Ollscoil na Gaillimhe
University of Galway

CT101 Computing Systems

Dr. Bharathi Raja Chakravarthi
Lecturer-above-the-bar
Email: bharathi.raja@universityofgalway.ie

University of Galway.ie

Ollscoil na Gaillimhe
University of Galway

Revision

University
ofGalway.ie

Introduction

- Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates.
- NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families.
- Rules and procedures have been developed for the conversion from Boolean functions given in terms of **AND, OR, and NOT** into equivalent **NAND and NOR** logic diagrams.

Binary numbers and others

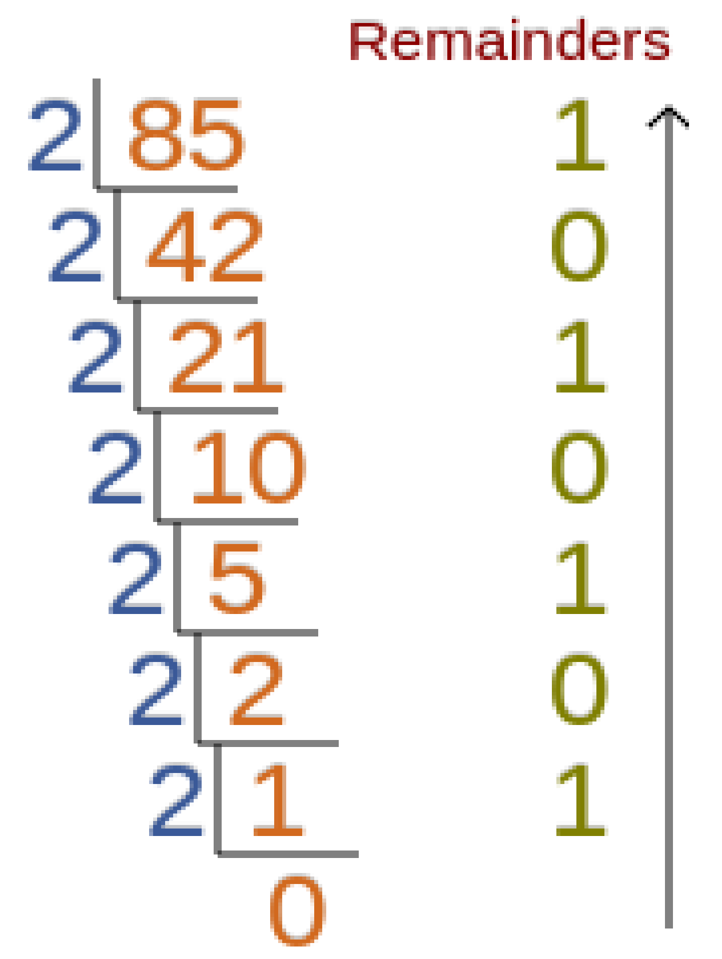
System	Radix	Allowable Digits
Binary	2	0 and 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	16	0 to 9 A, B, C, D, E, F

Decimal to Binary

STEP 1

WHOLE NUMBER PART

- We convert the whole number and fractional parts separately and then combine the results.
- The whole number part of **85.375** is **85**. Divide this number repeatedly by **2** until the quotient becomes **0**.



Write the remainders **from bottom to top**.

$(85)_{10} = (1010101)_2$

STEP 2

FRACTIONAL PART

The fractional part of **85.375** is **0.375**. Multiply the fractional part repeatedly by **2** until it becomes **0**.

- $0.375 \times 2 = 0.750$
- $0.750 \times 2 = 1.500$
- $0.500 \times 2 = 1.000$

STEP 3

From top to bottom, write the integer parts of the results to the fractional part of the number in base **2**.

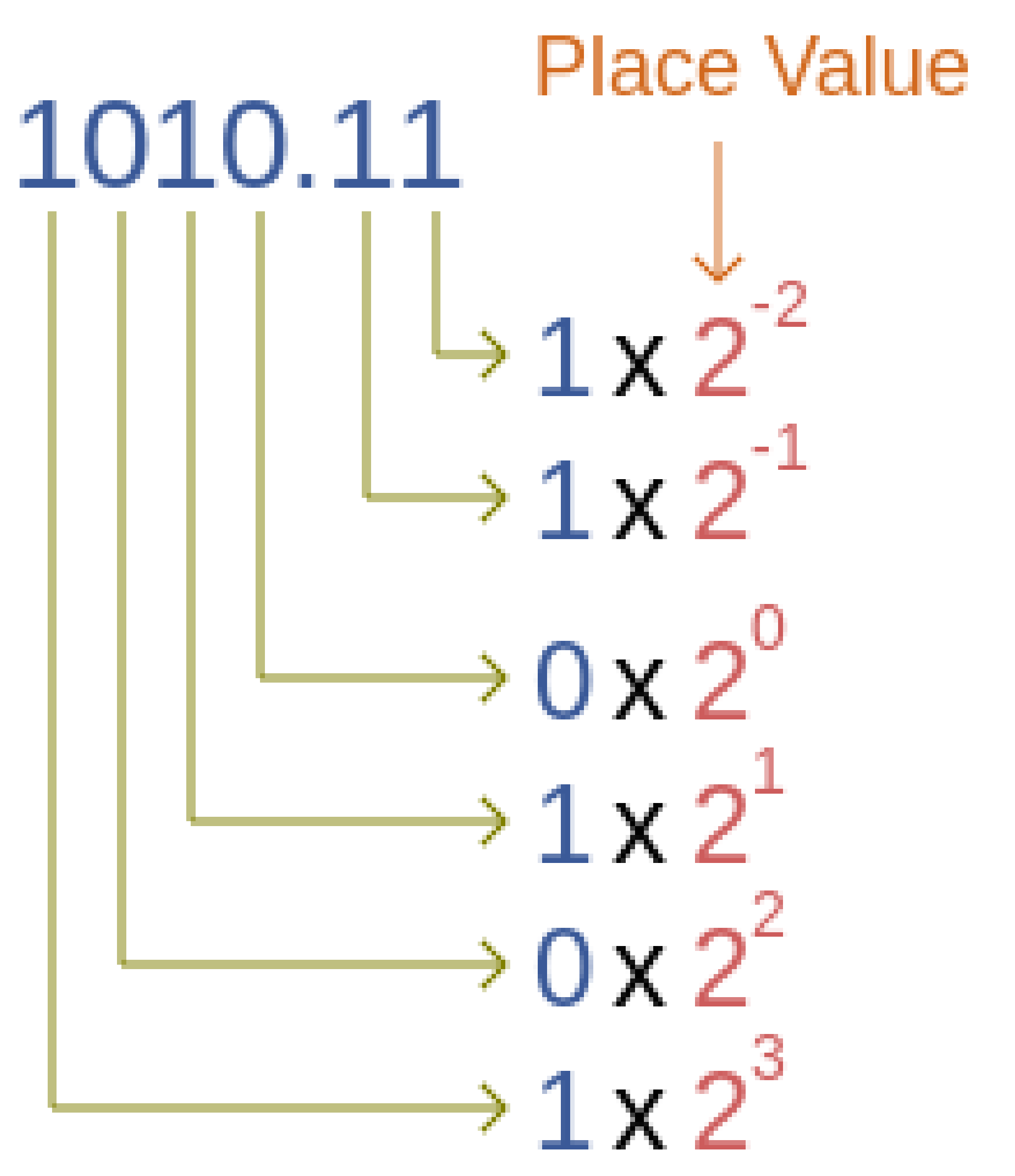
$$(0.375)_{10} = (0.011)_2$$

Combine the whole number and fractional parts to obtain the overall result.

$$(85.375)_{10} = (1010101)_2 + (0.011)_2 = (1010101.011)_2$$

Binary to Decimal

$(1010.11)_2$



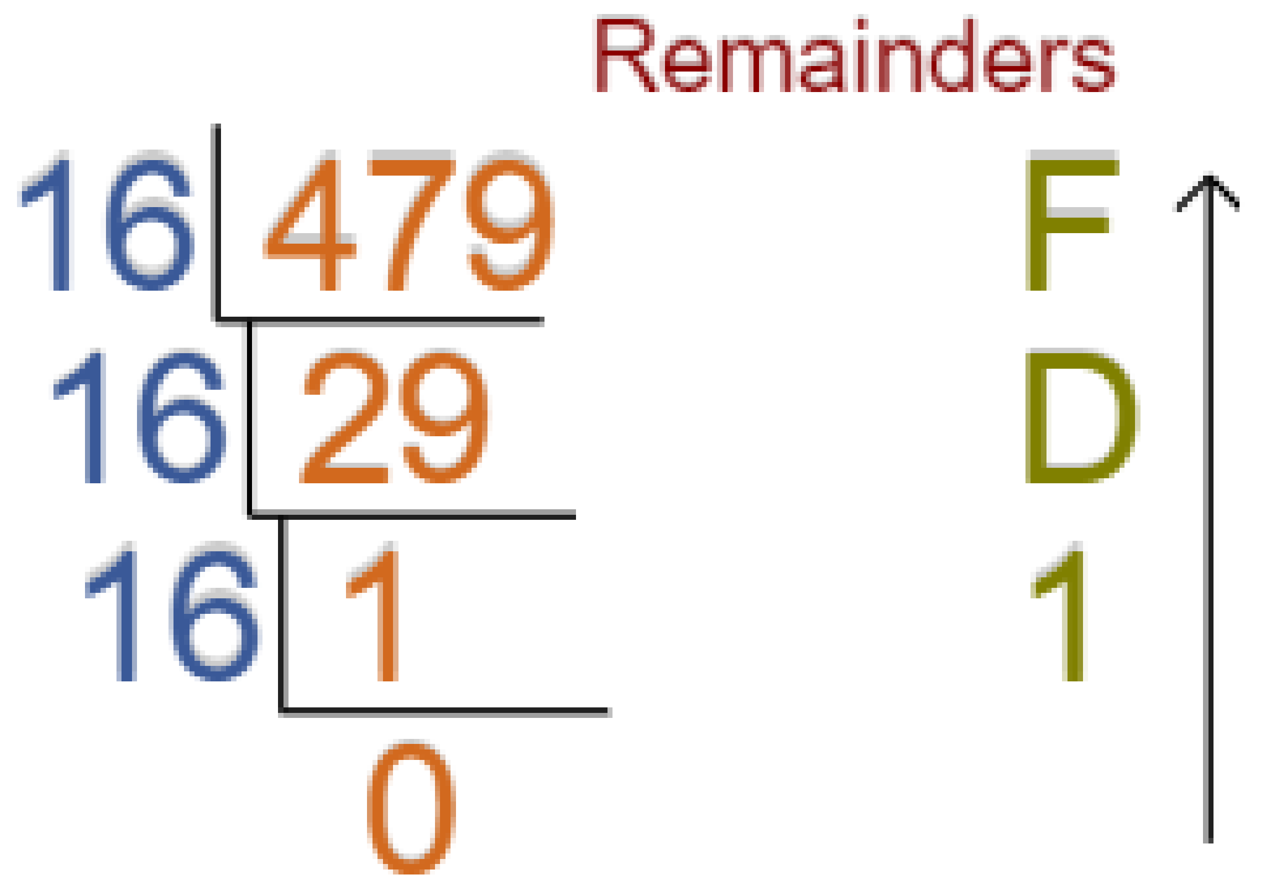
We multiply each binary digit with its place value and add the products.

$$(1010.11)_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

$$= 8 + 2 + \frac{1}{2} + \frac{1}{4}$$

$$= (10.75)_{10}$$

Decimal to Hexadecimal



Write the remainders from **bottom to top.**

$$(479)_{10} = (1DF)_{16}$$

Hexadecimal to Decimal

Steps

B 6 5 F

Place Value

15×16^0

5×16^1

6×16^2

11×16^3

$$(B65F)_{16} = (11 \times 16^3) + (6 \times 16^2) + (5 \times 16^1) + (15 \times 16^0)$$

$$= 45056 + 1536 + 80 + 15$$

$$= (46687)_{10}$$

Decimal to Octal

Divide the number repeatedly by 8 until the quotient becomes 0.

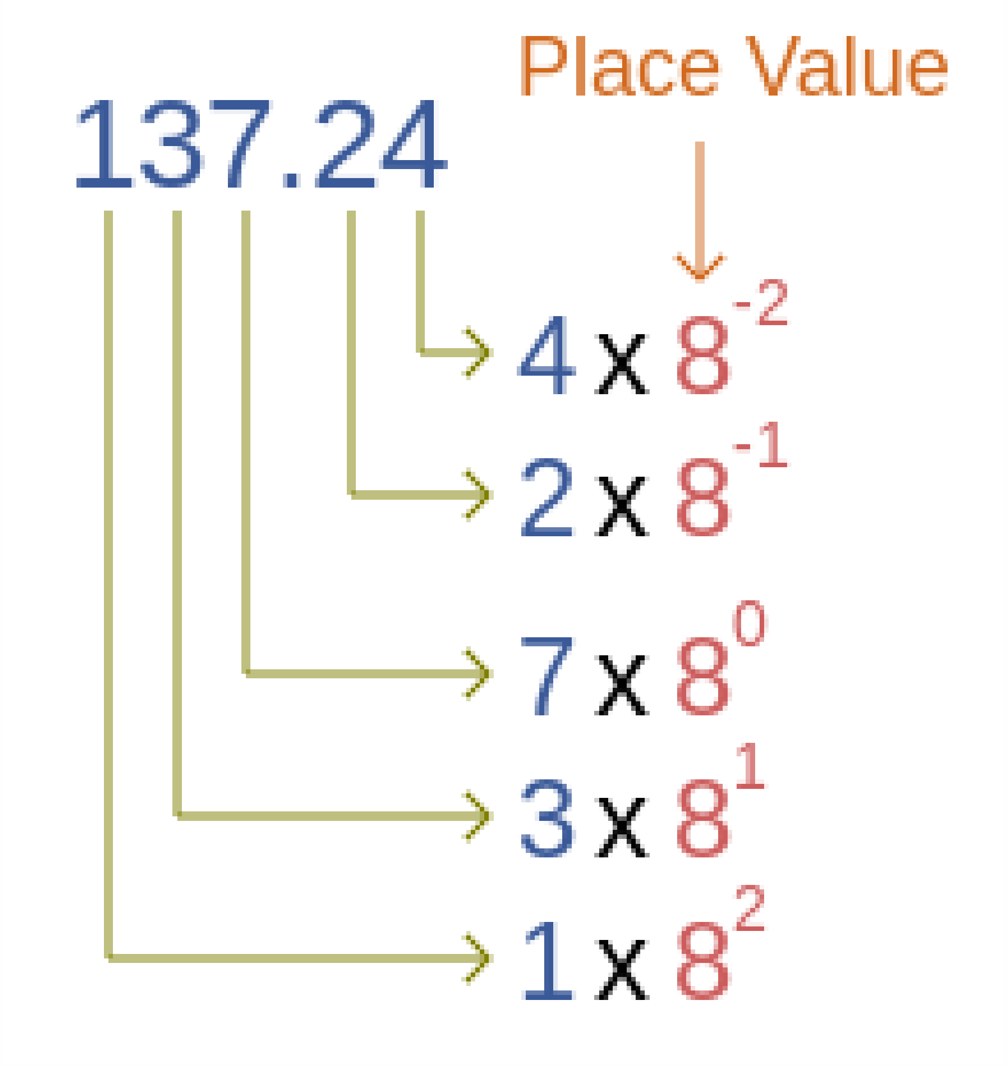
$$(739)_{10}$$

		Remainders
8	739	3
8	92	4
8	11	3
8	1	1
	0	

$$(739)_{10} = (1343)_8$$

Octal to Decimal

$(137.24)_8$



We multiply each digit with its place value and add the products.

$$(137.24)_8 = (1 \times 8^2) + (3 \times 8^1) + (7 \times 8^0) + (2 \times 8^{-1}) + (4 \times 8^{-2})$$

$$= 64 + 24 + 7 + \frac{2}{8} + \frac{4}{64}$$

$$= (95.3125)_{10}$$

$$(137.24)_8 = (95.3125)_{10}$$

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Hexadecimal to Octal

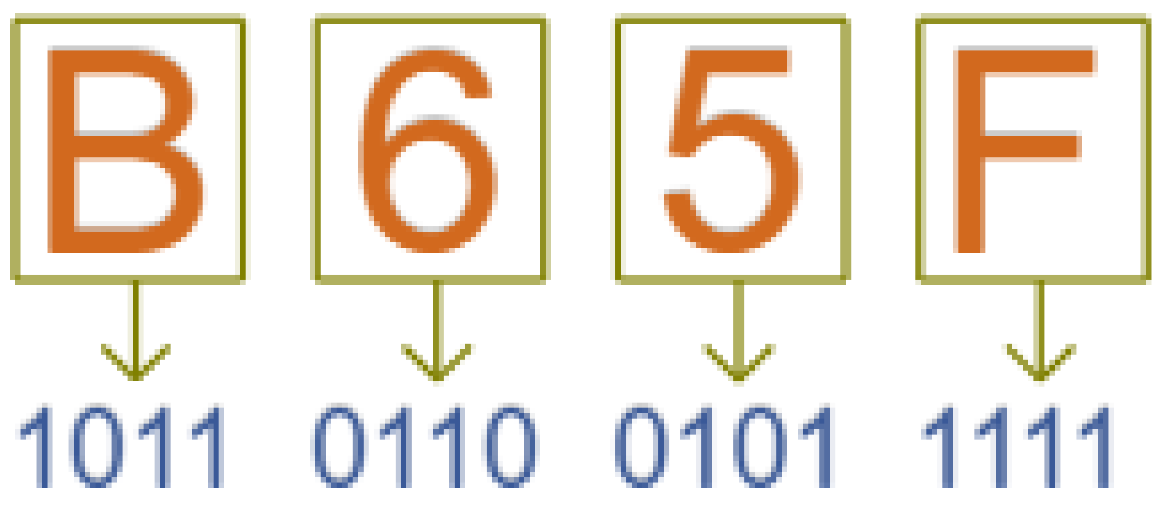
Convert each hex digit to 4 binary digits and then convert each 3 binary digits to octal digits.

Example, we can take $(B65F)_{16}$

STEP 1

Hexadecimal to Binary

In the first step, we convert the hexadecimal number to binary.



STEP 2

Binary to Octal

In the second step, we convert the binary number to octal.

001	011	011	001	011	111
↓	↓	↓	↓	↓	↓
1	3	3	1	3	7

Last STEP

Combining Results

Using the equalities we obtained in steps 1 and 2, we reach the following result.

$$(B65F)_{16} = (133137)_8$$

Hexadecimal to Binary

$(A46.09)_2$

To convert a hexadecimal number to binary, we write 4 bit binary equivalent of each hexadecimal digit in the same order.

A	4	6	.	0	9
1010	0100	0110		0000	1001

$$(A46.09)_2 = (101001000110.00001001)_8$$

Homework

❑ Decimal to Hexadecimal

Example: Convert $(5386)_{10}$ to a hexadecimal $(?)_{16}$ number.

Number (Division)	Quotient	Remainder
5386 / 16	336	10 = A
336 / 16	21	0
21 / 16	1	5
1 / 16	0	1

Decimal Value ⟶ Hexadecimal Value

$(5386)_{10} \longrightarrow (150\,A)_{16}$

Binary to Octal

$(1100.11011)_2$

Starting from the binary point, we partition the binary number into groups of three bits.

$$\underset{\downarrow \atop 1}{\boxed{001}}\;\underset{\downarrow \atop 4}{\boxed{100}}\;.\;\underset{\downarrow \atop 6}{\boxed{110}}\;\underset{\downarrow \atop 6}{\boxed{110}}$$

In the **integer part**, we proceed to the left. To complete the leftmost group of bits, we append **two zeros** to the left.

In the **fractional part**, we proceed to the right. To complete the rightmost group of bits, we append **a zero** to the right.

$(001)_2 = (1)_8$

$(100)_2 = (4)_8$

$(110)_2 = (6)_8$

$(110)_2 = (6)_8$

We convert each group of binary numbers to octal and write them in the same order.

$$(1100.11011)_2 = (14.66)_8$$

Octal to Binary

$(1743)_8$

To convert an octal number to binary, we write 3 bit binary equivalent of each octal digit in the same order.

1	7	4	3
001	111	100	011

$$(1743)_8 = (001111100011)_2$$

Octal to Hexadecimal

$(46.1)_8$

We can convert an octal number to hexadecimal in two steps.

STEP 1

In the first step, we convert the octal number to binary.

$$\begin{array}{cccc} \boxed{4} & \boxed{6} & . & \boxed{1} \\ \downarrow & \downarrow & & \downarrow \\ 100 & 110 & & 001 \end{array}$$

To convert an octal number to binary, we write 3 bit binary equivalent of each octal digit in the same order.

$$(46.1)_8 = (100110.001)_2$$

STEP 2

In the second step, we convert the binary number to hexadecimal.

$$\underbrace{0010}_{2}\ \underbrace{0110}_{6}\ .\ \underbrace{0010}_{2}$$

Starting from the binary point, we partition the binary number into groups of 4 bits. In the whole number part, we proceed to the left and in the fractional part, we proceed to the right.

$$(100110.001)_2 = (26.2)_{16}$$

STEP 3

COMBINING RESULTS

Using the equalities we obtained in steps 1 and 2, we reach the following result.

$$(46.1)_8 = (26.2)_{16}$$

Diminished Radix Complement

- **(r - 1)'s** complement of N is **$(r^n - 1) - N$**

Where,
N - number
r - base
n - digits

- **For decimal numbers**,
  r = 10 and **r - 1 = 9**, 9's complement of N is

  $$(10^n - 1) - N$$

- In this case, $\mathbf{10^n}$ represents a number that consists of a single **1** followed by **n 0's.**

  $10^n - 1$ is a represented by n 9's

Radix Complement

- r's complement of N is

  r^n - N for N≠0 and **0** for N = 0.

  Where,
  N - number
  r - base
  n - digits

- Radix complement is obtained **by adding 1** to the Diminished Radix Complement

$$r^n - N = [(r^n - 1) - N] + 1$$

Interpreting the Other Digits

Given n binary digits,

- the digit with **weight 2(n-1) is the sign** and
- the digits with **weights 2(n-2) down to 2(0) represents 2(n-1)** distinct elements.

There two popular ways to interpret the other digits:

1. Signed-Magnitude
2. Signed-Complement
    a) Signed One's Complement
    b) Signed Two's Complement

Signed-magnitude representation

01001 → 9 (unsigned) or +9 (signed)

leftmost bit is 0 denoted positive

Positive

11001 → 25 (unsigned) or -9 (signed)

leftmost bit is 1
denotes negative

bits represent binary 9

Negative

Signed complement representation

Signed 1's complement representation of -9:

```
 111 11
 01001
-------
[1][0110]
```

$\boxed{1}$ denotes -ve

$\boxed{0110}$ 1's complement of +9

Signed 2's complement representation of -9:

```
  1 0110   ← 1's complement of +9
       1
--------
[1][0111]
```

$\boxed{1}$ denotes -ve

$\boxed{0111}$ 2's complement of +9

Binary Logic

There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result, denoted by z.

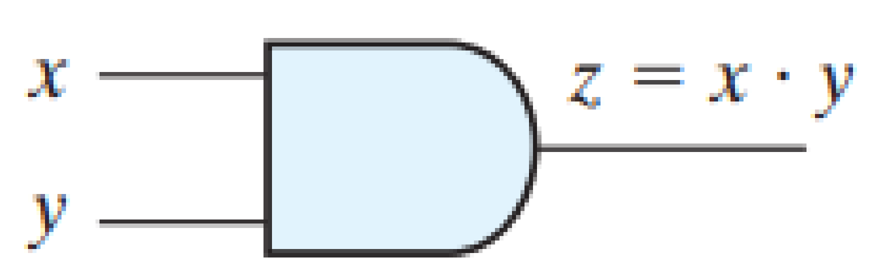
- AND – represented by a dot or absence of an operator. **E.g.**, $x \cdot y = z$ or $xy = z$
- OR – represented by a plus sign. **E.g.,** $x + y = z$
- NOT – represented by a prime (sometimes by an overbar). **E.g.**, $x' = z$ or $\bar{x} = z$

Logic Gates

A logic gate is a simple switching circuit that determines whether an input pulse can pass through to the output in digital circuits.

(a) **AND Gate:**

Two-input AND gate

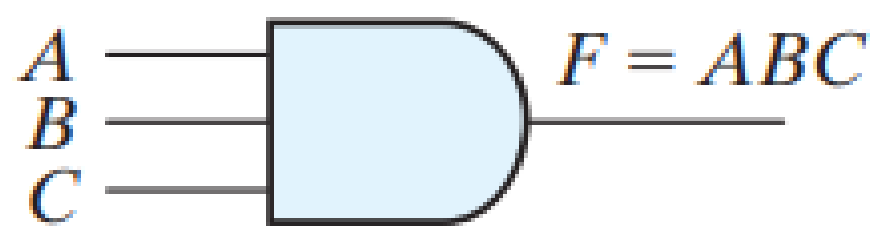


Truth Table

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Logic Gates

Three-input AND gate

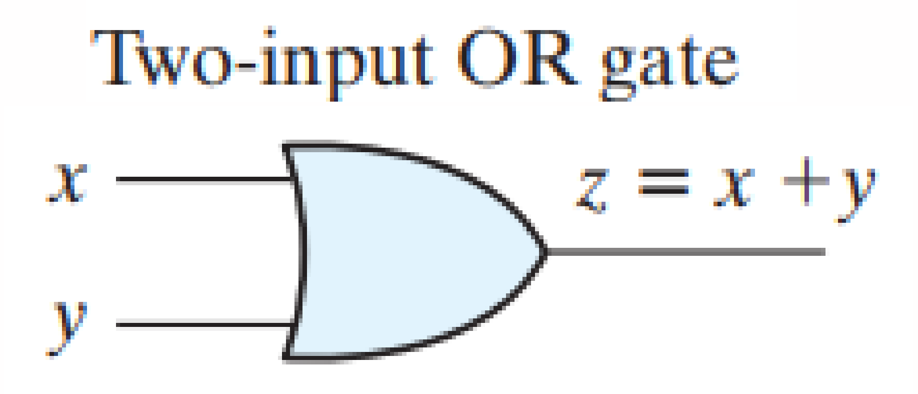


Truth Table

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

Logic Gates

(b) **OR Gate:**

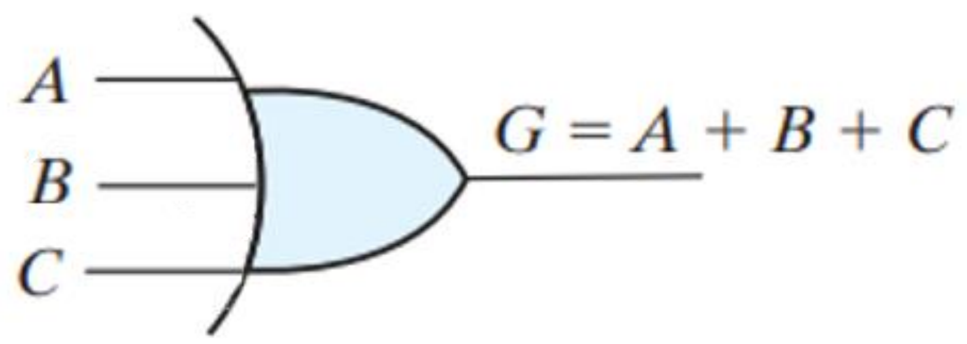


Truth Table

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

Logic Gates

Four-input OR gate



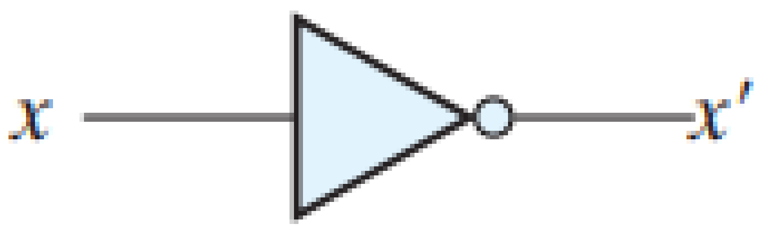
Truth Table

A	B	C	G
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Logic Gates

(c) **NOT Gate:**

NOT gate or inverter



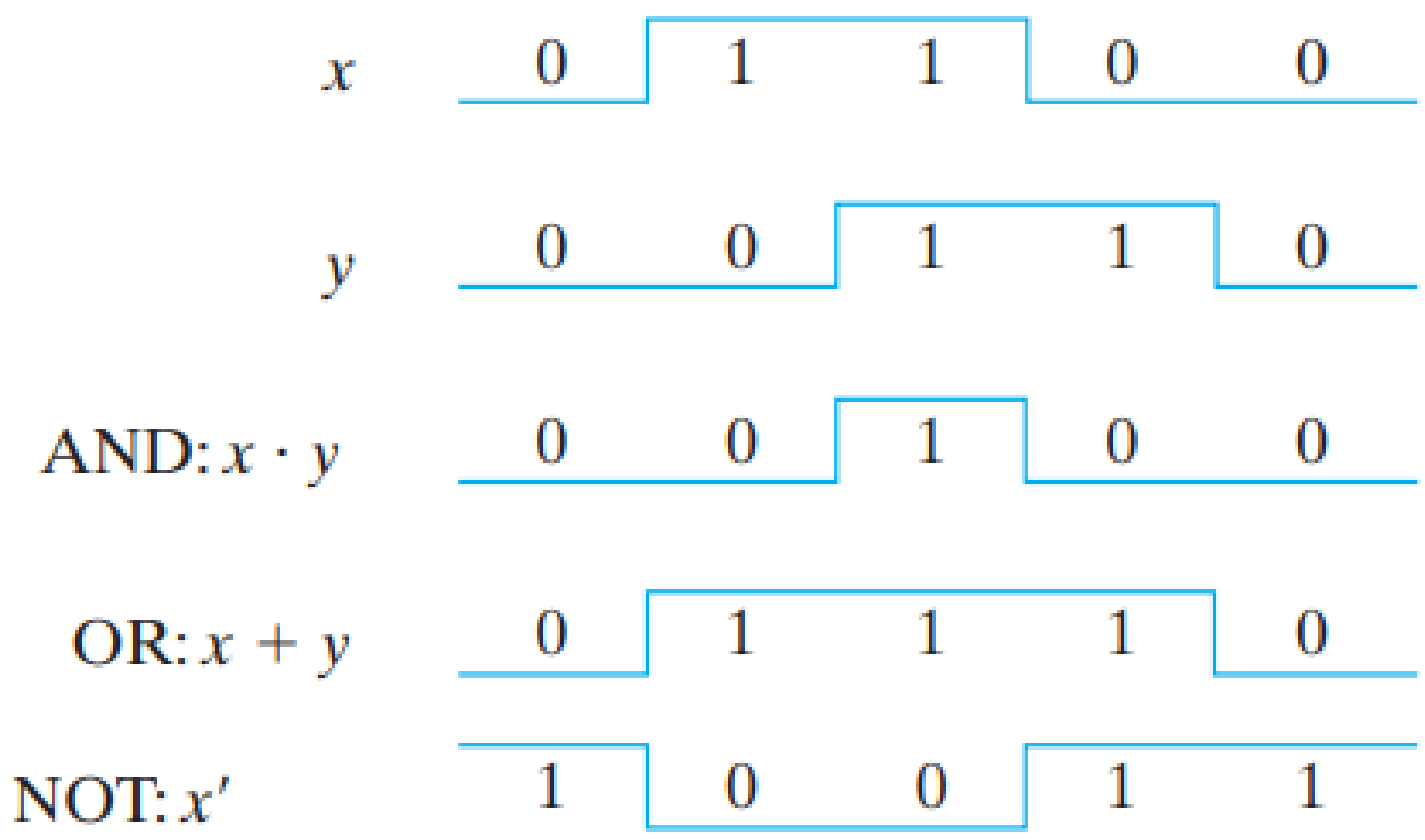
Truth Table

NOT

x	x'
0	1
1	0

Logic Gates

Input – Output signals for gates



Operator Precedence

- The operator precedence for evaluating Boolean expressions is
  - (1) parentheses
  - (2) NOT
  - (3) AND
  - (4) OR

- Expressions inside parentheses must be evaluated before all other operations.

- The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR.

Minterm/Standard product

The **2^n** different minterms determined by a method shown in Table for three variables.

			Minterms		Maxterms	
x	***y***	***z***	**Term**	**Designation**	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Maxterm/Standard sum

The **eight(2^n)** maxterms for three variables, together with their symbolic designations, are listed in Table.

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Sum of Minterms

- Express the Boolean function **F = A + B'C** as a sum of minterms.

- The function has three variables: **A**, **B**, and **C**. The first term **A** is missing two variables; therefore,

  A = A(B + B') = AB + AB'

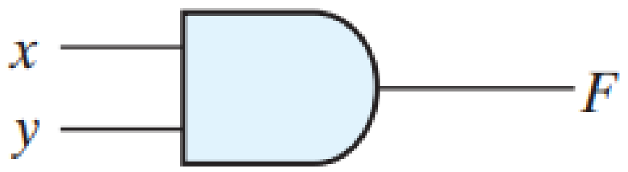
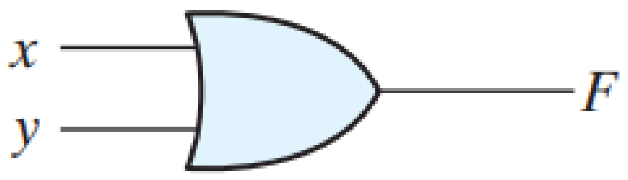
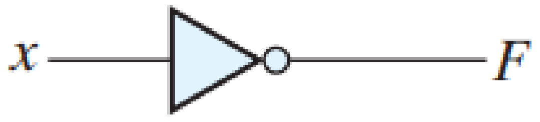
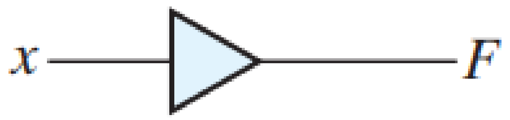
- This function is still missing one variable, so

  A = AB(C + C') + AB'(C + C')
  = ABC + ABC' + AB'C + AB'C'

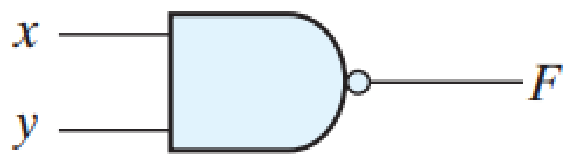
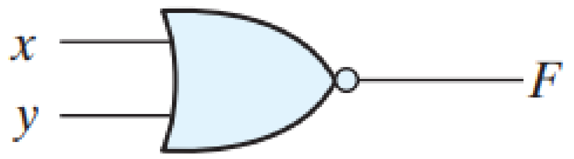
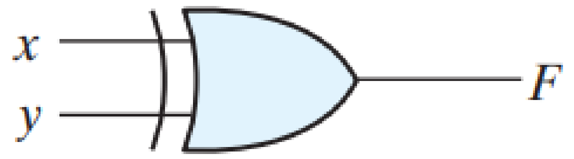
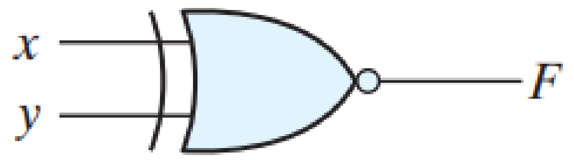
- The second term B'C is missing one variable; hence,

  B'C = B'C(A + A') = AB'C + A'B'C

Digital Logic Gates

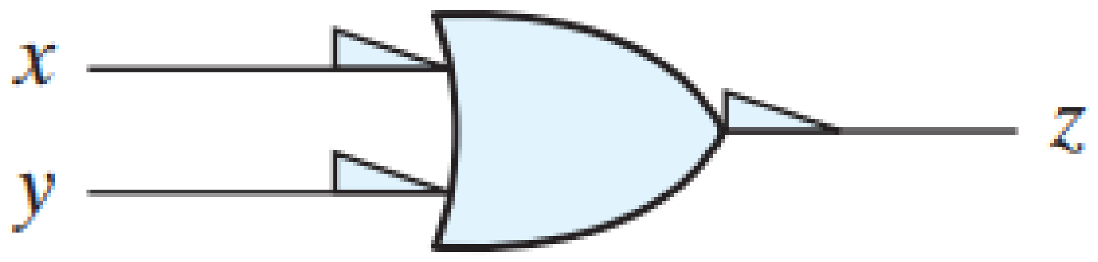
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1

Digital Logic Gates

Name	Graphic symbol	Algebraic function	Truth table
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

Positive and Negative Logic

- The graphic symbol for the negative- logic OR gate is shown in (f).



(f) Negative logic OR gate

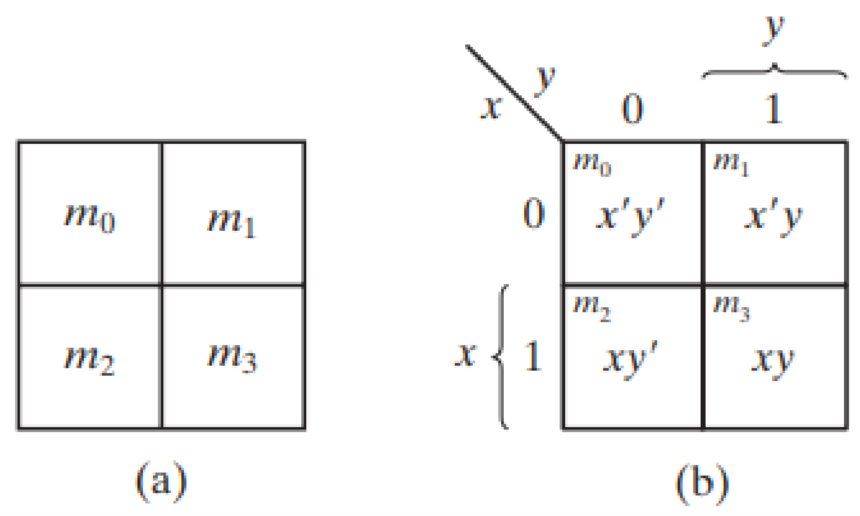
- The small triangles in the inputs and output designate a **polarity indicator**, the presence of which along a terminal signifies that negative logic is assumed for the signal.

- Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.

K-map

- A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized.
- The map presents a visual diagram of all possible ways a function may be expressed in standard form.
- The simplified expressions produced by the map are always in one of the two standard forms:
    - sum of products or
    - products of sums

Two-Variable K-Map

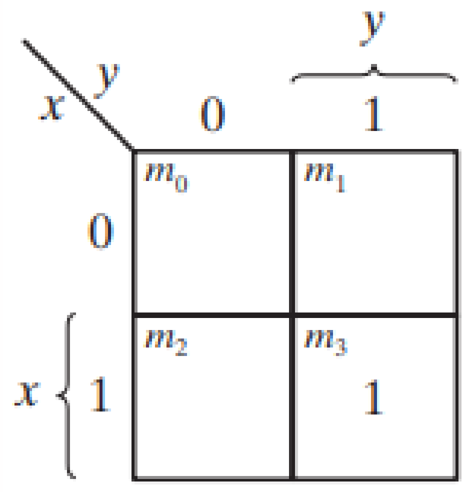


- The two-variable map is shown here.
- Four minterms for two variables.
- (b) shows the relationship between the squares and the two variables x and y .
- **0** and **1** designate the values of variables.
- Variable x appears primed in row 0 and unprimed in row 1.
- Similarly, y appears primed in column 0 and unprimed in column 1.

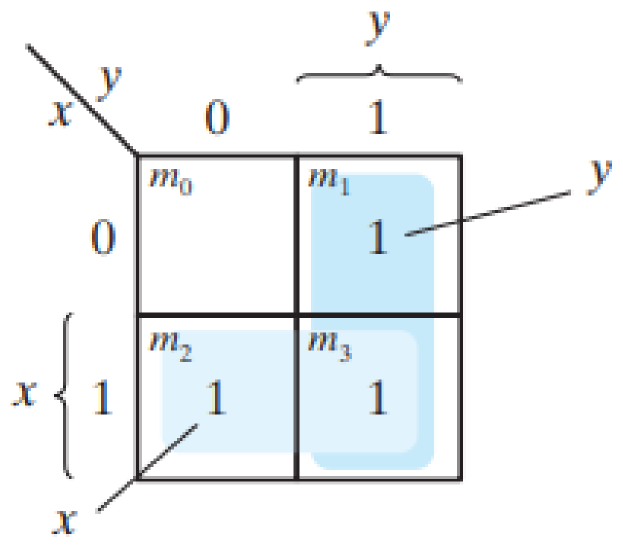
Two-Variable K-Map

- The two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables.

- Example, the function xy is shown Fig. (a).

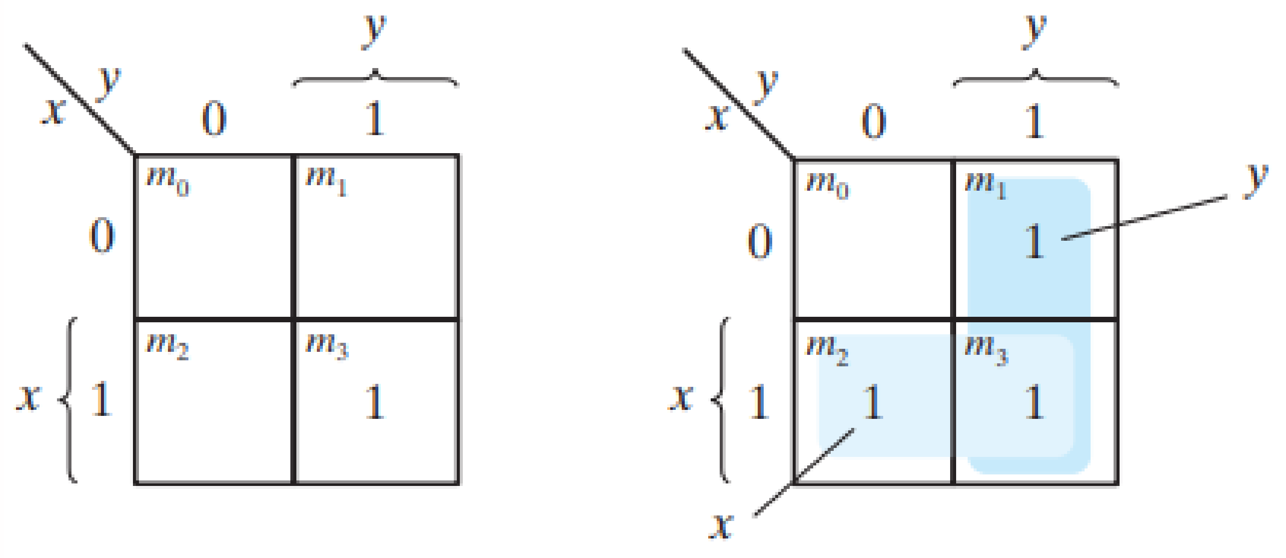


(a) xy



(b) $x + y$

Two-Variable K-Map

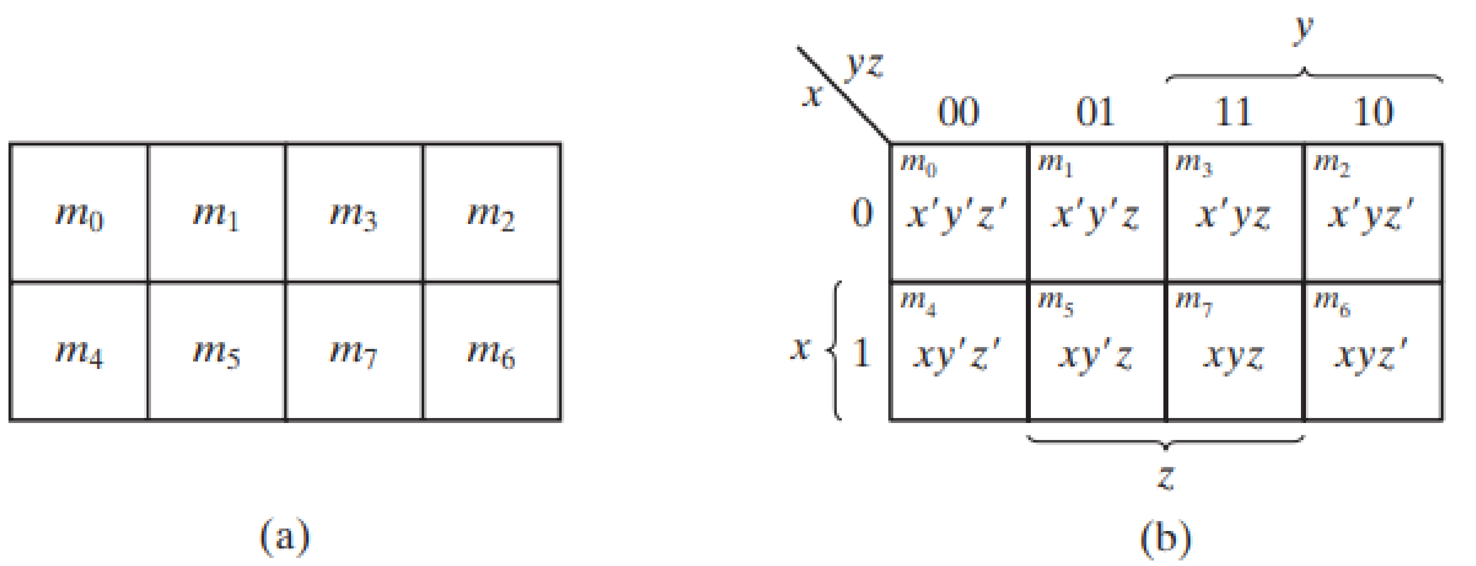


- Since **$xy = m_3$**, 1 is placed inside the square m_3.
- x + y function is represented in the map of Fig. (b) by three squares marked with 1's.
- These squares are found from the minterms of the function:

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$

Three-Variable K-Map

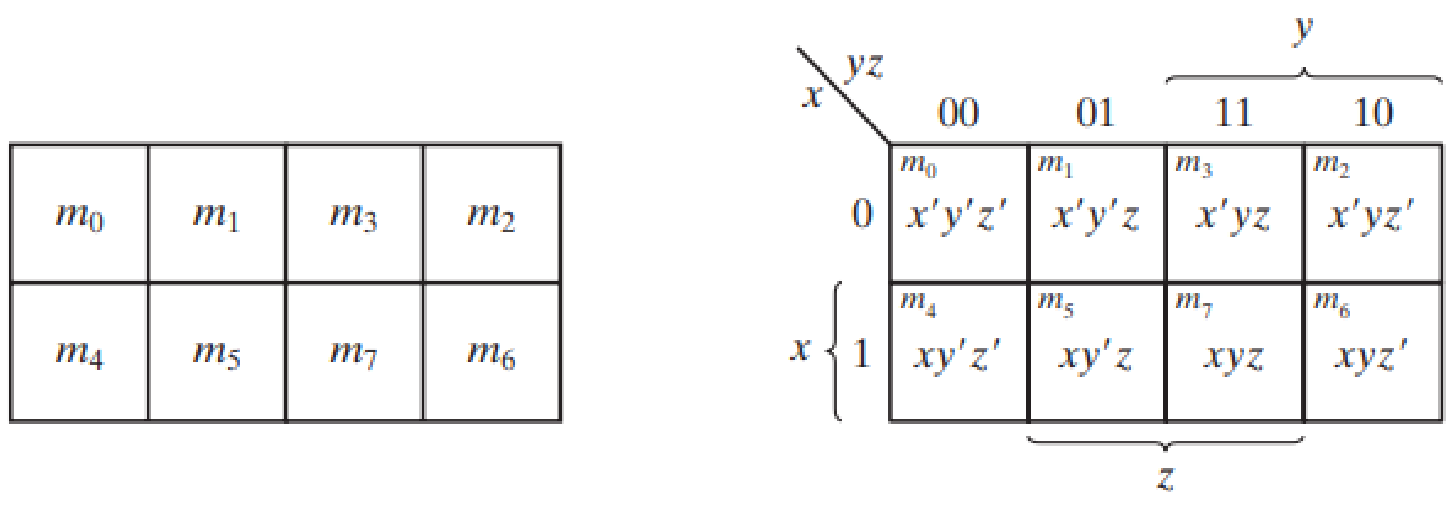
- A three-variable K-map is shown here.



- Eight minterms for three binary variables; the map consists of eight squares.

- Only one bit changes in value from one adjacent column to the next.

Three-Variable K-Map



(a) (b)

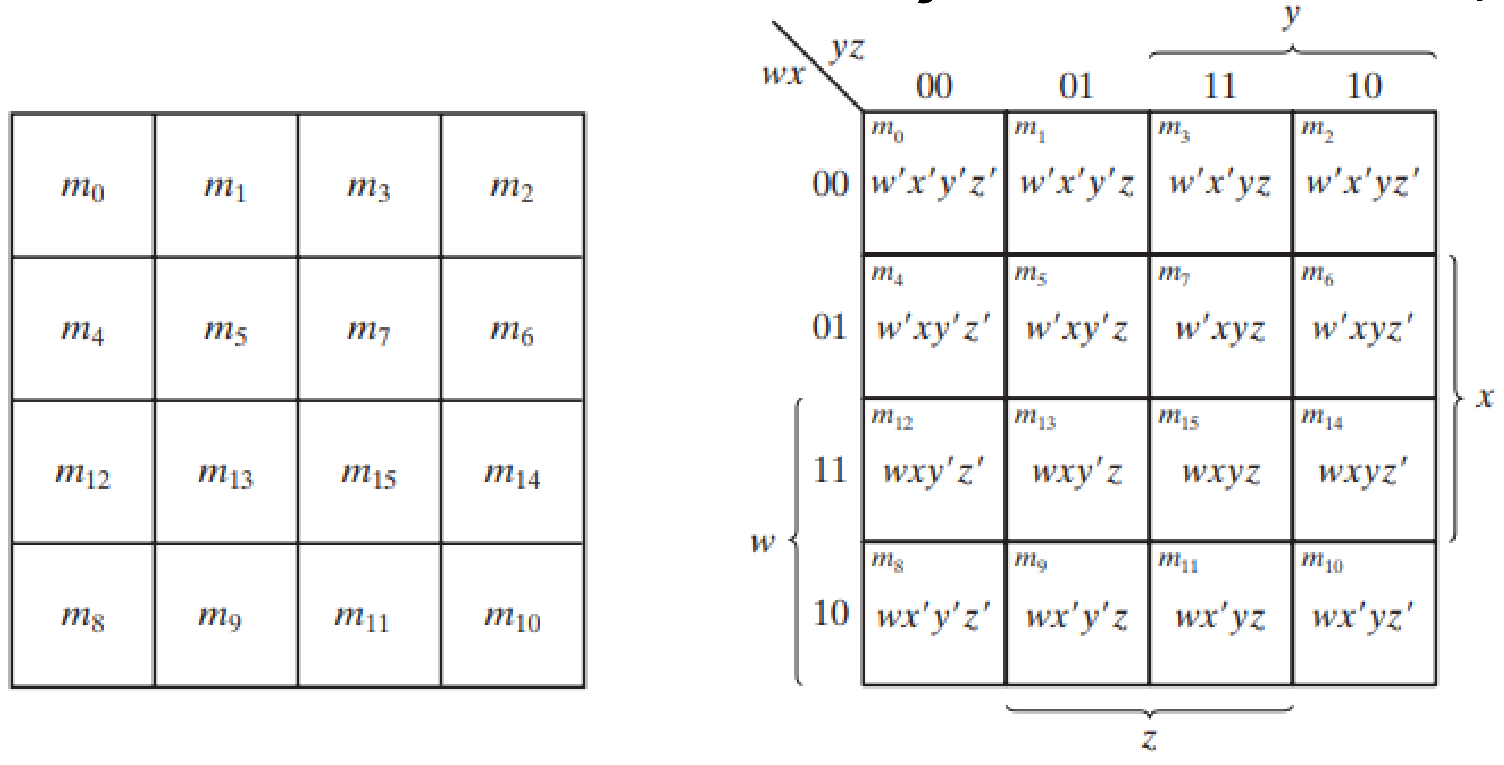
- The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables.

- For example, the square assigned to m_5:
  row - 1 and column - 01.

- When these two numbers are concatenated, they give the binary number **101**, whose decimal equivalent is **5**.

Four-Variable K-Map

- A map for Boolean functions with four binary variables (w, x, y, z) is presented in Figure.



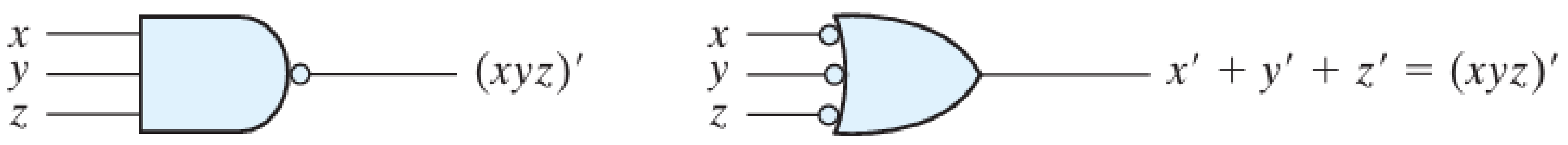
(a) (b)

The 16 minterms are listed along with the squares assigned to each.

Illustrate the relationship between squares and the four variables.

NAND Circuits

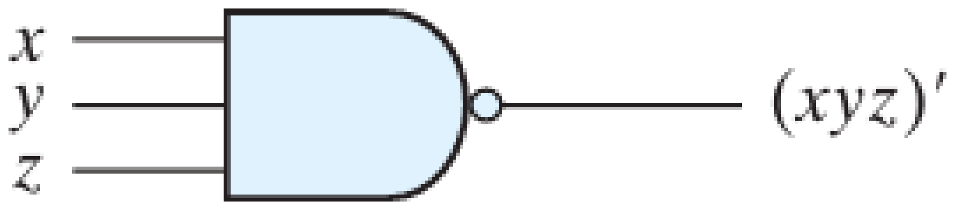
- To facilitate the conversion to NAND logic, defining an **alternative graphic symbol** for the gate is convenient.
- Two equivalent graphic symbols for the NAND gate are shown in Figure.



(a) AND-invert

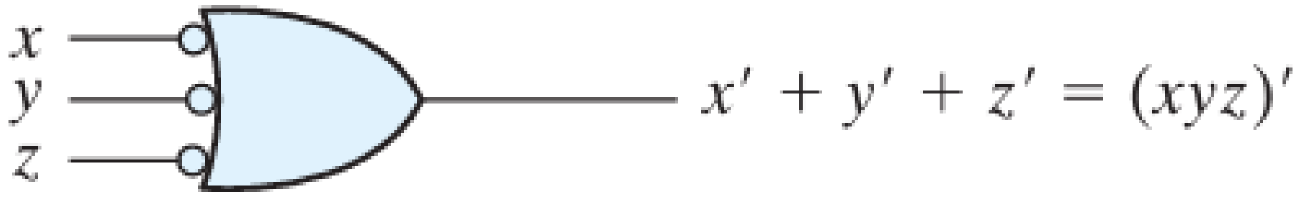
(b) Invert-OR

NAND Circuits



(a) AND-invert

- The **AND-invert symbol** has been defined previously and consists of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.



(b) Invert-OR

- The **invert-OR symbol** for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator (bubble) denotes complementation.

- The two graphic symbols' representations are useful in analyzing and designing NAND circuits.
- When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

NOR Implementation

- The **NOR** operation is the dual of the NAND operation.

- Therefore, all procedures and **rules for NOR logic** are the duals of the corresponding procedures and rules developed for NAND logic.

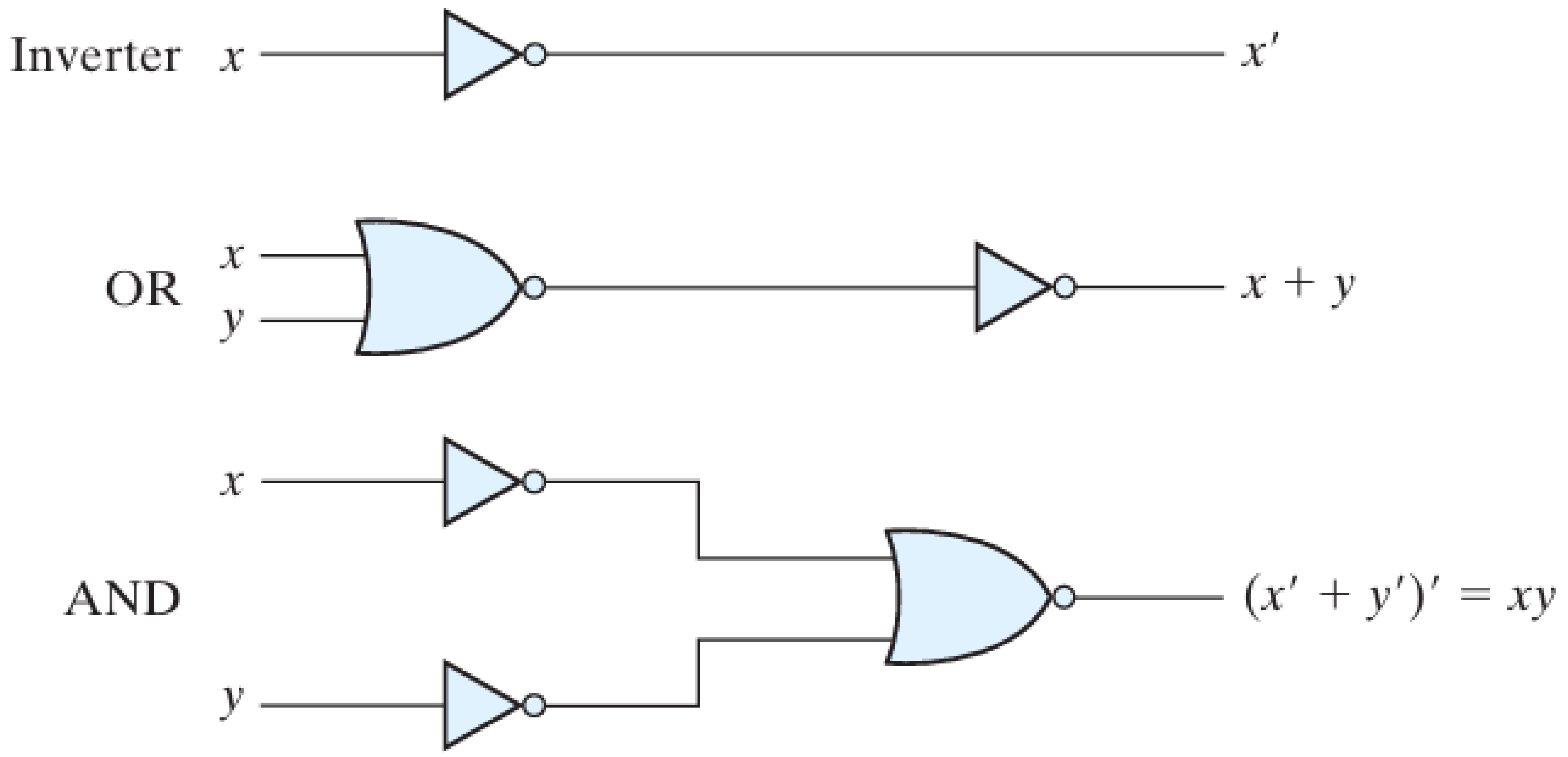
- The NOR gate is another universal gate that can be used to implement any Boolean function.

NOR Implementation

- The implementation of the complement, **OR, and AND operations with NOR** gates is shown in Figure below.

- The complement operation is obtained from a one input NOR gate that behaves exactly like an inverter.

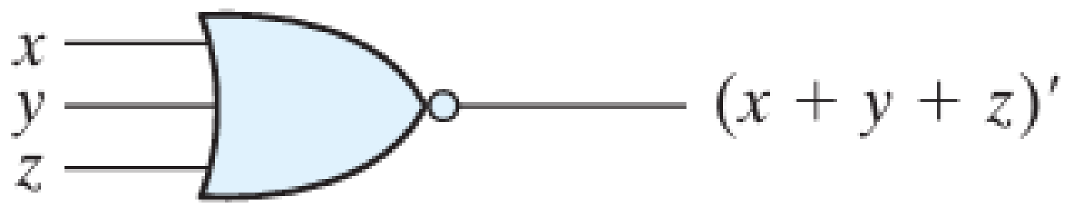
- The OR operation requires **two NOR gates**, and the **AND** operation is **obtained with a NOR** gate that has inverters in each input.



NOR Implementation

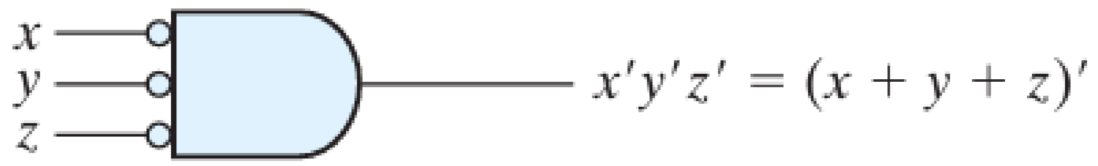
Two graphic symbols for the mixed notation:

- The OR-invert symbol defines the NOR operation as an OR followed by a complement.



(a) OR-invert

- The invert-AND symbol complements each input and then performs an AND operation.



(b) Invert-AND

- The two symbols designate the same NOR operation and are logically identical because of **DeMorgan's theorem**.

References

- Computer Organization and Architecture Designing for Performance Tenth Edition by William Stallings
- Digital Design With an Introduction to the Verilog HDL FIFTH EDITION by M Morris, M. and Michael, D., 2013.

Ollscoil na Gaillimhe
University of Galway

Thank *you*