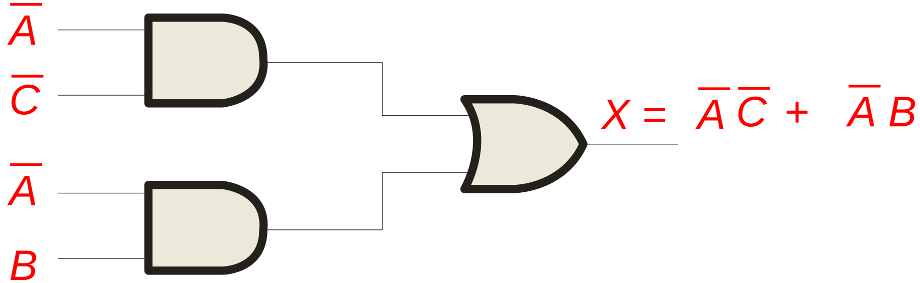
Solution

1. Group the 1’s into two overlapping groups as indicated.
2. Read each group by eliminating any variable that changes across a boundary.
3. The vertical group is read $\overline{A}\,\overline{C}$.
4. The horizontal group is read $\overline{A}B$.

The circuit is on the next slide:

Solution *continued…*

Circuit:



The result is shown as a sum of products.

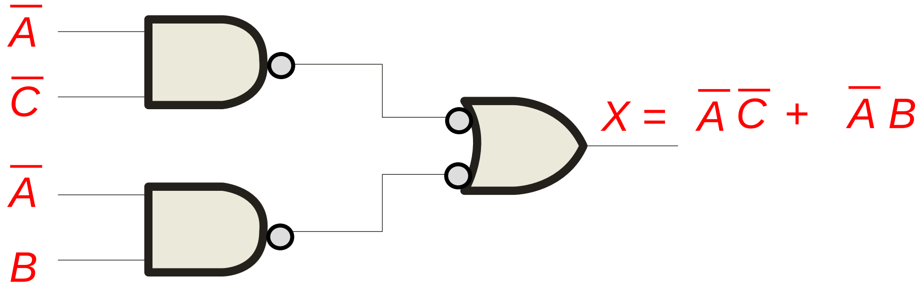
It is a simple matter to implement this form using only NAND gates.

NAND Logic

Example Convert the circuit in the previous example to one that uses only NAND gates.

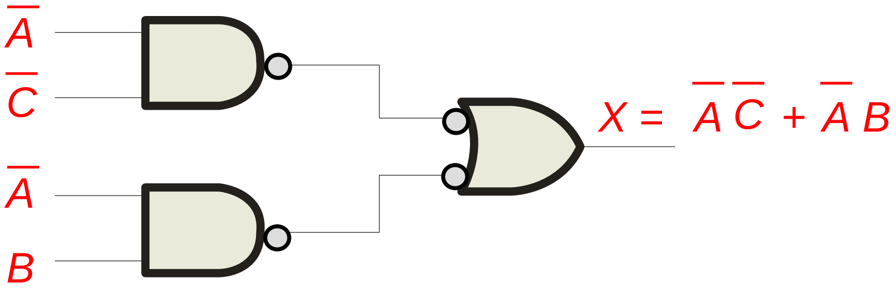
Solution

Recall from Boolean algebra that double inversion cancels. By adding inverting bubbles to above circuit, it is easily converted to NAND gates:



NAND Logic

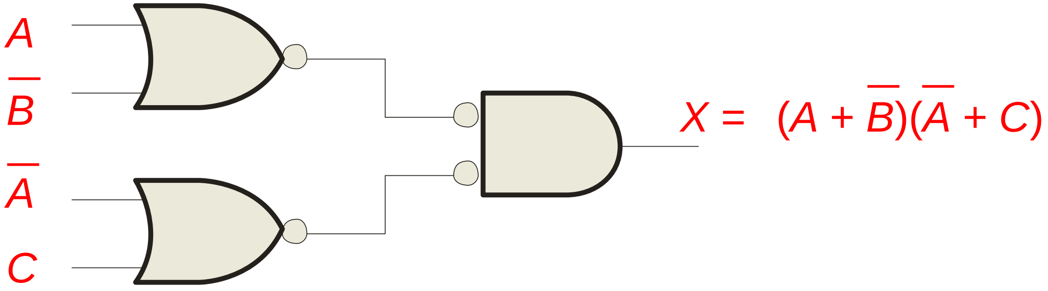
Recall from DeMorgan’s theorem that $\overline{AB} = \overline{A} + \overline{B}$. By using equivalent symbols, it is simpler to read the logic of SOP forms. The earlier example shows the idea:



The logic is easy to read if you cancel the two connected bubbles on a line.

NOR Logic

Alternatively, DeMorgan’s theorem can be written as $\overline{A + B} = \overline{A}\,\overline{B}$. By using equivalent symbols, it is simpler to read the logic of POS forms. For example,



Again, the logic is easy to read if you cancel the two connected bubbles on a line.

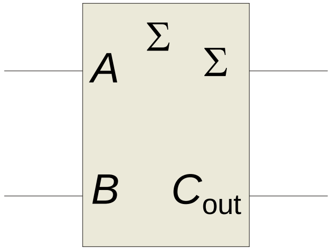
Half-Adder

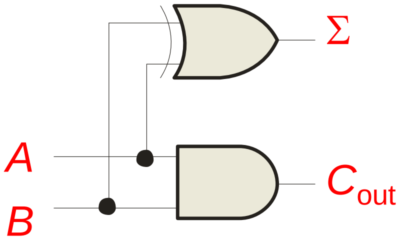
Basic rules of binary addition are performed by a **half adder**, which has two binary inputs (*A* and *B*) and two binary outputs (Carry out and Sum).

The inputs and outputs can be summarized on a truth table.

Inputs		Outputs	
A	*B*	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The logic symbol and equivalent circuit are:





$$\Sigma = A \oplus B$$

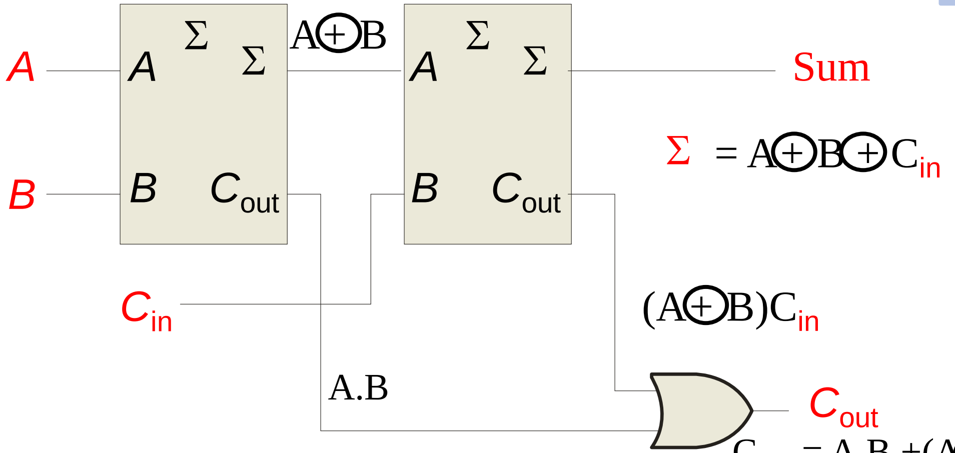
$$C_{out} = A.B$$

Full-Adder

A **full adder** has three binary inputs (*A*, *B*, and Carry in[C_{in}]) and two binary outputs (Carry out [C_{out}] and Sum). The truth table summarizes the operation.

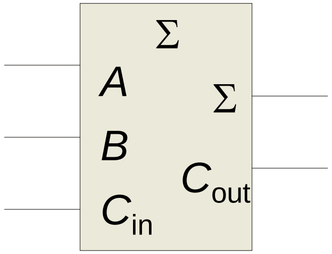
A full-adder can be constructed from two half adders as shown:

Inputs			Outputs	
A	B	C_{in}	C_{out}	Σ
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



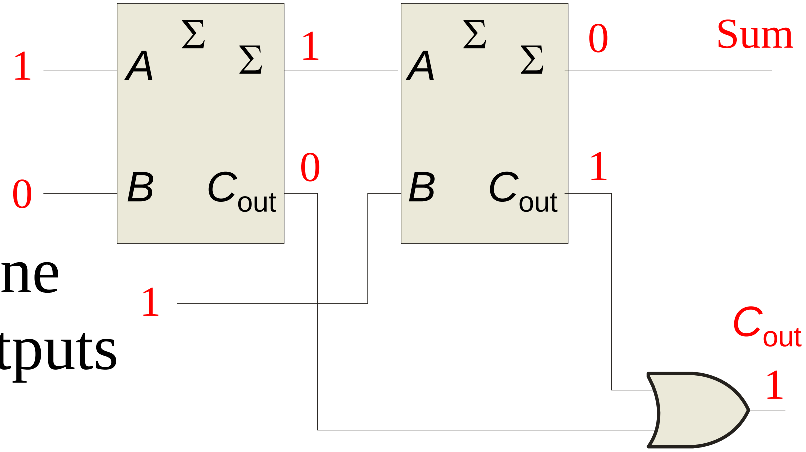
$\Sigma = A \oplus B \oplus C_{in}$

$C_{out} = A.B + (A \oplus B).C_{in}$



Symbol

Full-Adder



Example

For the given inputs, determine the intermediate and final outputs of the full adder.

Solution

The first half-adder has inputs of 1 and 0; therefore the Sum =1 and the Carry out = 0.

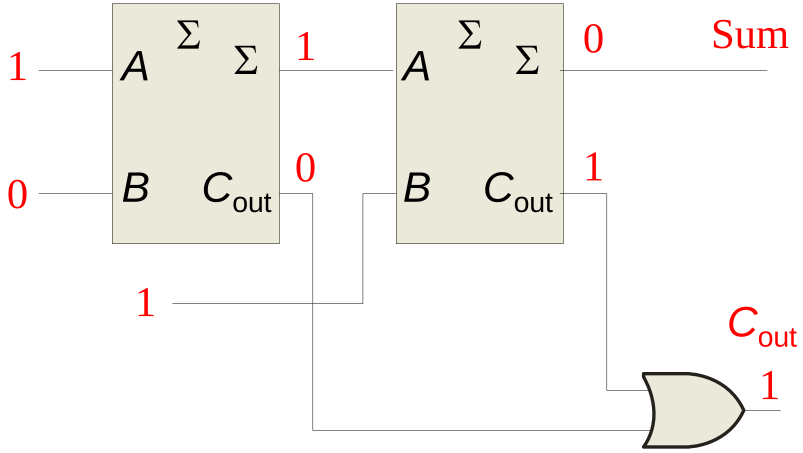
The second half-adder has inputs of 1 and 1; therefore the Sum = 0 and the Carry out = 1.

The OR gate has inputs of 1 and 0, therefore the final carry out = 1.

Full-Adder

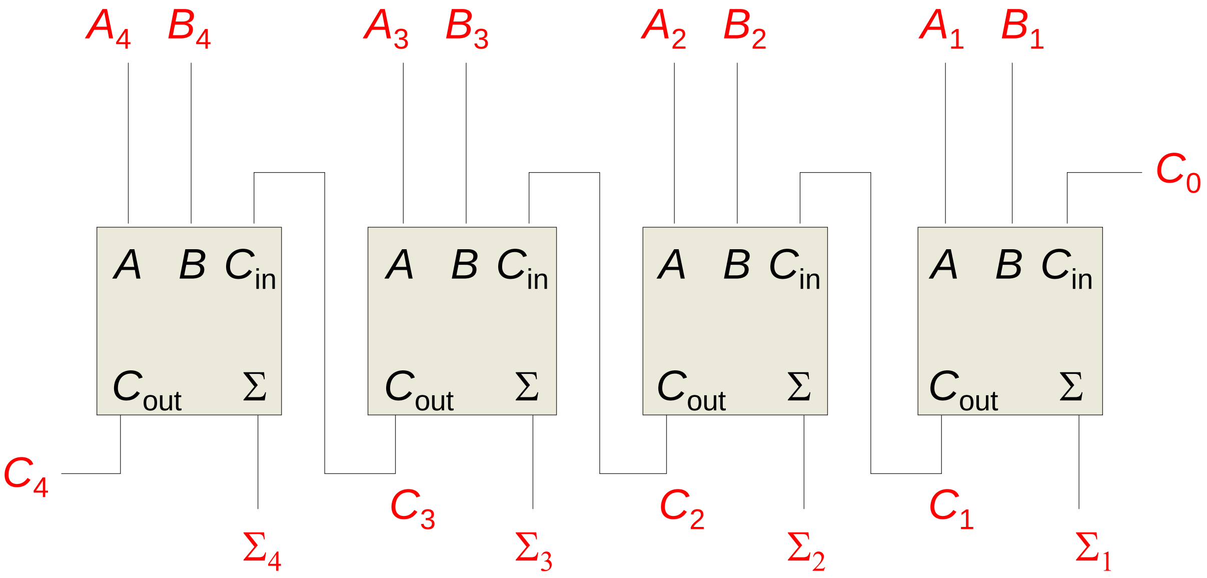
Notice that the result from the previous example can be read directly on the truth table for a full adder.

Inputs			Outputs	
A	B	C_{in}	C_{out}	Σ
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



Parallel Adders

Full adders are combined into parallel adders that can add binary numbers with multiple bits. A 4-bit adder is shown.

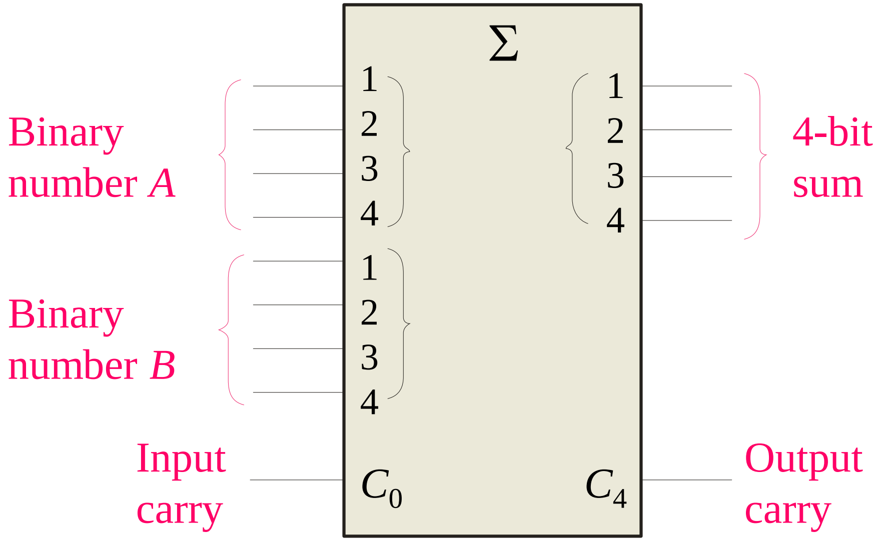


The output carry (C_4) is not ready until it propagates through all of the full adders. This is called *ripple carry*, delaying the addition process.

LOOK AHEAD CARRY ADDER – Carry Generation(A.B), Carry Propagation (A+B) , Cout = Cg + Cp . Cin

Parallel Adders

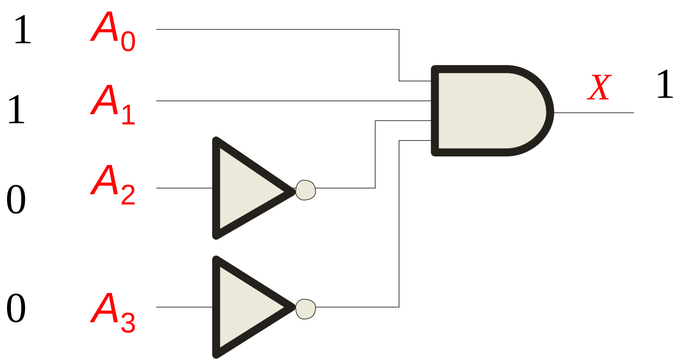
The logic symbol for a 4-bit parallel adder is shown. This 4-bit adder includes a carry in (labeled (C_0) and a Carry out (labeled C_4).



Decoders

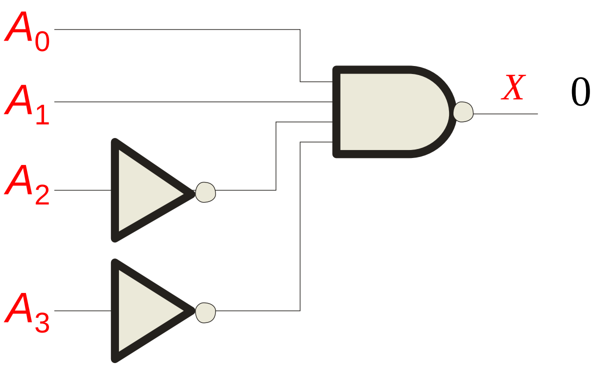
A **decoder** is a logic circuit that detects the presence of a specific combination of bits at its input.

Two simple decoders that detect the presence of the binary code 0011 are shown. The first has an active HIGH output; the second has an active LOW output.



Active HIGH decoder for 0 0 1 1

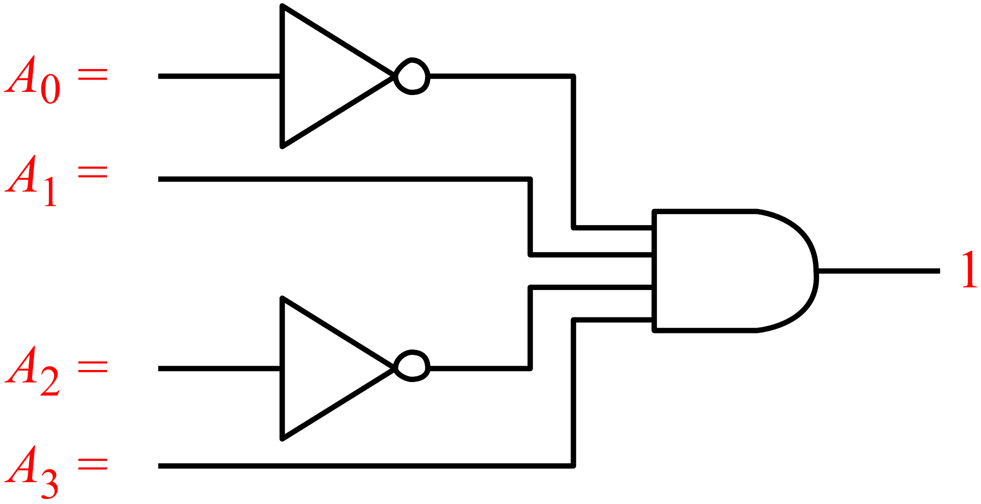
$A_3A_2A_1A_0$



Active LOW decoder for 0011

Decoders

Question Assume the output of the decoder shown is a logic 1. What are the inputs to the decoder?

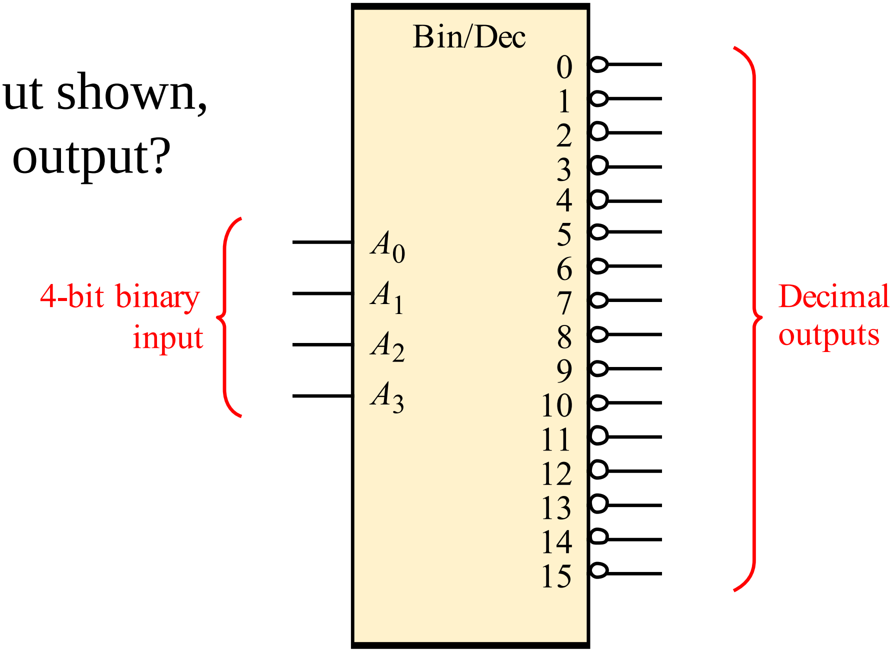


Decoders

IC decoders have multiple outputs to decode any combination of inputs. For example the binary-to-decimal decoder shown here has 16 outputs – one for each combination of binary inputs.

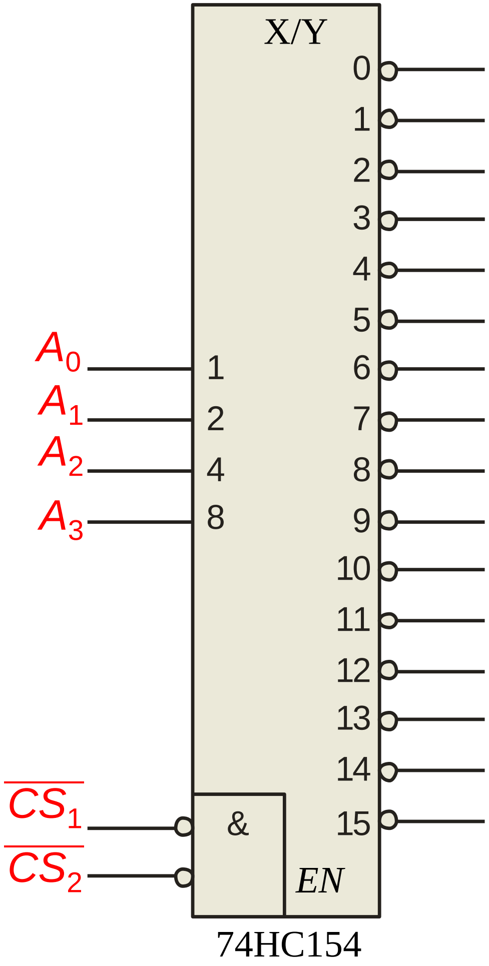
Question

For the input shown, what is the output?



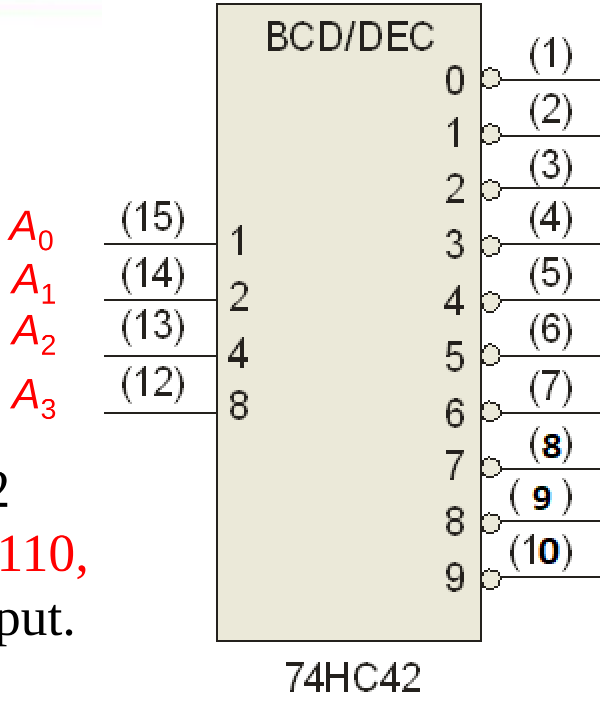
Decoders

A specific integrated circuit decoder is the 74HC154 (shown as a 4-to-16 decoder). It includes two active LOW chip select lines which must be at the active level to enable the outputs. These lines can be used to expand the decoder to larger inputs.



Decoders

BCD-to-decimal decoders accept a binary coded decimal input and activate one of ten possible decimal digit indications.



Example

Assume the inputs to the 74HC42 decoder are the sequence 0101, 0110, 0011, and 0010. Describe the output.

Solution

All lines are HIGH except for one active output, which is LOW. The active outputs are 5, 6, 3, and 2 in that order.

Encoders

An **encoder** accepts a
inputs and converts it
binary.

The decimal to BCD is
with an input for each
decimal digits and four o
represent the BCD code fo
digit.
The basic logic diagram
There is no zero input b
outputs are all LOW whe
is zero.

Decimal Digit	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

D = 8 + 9

C = 4 + 5 + 6 + 7

B = 2 + 3 + 6 + 7

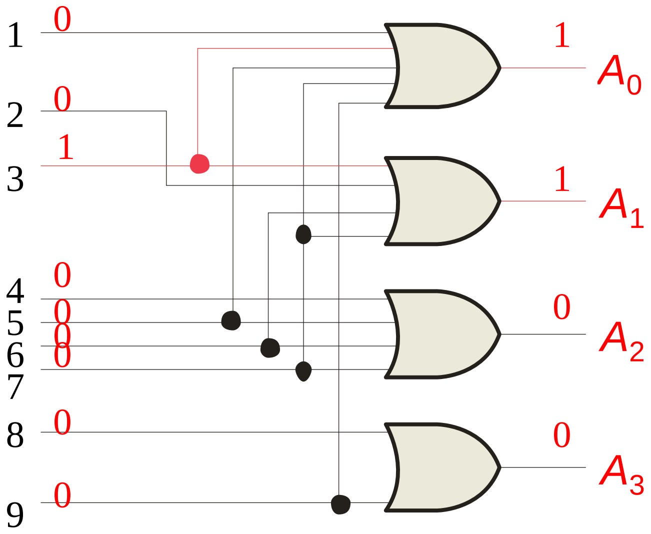
A = 1+3+5+7+9

A 0 digit input is not needed because the BCD outputs are all LOW when there are no HiGH inputs.

Encoders

Example Show how the decimal-to-BCD encoder converts the decimal number 3 into a BCD 0011.

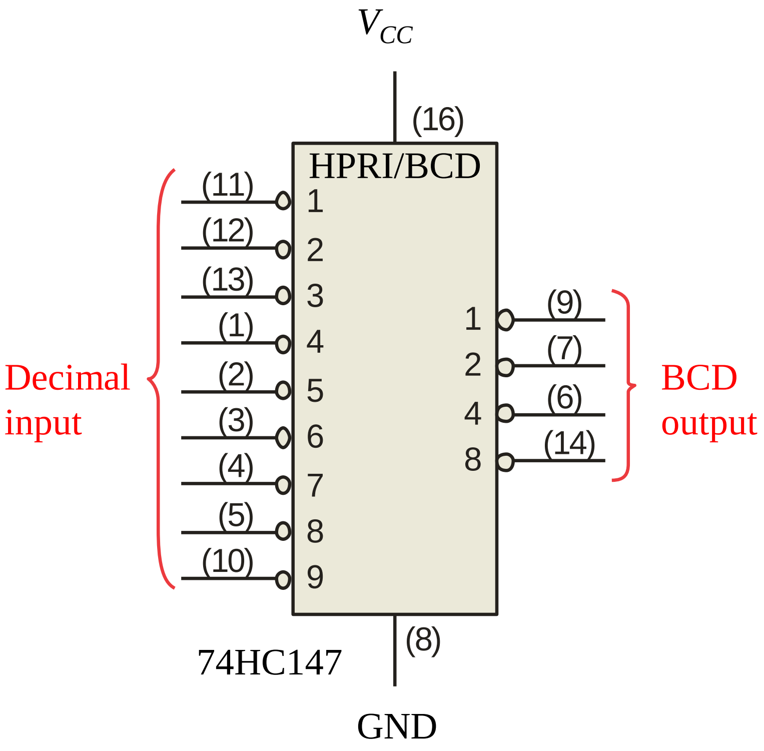
Solution The top two OR gates have ones as indicated with the red lines. Thus the output is 0011.



Encoders

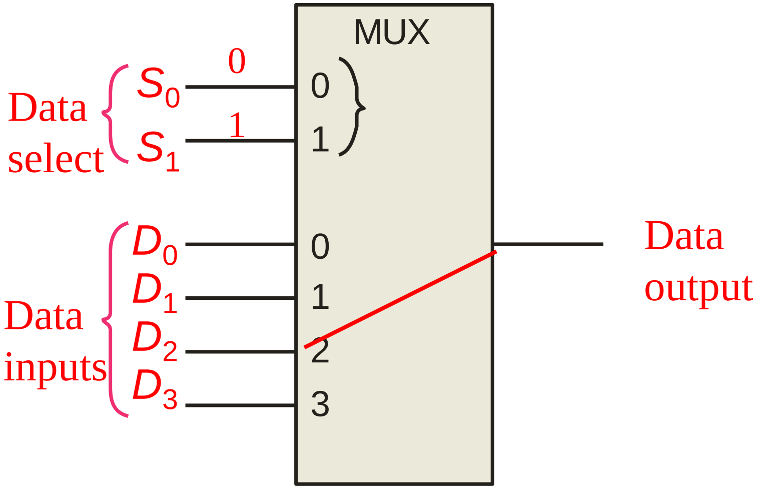
The 74HC147 is an example of an IC encoder. It is has active-LOW inputs and converts the active input to an active-LOW BCD output.

This device is offers additional flexibility in that it is a **priority encoder**. This means that if more than one input is active, the one with the highest order decimal digit will be active.



Multiplexers

A multiplexer (MUX) selects one data line from two or more input lines and routes data from the selected line to the output. The particular data line that is selected is determined by the select inputs.



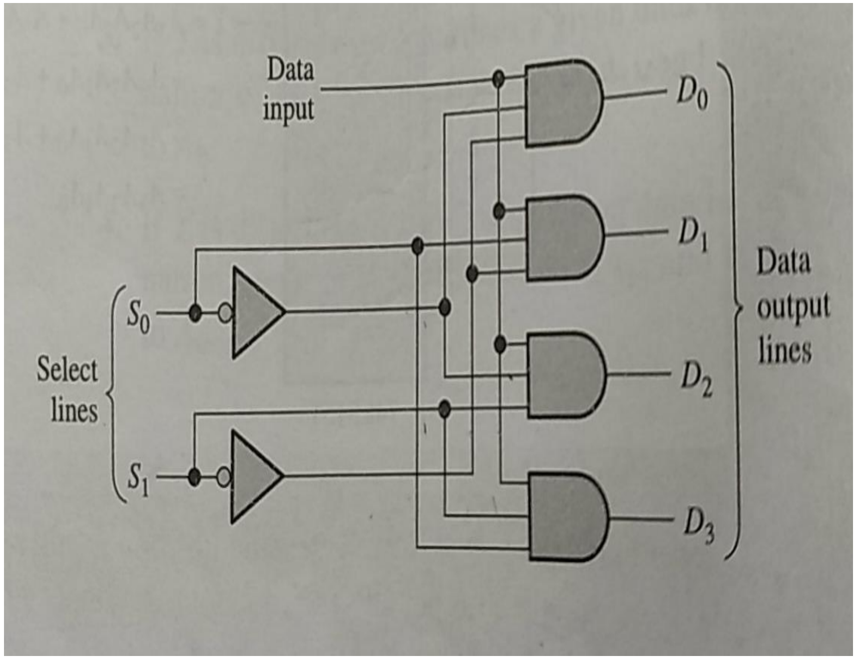
Two select lines are shown here to choose any of the four data inputs.

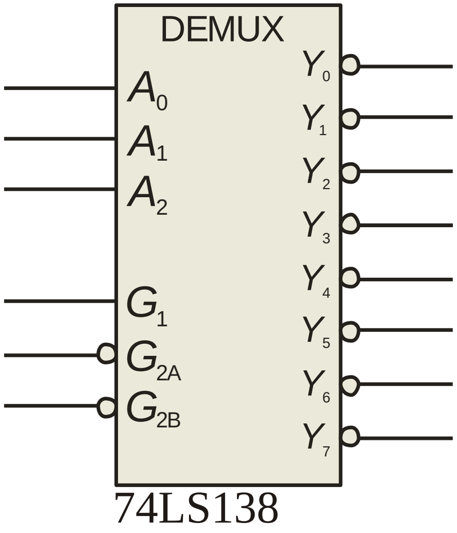
Question

Which data line is selected if $S_1S_0 = 10$? D_2

Demultiplexers

A demultiplexer (DEMUX) performs the opposite function from a MUX. It switches data from one input line to two or more data lines depending on the select inputs.





Data select lines

Enable inputs

Data outputs