SWITCHING THEORY AND LOGIC CIRCUITS

COURSE OBJECTIVES

1. To understand the concepts and techniques associated with the number systems and codes
2. To understand the simplification methods (Boolean algebra & postulates, k-map method and tabular method) to simplify the given Boolean function.
3. To understand the fundamentals of digital logic and to design various combinational and sequential circuits.
4. To understand the concepts of programmable logic devices(PLDs)
5. To understand formal procedure for the analysis and design of synchronous and asynchronous sequential logic

COURSE OUTCOMES

After completion of the course the student will be able to

1. Understand the concepts and techniques of number systems and codes in representing numerical values in various number systems and perform number conversions between different number systems and codes.
2. Apply the simplification methods to simplify the given Boolean function (Boolean algebra, k-map and Tabular method).
3. Implement given Boolean function using logic gates, MSI circuits and/ or PLD's.

COURSE OUTCOMES

After completion of the course the student will be able to

4. Design and analyze various combinational circuits like decoders, encoders, multiplexers, and de-multiplexers, arithmetic circuits (half adder, full adder, multiplier etc).
5. Design and analyze various sequential circuits like flip-flops, registers, counters etc.
6. Analyze and Design synchronous and asynchronous sequential circuits.

UNIT-I
Introductory Concepts
(Number systems, Base conversions)

Digital Systems

- Digital systems consider *discrete* amounts of data
- Examples
  - 26 letters in the alphabet
  - 10 decimal digits
- Larger quantities can be built from discrete values:
  - Words made of letters
  - Numbers made of decimal digits (e.g. 239875.32)
- Computers operate on binary values (0 and 1)
- Easy to represent binary values electrically
  - Voltages and currents
  - Can be implemented using circuits
  - Create the building blocks of modern computers

Understanding Decimal Numbers

- Decimal numbers are made of decimal digits: (0,1,2,3,4,5,6,7,8,9) $\Rightarrow$ Base = 10
- How many items does decimal number 8653 represents? 1000 100 10 1 ← Weight
  - 8653 = $8 \times 10^3 + 6 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$
  - Number = $d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$ = Value
- What about fractions?
  - $97654.35 = 9 \times 10^4 + 7 \times 10^3 + 6 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + 5 \times 10^{-2}$
  - In formal notation → $(97654.35)_{10}$

Understanding Octal Numbers

- Octal numbers are made of octal digits: (0,1,2,3,4,5,6,7)
- How many items does an octal number represent?
  - 512 64 8 1 = Weights
  - $(4536)_8 = 4 x 8^3 + 5 x 8^2 + 3 x 8^1 + 6 x 8^0 = (2398)_{10}$
- What about fractions?
  - $(465.27)_8 = 4 x 8^2 + 6 x 8^1 + 5 x 8^0 + 2 x 8^{-1} + 7 x 8^{-2}$
- Octal numbers don’t use digits 8 or 9

Understanding Hexadecimal Numbers

- Hexadecimal numbers are made of 16 digits:
  - (0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F)
- How many items does a hex number represent?

  4096 256 16 1 = Weights

  - $(3A9F)_{16} = 3x16^{3} + 10x16^{2} + 9x16^{1} + 15x16^{0} = 14999_{10}$
- What about fractions?
  - $(2D3.5)_{16} = 2x16^{2} + 13x16^{1} + 3x16^{0} + 5x16^{-1} = 723.3125_{10}$
- Note that each hexadecimal digit can be represented with four bits
  - $(1110)_{2} = (E)_{16}$
- Groups of four bits are called a nibble
  - $(1110)_{2}$

Understanding Binary Numbers

- Binary numbers are made of binary digits (bits):
  - 0 and 1
- How many items does a binary number represent?
  - 8 4 2 1 = Weights
  - $(1011)_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = (11)_{10}$
- What about fractions?
  - $(110.10)_2 = 1x2^2 + 1x2^1 + 0x2^0 + 1x2^{-1} + 0x2^{-2}$
- Groups of eight bits are called a byte
  - $(11001001)_2$
- Groups of four bits are called a nibble
  - $(1101)_2$

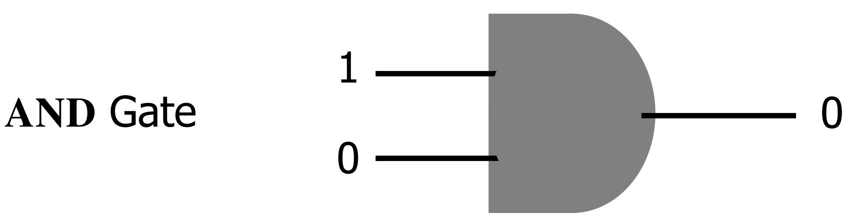
Putting It All Together

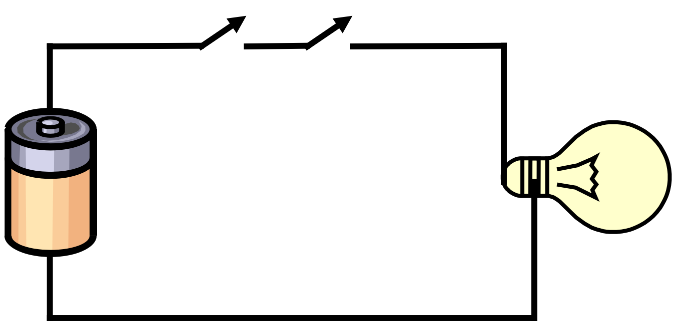
- Binary, octal, and hexadecimal are similar
- Easy to build circuits to operate on these representations
- Possible to convert between the three formats

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Why Use Binary Numbers?

- Easy to represent 0 and 1 using electrical values
- Possible to tolerate noise
- Easy to transmit data
- Easy to build binary circuits





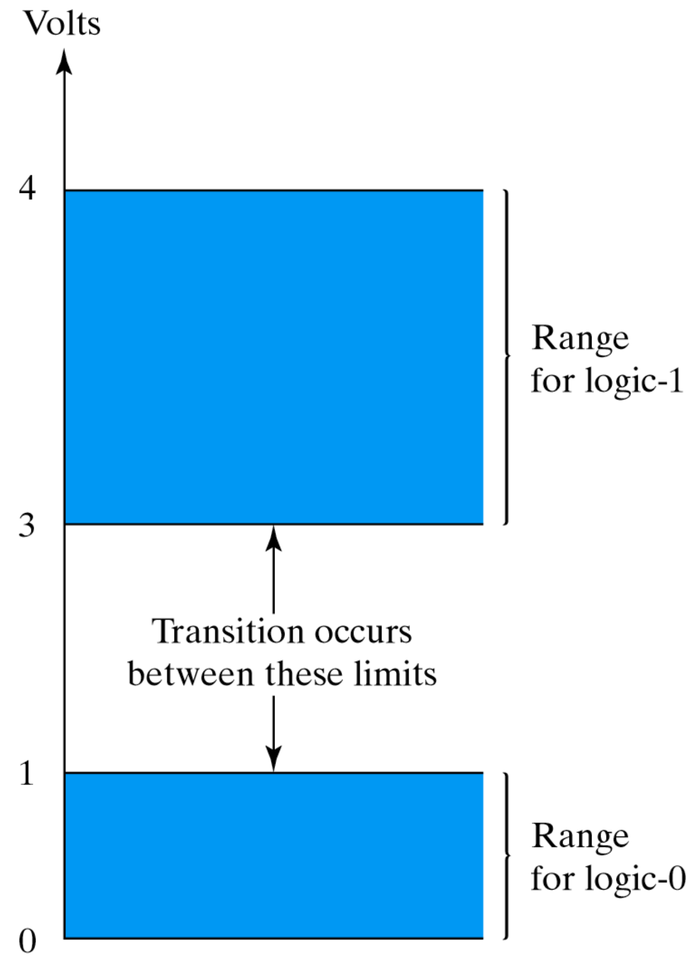
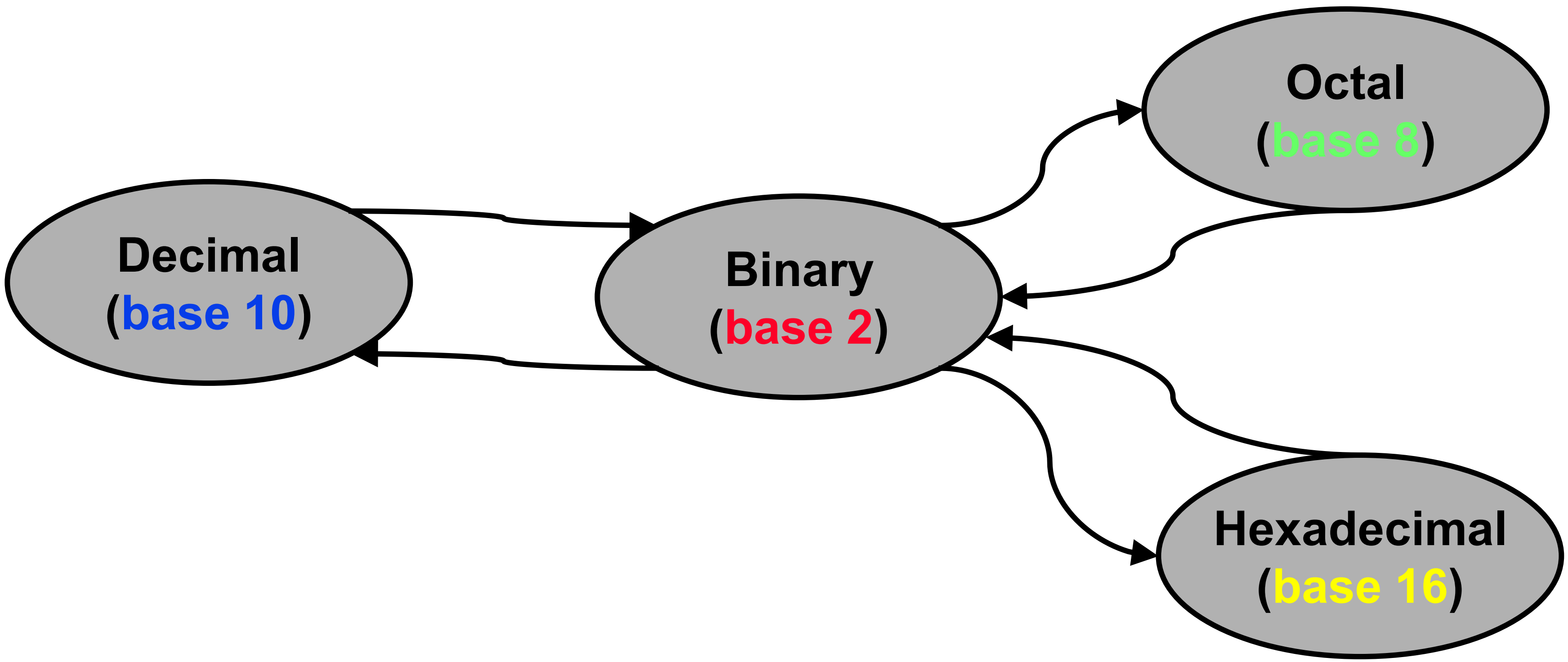


Fig. 1-3 Example of binary signals

Conversion Between Number Bases

- Learn to convert between bases
- Already demonstrated how to convert from binary to decimal



Convert an Integer *from* Decimal *to* Another Base

For each digit position:

1. Divide decimal number by the base (e.g. 2)
2. The *remainder* is the lowest-order digit
3. Repeat first two steps until no *divisor* remains

Example for $(13)_{10}$:

	Quotient		Remainder	Coefficient
13/2 =	6	+	1	$a_0 = 1$
6/2 =	3	+	0	$a_1 = 0$
3/2 =	1	+	1	$a_2 = 1$
1/2 =	0	+	1	$a_3 = 1$

Answer $(13)_{10} = (a_3\ a_2\ a_1\ a_0)_2 = (1101)_2$

MSB LSB

Convert a Fraction *from* Decimal *to* Another Base

For each digit position:

1. Multiply decimal number by the base (e.g. 2)
2. The *integer* is the highest-order digit
3. Repeat first two steps until fraction becomes zero

Example for $(0.625)_{10}$:

	Integer		Fraction	Coefficient
0.625 x 2 =	1	+	0.25	$a_{-1} = 1$
0.250 x 2 =	0	+	0.50	$a_{-2} = 0$
0.500 x 2 =	1	+	0	$a_{-3} = 1$

Answer $(0.625)_{10} = (0.a_{-1}\ a_{-2}\ a_{-3})_2 = (0.101)_2$

MSB LSB

The Growth of Binary Numbers

n	2^n
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$

n	2^n	
8	$2^8=256$	
9	$2^9=512$	
10	$2^{10}=1024$	Kilo
11	$2^{11}=2048$	
12	$2^{12}=4096$	
20	$2^{20}=1M$	Mega
30	$2^{30}=1G$	Giga
40	$2^{40}=1T$	Tera

Convert an Integer *from* Decimal *to* Octal

For each digit position:

1. Divide decimal number by the base (8)
2. The ***remainder*** is the lowest-order digit
3. Repeat first two steps until no *divisor* remains

Example for $(175)_{10}$:

	Quotient		Remainder	Coefficient
175/8 =	21	+	7	$a_0 = 7$
21/8 =	2	+	5	$a_1 = 5$
2/8 =	0	+	2	$a_2 = 2$

Answer $(175)_{10} = (a_2\, a_1\, a_0)_8 = (257)_8$

Convert a Fraction *from* Decimal *to* Octal

For each digit position:

1. Multiply decimal number by the base (e.g. 8)
2. The *integer* is the highest-order digit
3. Repeat first two steps until fraction becomes zero

Example for $(0.3125)_{10}$:

	Integer		Fraction	Coefficient
$0.3125 \times 8 =$	2	+	0.5	$a_{-1} = 2$
$0.5000 \times 8 =$	4	+	0.0	$a_{-2} = 4$

Answer $(0.3125)_{10} = (0.24)_8$

Conversion Between Base 16 and Base 2

- Conversion is easy!

  Determine the 4-bit binary value for each hex digit
- Note that there are 16 different values of four bits
- Easier to read and write in hexadecimal
- Representations are equivalent!

$$3A9F_{16} = \underbrace{0011}_{3}\ \underbrace{1010}_{A}\ \underbrace{1001}_{9}\ \underbrace{1111}_{F}{}_2$$

Conversion Between Base 16 and Base 8

1. Convert from Base 16 to Base 2
2. Regroup bits into groups of three starting from right
3. Ignore leading zeros
4. Each group of three bits forms an octal digit

$$3A9F_{16} = \underbrace{0011}_{3}\ \underbrace{1010}_{A}\ \underbrace{1001}_{9}\ \underbrace{1111}_{F}{}_2$$

$$35237_8 = \underbrace{011}_{3}\ \underbrace{101}_{5}\ \underbrace{010}_{2}\ \underbrace{011}_{3}\ \underbrace{111}_{7}{}_2$$

Binary Addition

- **Binary addition is very simple**

```
 1   1   1   1    1   1       <------------  carries
     1   1   1    1   0   1            = 61
+            1    0   1   1   1        = 23
 ------------------------------
 1   0   1   0    1   0   0            = 84
```

Binary Subtraction

- We can also perform subtraction (with borrows in place of carries)
- Let's subtract $(10111)_2$ from $(1001101)_2$ …

1 10

0 ~~10~~ 10 0 ~~0~~ 10 borrows

~~1~~ ~~0~~ ~~0~~ ~~1~~ ~~1~~ ~~0~~ 1 = 77

\- 1 0 1 1 1 = 23

\-

1 1 0 1 1 0 = 54

Binary Multiplication

Binary multiplication is much the same as decimal multiplication, except that the multiplication operations are much simpler…

```
              1   0   1   1   1
x                 1   0   1   0
-------------------------------
              0   0   0   0   0
          1   0   1   1   1
      0   0   0   0   0
  1   0   1   1   1
-------------------------------
  1   1   1   0   0   1   1   0
```

Summary

- Binary numbers are made of binary digits (bits)
- Binary and octal number systems
- Conversion between number systems
- Addition, subtraction, and multiplication in binary

Introductory Concepts
(Complements)

How To Represent Signed Numbers

- Plus and minus signs are used for decimal numbers:
  - 25 (or +25), −16, etc
- In computers, everything is represented as *bits*
- Three types of signed binary number representations:
  - signed magnitude
  - 1's complement
  - 2's complement
- In each case: left-most bit indicates the sign: '0' for positive and '1' for negative

Signed Magnitude Representation

- The left most bit is designated as the *sign* bit while the remaining bits form the *magnitude*
- The sign bit should not be included in addition / subtraction operations

$$\underset{\text{Sign bit}}{0}\underbrace{0001100}_{\text{Magnitude}}{}_2 = 12_{10}$$

$$\underset{\text{Sign bit}}{1}\underbrace{0001100}_{\text{Magnitude}}{}_2 = -12_{10}$$

One's Complement Representation

- The one's complement of a binary number is done by complementing (i.e. inverting) all bits

  1's comp of 00110011 is 11001100

  1's comp of 10101010 is 01010101

- For a n-bit number N the 1's complement is $(2^n - 1) - N$
- Called "*diminished radix complement*" by Mano
- To find the negative of a 1's complement number take its 1's complement

$\underbrace{0}_{\text{Sign bit}}\underbrace{0001100}_{\text{Magnitude}}{}_2 = 12_{10}$

$\underbrace{1}_{\text{Sign bit}}\underbrace{1110011}_{\text{Code}}{}_2 = -12_{10}$

One's Complement Representation

4 bits

$\Downarrow$

16 combinations

7	0111
6	0110
.	.
.	.
1	0001
0	0000
– 0	1111
– 1	1110
.	.
.	.
– 6	1001
– 7	1000

Two's Complement Representation

- The two's complement of a binary number is done by complementing (inverting) all bits then adding 1

  2's comp of 00110011 is 11001101

  2's comp of 10101010 is 01010110

- For an **n**-bit number **N** the 2's complement is $(2^n - 1) - N + 1$
- Called "*radix complement*" by Mano
- To find the negative of a 2's complement number take its 2's complement

$\underset{\text{Sign bit}}{0}\underbrace{0001100}_{\text{Magnitude}}{}_2 = 12_{10}$

$\underset{\text{Sign bit}}{1}\underbrace{1110100}_{\text{Code}}{}_2 = -12_{10}$

Two's Complement Shortcuts

- ***Algorithm 1***: Complement each bit then add 1 to the result

N = 01100101	[N] = 10011011
10011010	01100100
+ 1	+ 1
10011011	01100101

- ***Algorithm 2***: Starting with the least significant bit, copy all of the bits up to and including the first '1' bit, then complement the remaining bits

N = 0 1 1 0 0 1 1 0

[N] = 1 0 0 1 1 0 1 0

Two's Complement Representation

4 bits

$\Downarrow$

16 combinations

7	0111
6	0110
.	.
.	.
1	0001
0	0000
− 1	1111
− 2	1110
.	.
.	.
− 7	1001
− 8	1000

Finite-Precision Number Representation

- Machines that use 2's complement arithmetic can represent integers in the range

$$-2^{n-1} \le N \le 2^{n-1} - 1$$

n is the number of bits used for representing N

Note that $2^{n-1} - 1 = (011..11)_2$ and $-2^{n-1} = (100..00)_2$

- 2's complement code has more negative numbers than positive
- 1's complement code has 2 representations for zero
- For a n-bit number in base (i.e. radix) z there are z^n different unsigned values (combinations)

$$(0, 1, \ldots z^{n-1})$$

1's Complement Subtraction

- Using 1's complement representation, subtracting numbers is also easy

  Step 1: Take 1's complement of 2^{nd} operand

  Step 2: Add binary numbers

  Step 3: Add carry as a low order bit

- For example: $(+12)_{10} - (1)_{10}$

$(+12)_{10} = +(1100)_2$

$= 01100_2$

$(-1)_{10} = -(0001)_2$

$= 11110_2$ *in 1's comp.*

```
      0 1 1 0 0
  –   0 0 0 0 1
                        1's comp
      1 1
      0 1 1 0 0
+     1 1 1 1 0
Add   ---------
    1 0 1 0 1 0
Add carry     1
      ---------
Final Result
      0 1 0 1 1
```

2’s Complement Subtraction

- **Using 2’s complement representation, subtracting numbers is also easy**

  Step 1: Take 2’s complement of 2nd operand

  Step 2: Add binary numbers

  Step 3: Ignore the resulting carry bit

- **For example: $(+12)_{10} - (1)_{10}$**

  $(+12)_{10} = +(1100)_2$

  $= 01100_2$

  $(-1)_{10} = -(0001)_2$

  $= 11111_2$ *in 2’s comp.*

$$
\begin{array}{llcccccc}
 & & & 0 & 1 & 1 & 0 & 0 \\
\text{2's comp} & & & \boxed{-\ 0\ \ 0\ \ 0\ \ 0} & & & & 1 \\
 & & & 1 & 1 & & & \\
 & & & 0 & 1 & 1 & 0 & 0 \\
\text{Add} & + & & 1 & 1 & 1 & 1 & 1 \\
\hline
\text{Final Result} & & 1 & \boxed{0\ \ 1\ \ 0\ \ 1\ \ 1} & & & &
\end{array}
$$

Ignore Carry

2's Complement Subtraction

- **Example 2**: $(13)_{10} - (5)_{10}$

  $(13)_{10} = +(1101)_2 = (01101)_2$

  $(-5)_{10} = -(0101)_2 = (11011)_2$

- Adding these two 5-bit codes:

$$
\begin{array}{rr}
 & 0\,1\,1\,0\,1 \\
+ & 1\,1\,0\,1\,1 \\
\hline
\text{Carry} \longrightarrow \boxed{1} & 0\,1\,0\,0\,0
\end{array}
$$

- Discarding the carry bit, the sign bit is seen to be zero, indicating a positive result

  Indeed: $(01000)_2 = +(8)_{10}$

2's Complement Subtraction

- **Example 3**: $(5)_{10} - (12)_{10}$

  $(5)_{10} = +(0101)_2 = (00101)_2$

  $(-12)_{10} = -(1100)_2 = (10100)_2$

- Adding these two 5-bit codes:

$$
\begin{array}{rr}
 & 0\,0\,1\,0\,1 \\
+ & \underline{1\,0\,1\,0\,0} \\
\text{Carry} \longrightarrow \boxed{0} & 1\,1\,0\,0\,1
\end{array}
$$

- Here, there is no carry bit and the sign bit is 1. This indicates a negative result, which is what we expect: $(11001)_2 = -(7)_{10}$

Summary

- Binary numbers can also be represented in octal and hexadecimal
- Easy to convert between binary, octal, and hexadecimal
- Signed numbers are represented in 3 codes: signed magnitude, 1's complement, or 2's complement
- 2's complement code is most important (only 1 representation for zero)
- Important to understand the treatment of the sign bit for 1's and 2's complement codes

Introductory Concepts

(Codes)

Binary Coded Decimal

- *Binary Coded Decimal* (BCD) represents each decimal digit with four bits

  Ex. $\underbrace{0011}_{3}\ \underbrace{0010}_{2}\ \underbrace{1001}_{9} = 329_{10}$

- This is NOT the same as 001100101001_2
- Why do this? Because people think in decimal

Digit	BCD Code	Digit	BCD Code
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Putting It All Together

- BCD is not very efficient
- Used in early computers (1940s, 1950s)
- Used to encode numbers for seven-segment displays
- Easier to read?

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

Gray Code

- Gray code is not a number system

  It is an alternate way to represent four bit data

- Only one bit changes from one decimal digit to the next
- Useful for reducing errors in communication
- Can be scaled to larger numbers

Digit	Binary	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

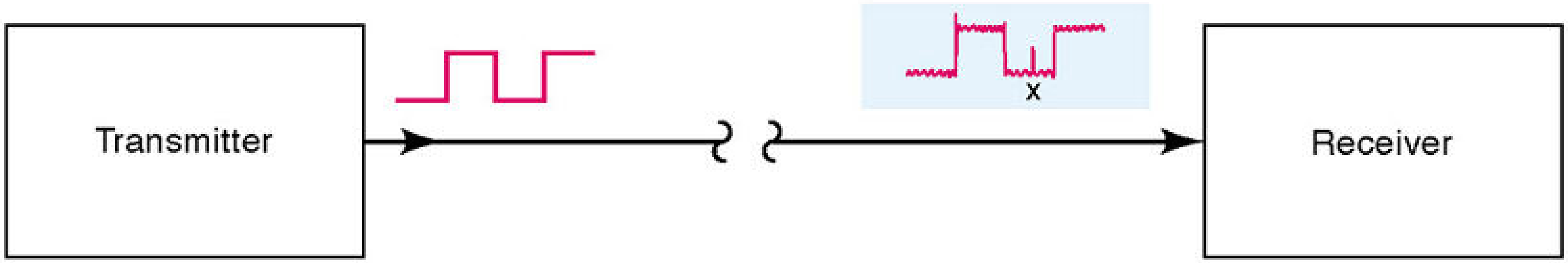
ASCII Code

- **American Standard Code for Information Interchange**
- **ASCII is a 7-bit code, frequently used with a 8th bit for error detection (more about that later)**

Character	ASCII (bin)	ASCII (hex)	Decimal	Octal
A	1000001	41	65	101
B	1000010	42	66	102
C	1000011	43	67	103
…				
Z				
a				
…				
1				
‘				

ASCII Codes and Data Transmission

- ASCII Codes
  - A – Z (26 codes), a – z (26 codes)
  - 0 – 9 (10 codes), others (@#$%^&*….)
- Transmission susceptible to noise
- Typical transmission rates (1500 Kbps, 56.6 Kbps)
- How to keep data transmission accurate?



Parity Codes

- Parity codes are formed by concatenating a ***parity bit***, ***P*** to each code word ***C***
- In an ***even-parity*** code, the parity bit is specified so that the total number of ones is even
- In an ***odd-parity*** code, the parity bit is specified so that the total number of ones is odd

P	Information Bits

```
1 1 0 0 0 0 1 1               0 1 0 0 0 0 1 1
↑                             ↑
Added even parity bit         Added odd parity bit
```

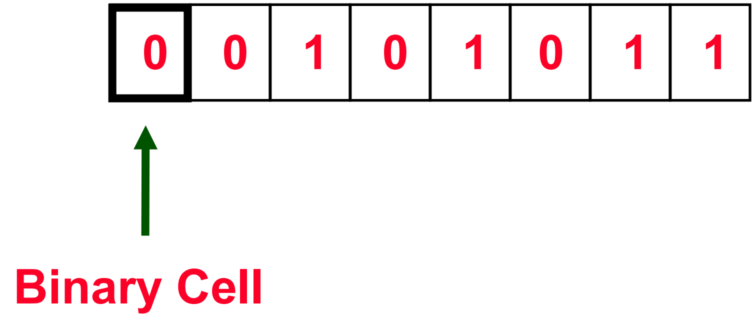
Parity Code Example

Concatenate a parity bit to the ASCII code for the characters “0”, “X”, and “=” to produce both odd-parity and even-parity codes

Character	ASCII	Odd-Parity ASCII	Even-Parity ASCII
0	0110000	10110000	00110000
X	1011000	01011000	11011000
=	0111100	10111100	00111100

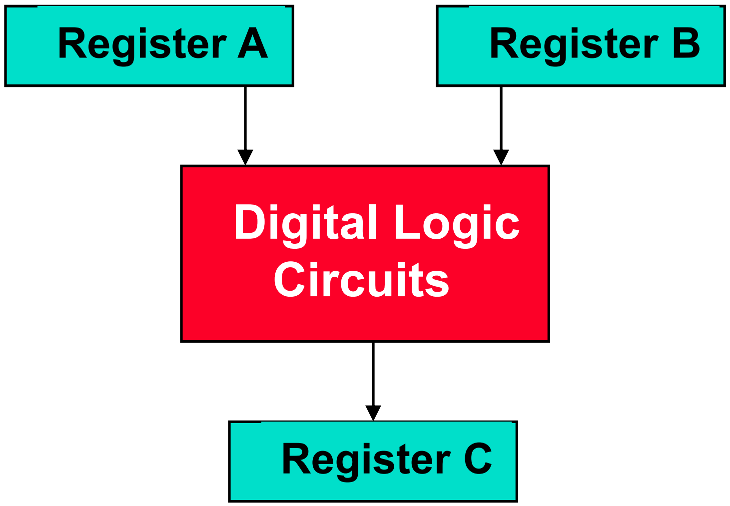
Binary Data Storage

- Binary *cells* store individual bits of data
- Multiple cells form a register
- Data in registers can indicate different values
  - Hex (binary)
  - BCD
  - ASCII



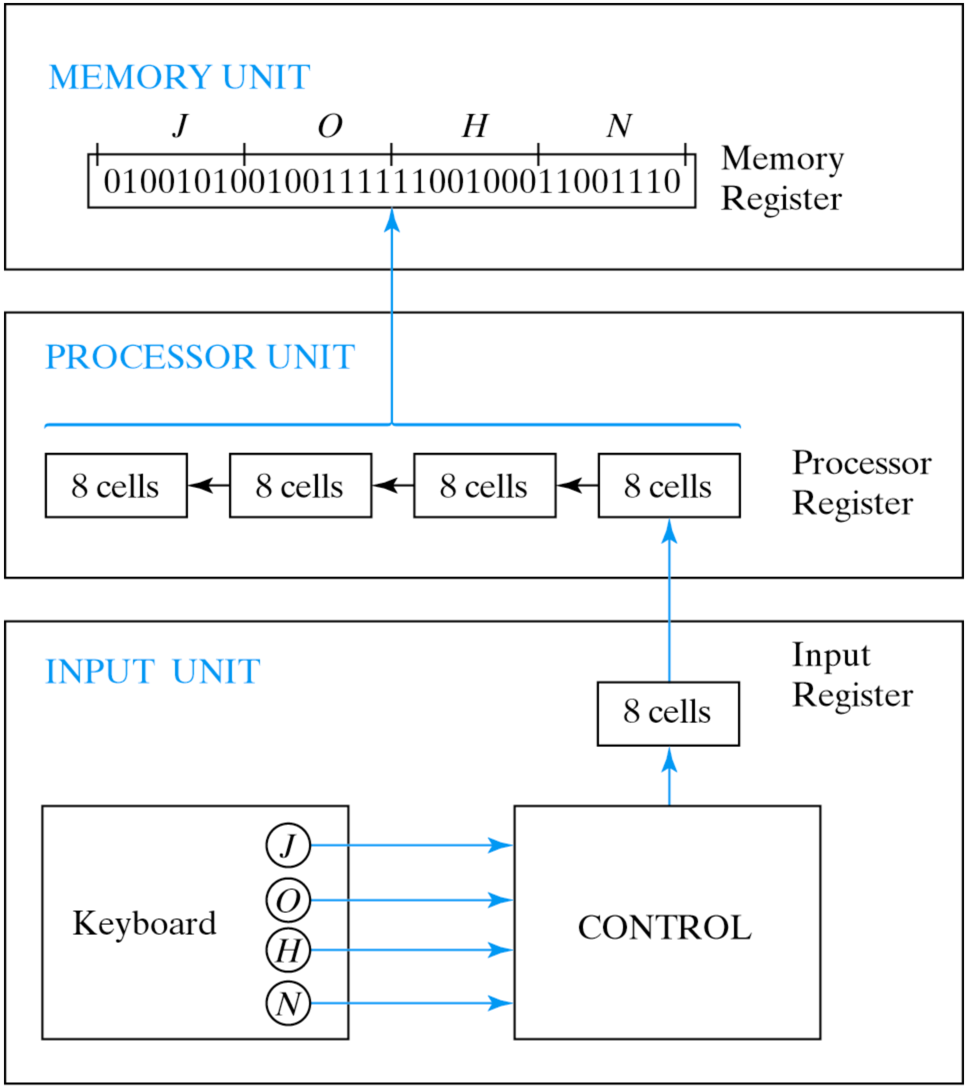
Register Transfer

- Data can move from a register to a register
- Digital logic used to process data



Transfer of Information

- Data input at keyboard
- Shifted into place
- Stored in memory



NOTE: Data input in ASCII

Fig. 1-1 Transfer of information with registers

Building a Computer

- We need processing
- We need storage
- We need communication

- You will learn to use and design these components

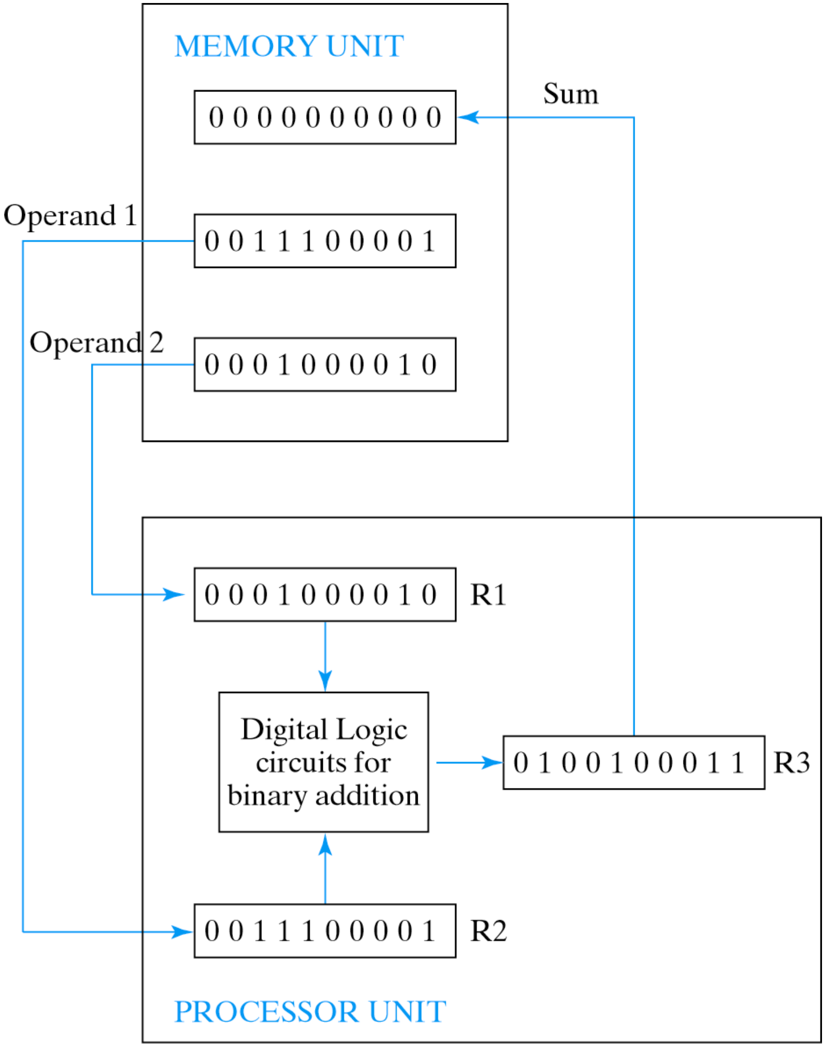


Fig. 1-2 Example of binary information processing

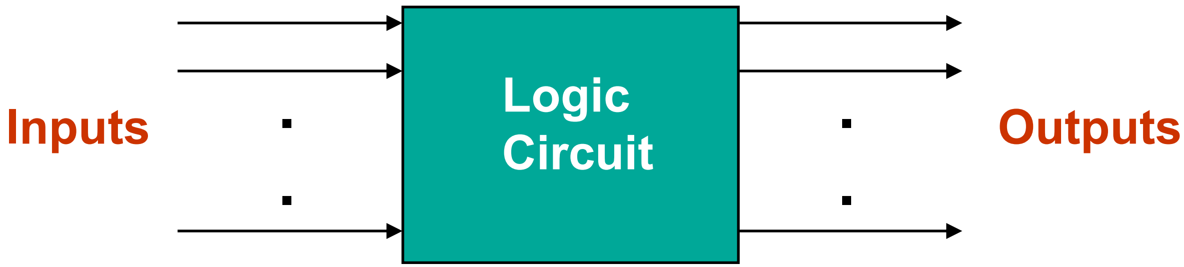
Summary

- Although 2's complement is most important, other number codes exist
- ASCII code is used to represent characters (such as those on the keyboard)
- Registers store binary data

Unit-II
Boolean Algebra and Logic gates

Digital Systems

- *Analysis* problem:

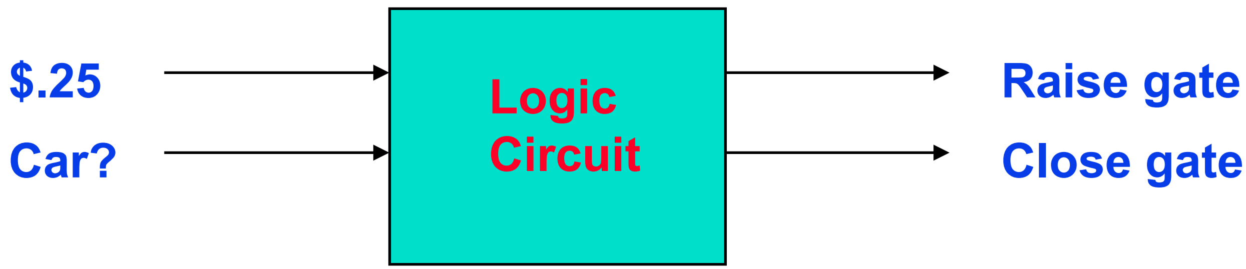


  - Determine the binary output for each input combination

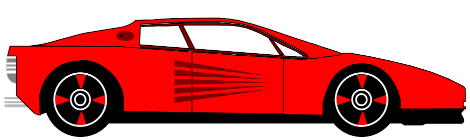
- *Design* problem: given a task, develop a circuit that accomplishes that task
  - Many possible implementations
  - “Best” circuit: based on some criterion (size, power, performance, etc.)

Toll Booth Controller

- Consider the design of a toll booth controller
- Inputs: quarter, car sensor
- Outputs: gate-lift signal, gate-close signal

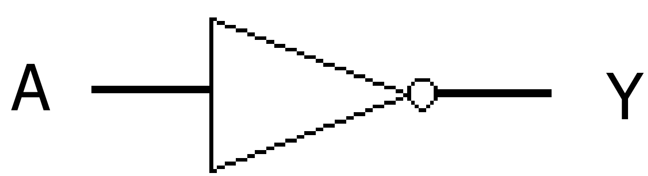


- If driver pitches in quarter, raise gate
- When car has cleared gate, close gate



Describing Circuit Functionality: Inverter

- Basic logic functions have symbols
- The same functionality can be represented with a truth table
  - Truth table completely specifies outputs for all input combinations
- This is an inverter
  - An input of 0 is inverted to a 1
  - An input of 1 is inverted to a 0



Symbol

Truth Table

A	Y
0	1
1	0

Input

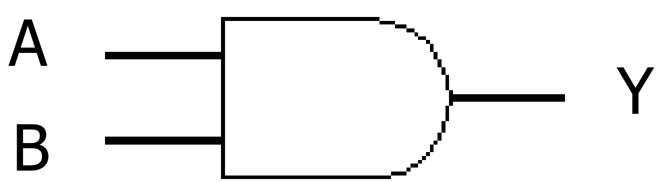
Output

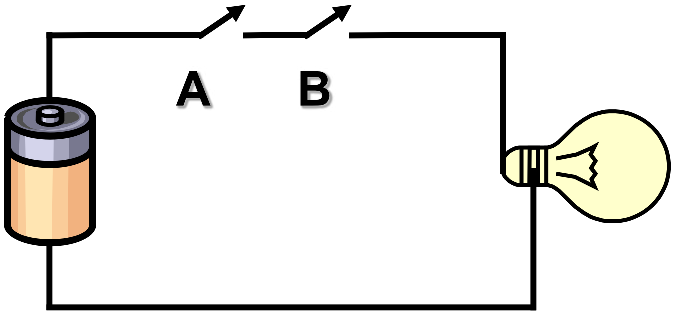
The AND Gate

- This is an AND gate
- If the two input signals are *asserted* (i.e. high) the output will also be asserted. Otherwise, the output will be *deasserted* (i.e. low)

Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

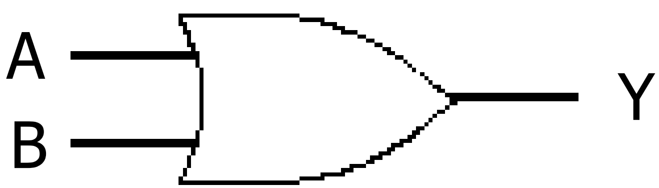


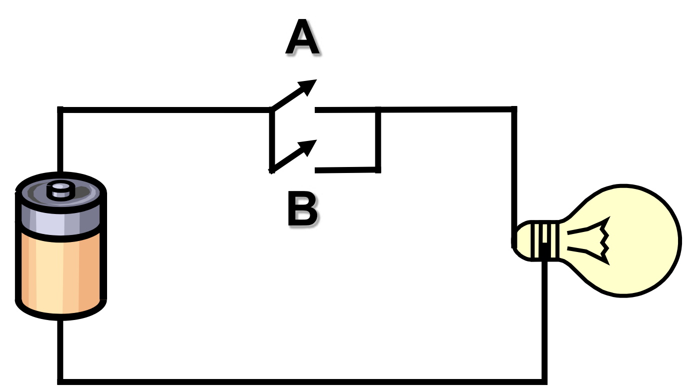


The OR Gate

- This is an **OR** gate
- If either of the two input signals is ***asserted***, or both of them are, the output will be ***asserted***

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1





Describing Circuit Functionality: Waveforms

- Waveforms provide another approach for representing functionality
- Values are either high (logic 1) or low (logic 0)
- Can you create a truth table from the waveforms?

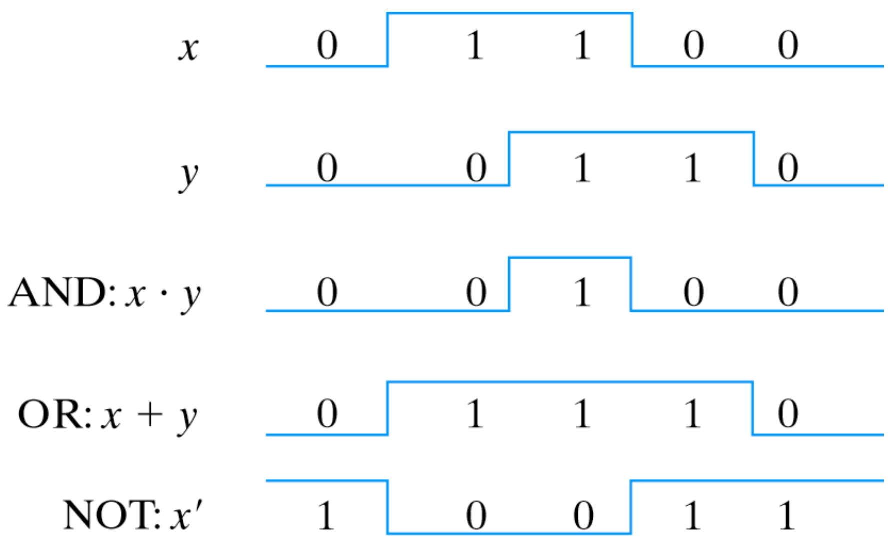


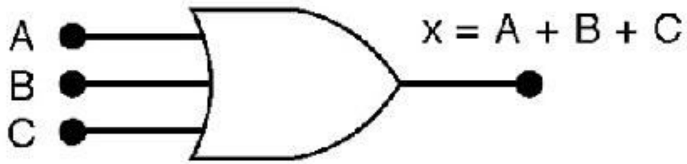
Fig. 1-5 Input-output signals for gates

AND Gate

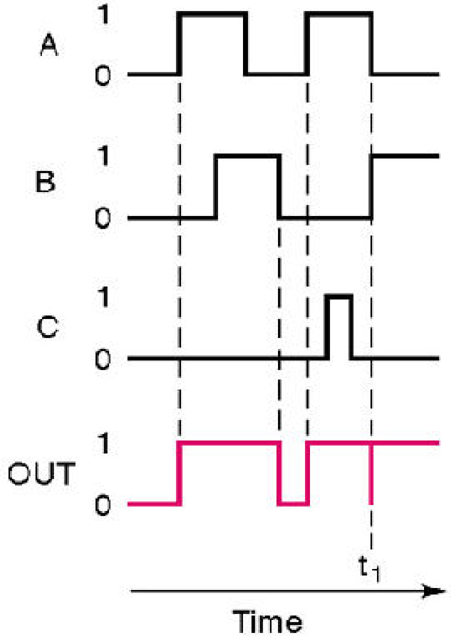
x	y	f
0	0	0
0	1	0
1	0	0
1	1	1

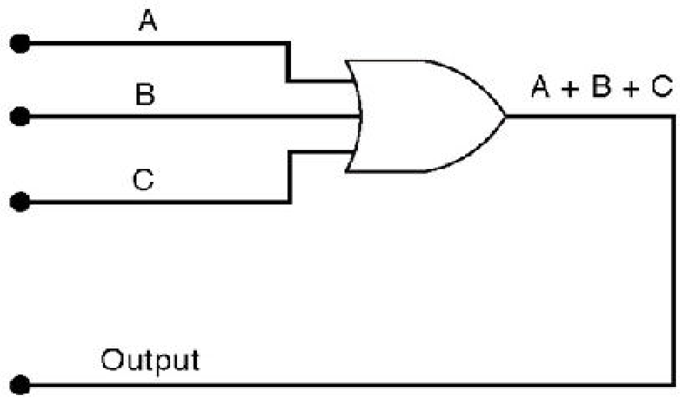
Consider three-input gates

3 Input OR Gate



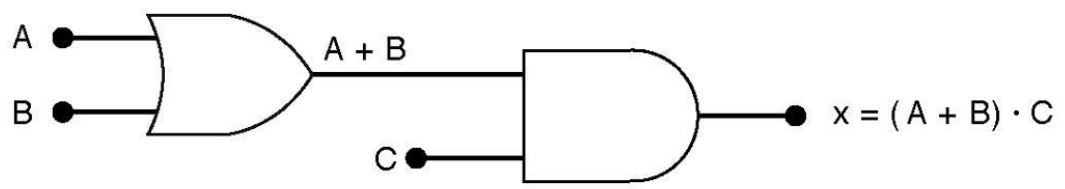
A	B	C	x = A + B + C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1





Ordering Boolean Functions

- How to interpret A • B + C?
  - Is it A • B ORed with C ?
  - Is it A ANDed with B + C ?
- Order of precedence for Boolean algebra: AND before OR
- Note that parentheses are needed here:



Boolean Algebra

- A Boolean algebra is defined as a closed algebraic system containing a set K of two or more elements and the two operators, • and +
- Useful for identifying and *minimizing* circuit functionality
- Identity elements
  - $a + 0 = a$
  - $a \cdot 1 = a$
- 0 is the identity element for the + operation
- 1 is the identity element for the • operation

Commutativity and Associativity of the Operators

- **Commutative Property:**

  For every ‘a’ and ‘b’ in K,

  - $a + b = b + a$
  - $a \bullet b = b \bullet a$

- **Associative Property:**

  For every ‘a’, ‘b’, and ‘c’ in K,

  - $a + (b + c) = (a + b) + c$
  - $a \bullet (b \bullet c) = (a \bullet b) \bullet c$

Distributivity of the Operators and Complements

- **Distributive Property:**

  For every 'a', 'b', and 'c' in K,

  - $a + (b \cdot c) = (a + b) \cdot (a + c)$
  - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

- **The Existence of the Complement:**

  For every 'a' in K there exists a unique element called a' (or $\bar{a}$) (complement of a) such that,

  - $a + a' = 1$
  - $a \cdot a' = 0$

- To simplify notation, the • operator is frequently omitted. When two elements are written next to each other, the AND (•) operator is implied

  - $a + b \cdot c = (a + b) \cdot (a + c)$
  - $a + bc = (a + b)(a + c)$

Duality

- The principle of duality is an important concept: If an expression is valid in Boolean algebra, the dual of that expression is also valid
- To form the dual of an expression, replace all + operators with • operators, all • operators with + operators, all ones with zeros, and all zeros with ones
- Form the dual of the equation:
  - $a + (bc) = (a + b)(a + c)$

  Following the replacement rules:

  - $a(b + c) = ab + ac$
- Take care not to alter the location of the parentheses if they are present

Involution

- This theorem states:

  $a'' = a$ $\qquad\qquad$ $\overline{\overline{a}} = a$

- Remember that:

  $aa' = 0$ $\qquad\qquad$ $a\,\overline{a} = 0$

  $a+a'=1$ $\qquad\qquad$ $a + \overline{a} = 1$

  - Therefore, a’ is the complement of a and a is also the complement of a’

- Taking the double inverse of a value produces the initial value

Absorption

- This theorem states:

$a + ab = a \qquad\qquad\qquad a(a+b) = a$

- To prove the first half of this theorem:

$$
\begin{aligned}
a + ab &= a \bullet 1 + ab \\
&= a\,(1 + b) \\
&= a\,(b + 1) \\
&= a\,(1) \\
a + ab &= a
\end{aligned}
$$

DeMorgan's Theorem

- A key theorem in simplifying Boolean algebra expressions is DeMorgan's Theorem. It states:

  $(a + b)' = a'b'$ $\qquad\qquad$ $(ab)' = a' + b'$

  $\overline{a + b} = \overline{a} \bullet \overline{b}$ $\qquad\qquad$ $\overline{a \bullet b} = \overline{a} + \overline{b}$

- Example: Complement and simplify the expression

  $a(b + z(x + a'))$

$$\overline{a\,(b + z\,(x + a'))} = \overline{a} + \overline{(b + z\,(x + a'))}$$

$$= \overline{a} + \overline{b}\;\overline{(z\,(x + a'))}$$

$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{(x + a')})$$

$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{x}\;\overline{\overline{a}})$$

$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{x}\,a)$$

Summary

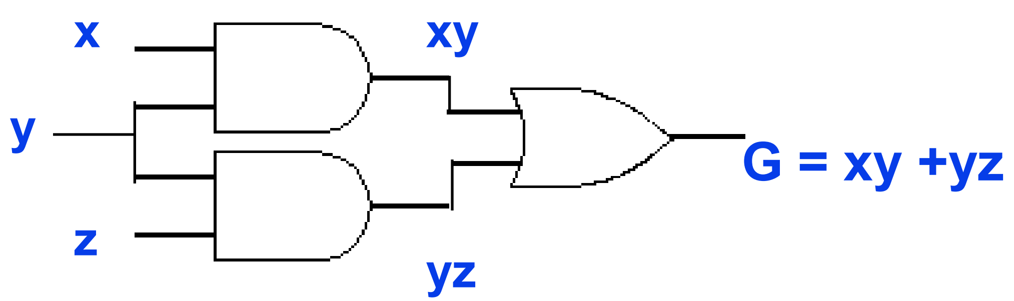
- Basic logic functions can be made from AND, OR, and NOT (invert) functions
- The behavior of digital circuits can be represented with waveforms, truth tables, or symbols
- Primitive gates can be combined to form larger circuits
- Boolean algebra defines how binary variables can be combined
- Rules for associativity, commutativity, and distribution are similar to algebra
- DeMorgan's rules are important
  - Will allow us to reduce circuit sizes

UNIT-II
Boolean Algebra and Logic gates

Boolean Functions

- Boolean algebra deals with binary variables and logic operations
- Function results in binary 0 or 1

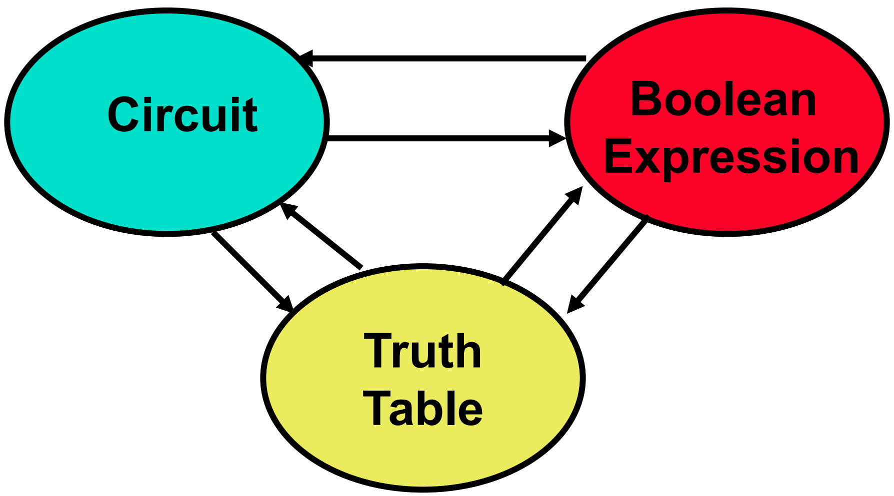
x	y	z	xy	yz	G
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	1	1



How to transit between an equation, a circuit, and a truth table?

Representation Conversion

- Need to transit between a Boolean expression, a truth table, and a circuit (symbols)
- Conversion between truth table and expression is easy
- Conversion between expression and circuit is easy
- Conversion to truth table is more difficult



Truth Table to Expression

- Converting a truth table to an expression
  - Each row with an output of 1 becomes a "product term"
  - Sum the "product terms" together

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

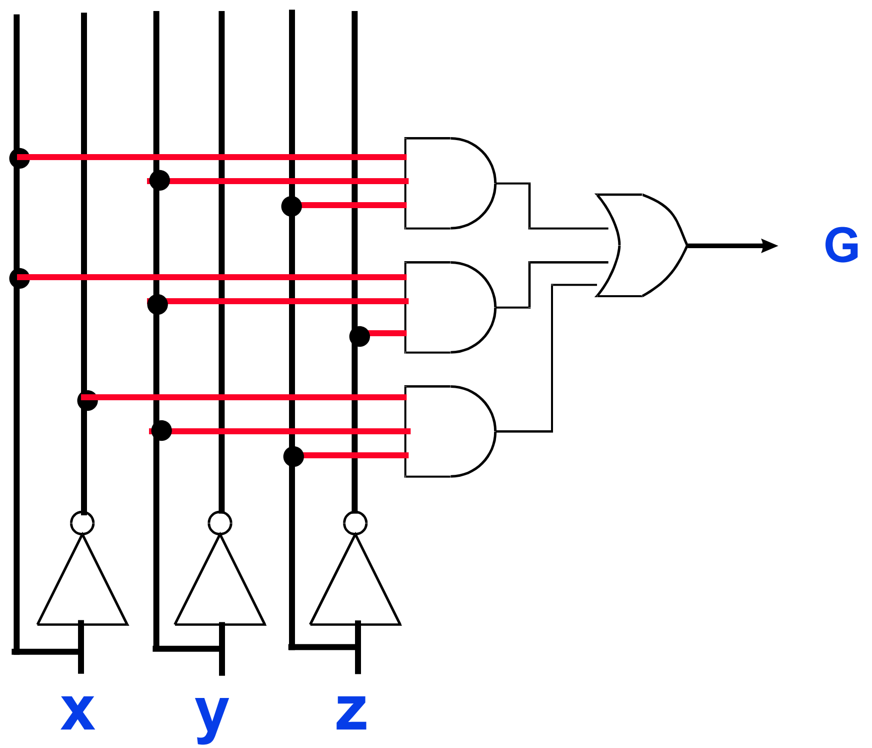
Any Boolean Expression can be represented in sum of products form!

$xyz + xyz' + x'yz$

Equivalent Representations of Circuits

- All three formats are equivalent
- Number of 1's in truth table output column equals AND terms for Sum-of-Products (SOP)

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



$G = xyz + xyz' + x'yz$

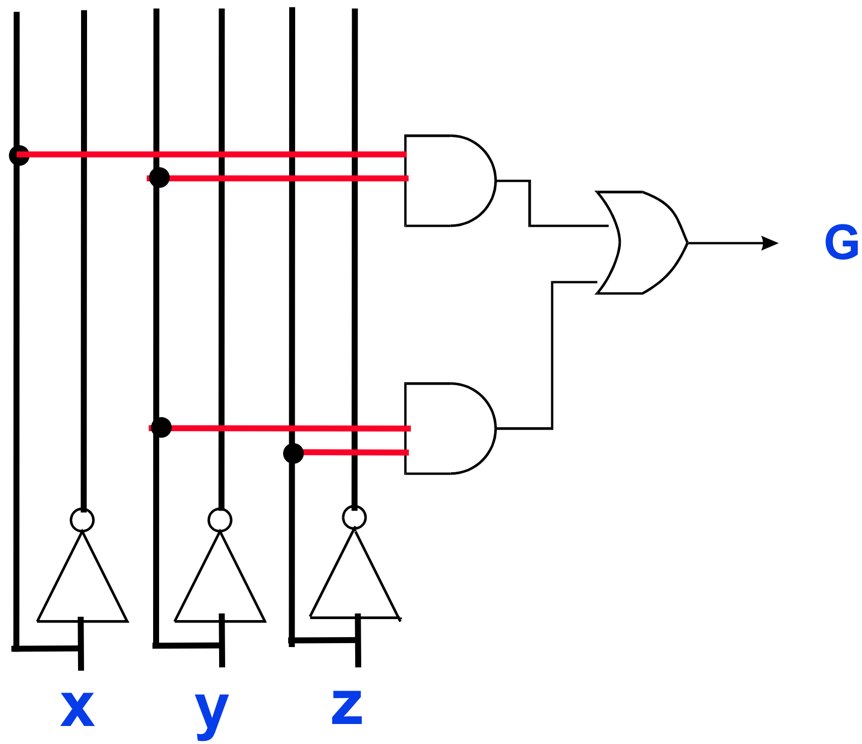
Reducing Boolean Expressions

- Is this the smallest possible implementation of this expression? **No!** $G = xyz + xyz' + x'yz$
- Use Boolean Algebra rules to reduce complexity while preserving functionality
- Step 1: Use Theorem 1 ($a + a = a$)
  - $xyz + xyz' + x'yz = xyz + xyz + xyz' + x'yz$
- Step 2: Use distributive rule $a(b + c) = ab + ac$
  - $xyz + xyz + xyz' + x'yz = xy(z + z') + yz(x + x')$
- Step 3: Use Postulate 3 ($a + a' = 1$)
  - $xy(z + z') + yz(x + x') = xy.1 + yz.1$
- Step 4: Use Postulate 2 ($a . 1 = a$)
  - $xy.1 + yz.1 = xy + yz = xyz + xyz' + x'yz$

Reduced Hardware Implementation

- Reduced equation requires less hardware!
- Same function is implemented!

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



$G = xyz + xyz' + x'yz = xy + yz$

Minterms and Maxterms

- Each variable in a Boolean expression is a *literal*
- Boolean variables can appear in normal (x) or complemented form (x')
- Each **AND** combination of terms is a *minterm*
- Each **OR** combination of terms is a *maxterm*

For example:

x	y	z	Minterm	
0	0	0	x'y'z'	m_0
0	0	1	x'y'z	m_1
…				
1	0	0	xy'z'	m_4
…				
1	1	1	xyz	m_7

For example:

x	y	z	Maxterm	
0	0	0	x+y+z	M_0
0	0	1	x+y+z'	M_1
…				
1	0	0	x'+y+z	M_4
…				
1	1	1	x'+y'+z'	M_7

Representing Functions with Minterms

- Minterm number is same as row position in truth table (starting with 0 at the top)
- Shorthand way to represent functions

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

$$G = xyz + xyz' + x'yz$$

$$\downarrow$$

$$G = m_7 + m_6 + m_3 = \Sigma(3, 6, 7)$$

Complementing Functions

- **Minterm number is same as row position in truth table (starting with 0 at the top)**
- **Shorthand way to represent functions**

x	y	z	G	G'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0

$G = xyz + xyz' + x'yz$

$G' = (xyz + xyz' + x'yz)' = \ ?$

Can we find a simpler representation?

Complementing Functions

Step 1: assign temporary names

- $b + c \rightarrow z$
- $(a + z)' = G'$

$G = a + b + c$

$G' = (a + b + c)'$

Step 2: Use DeMorgans' Law

- $(a + z)' = a' \cdot z'$

Step 3: Resubstitute (b+c) for z

- $a' \cdot z' = a' \cdot (b + c)'$

Step 4: Use DeMorgans' Law

- $a' \cdot (b + c)' = a' \cdot (b' \cdot c')$

$G = a + b + c$

$G' = a' \cdot b' \cdot c' = a'b'c'$

Step 5: Associative rule

- $a' \cdot (b' \cdot c') = a' \cdot b' \cdot c'$

Complementation Example

- Find complement of $F = x'z + yz$

  $F' = (x'z + yz)'$
- DeMorgan's

  $F' = (x'z)' \cdot (yz)'$
- DeMorgan's

  $F' = (x''+z')\,(y'+z')$
- Reduction → eliminate double negation on x

  $F' = (x+z')\,(y'+z')$

This format is called product of sums

Conversion Between Canonical Forms

- Easy to convert between *minterm* and *maxterm* representations
- For *maxterm* representation, select rows with 0's

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

$G = xyz + xyz' + x'yz$

$\downarrow$

$G = m_7 + m_6 + m_3 = \Sigma(3, 6, 7)$

$\downarrow$

$G = M_0 M_1 M_2 M_4 M_5 = \Pi(0,1,2,4,5)$

$\downarrow$

$G = (x+y+z)(x+y+z')(x+y'+z)(x'+y+z)(x'+y+z')$

Representation of Circuits

- Any logic expression can be represented in a 2-level circuit
- Circuits can be reduced to minimal 2-level representations
- Sum−of−products representation is most common in industry

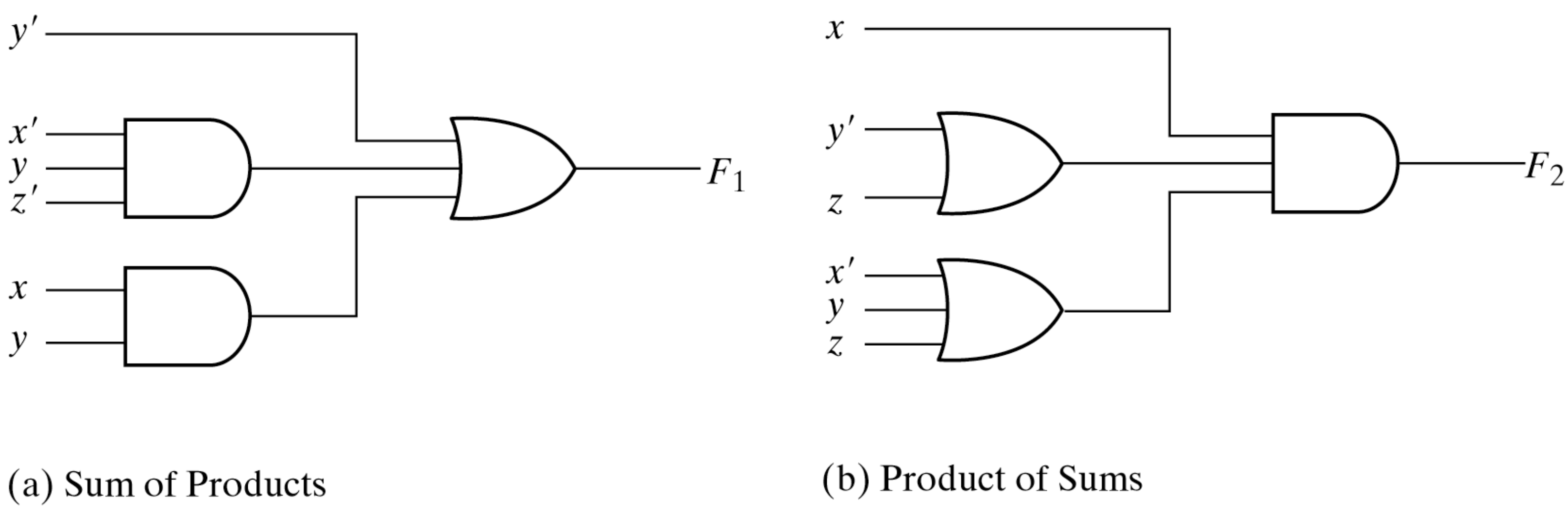


Fig. 2-3 Two-level implementation

Summary

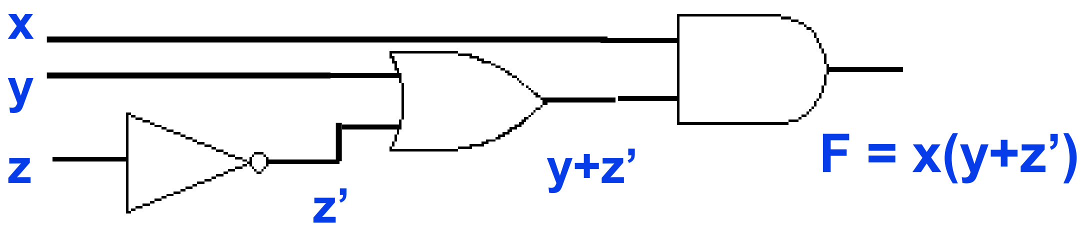
- Truth table, circuit, and Boolean expression formats are equivalent
- Easy to translate a truth table to SOP and POS representations
- Boolean algebra rules can be used to reduce circuit size while maintaining functionality
- All logic functions can be made from AND, OR, and NOT
- Easiest way to understand: Do examples!

UNIT-II
Boolean Algebra and Logic Gates

Boolean Functions

- Boolean algebra deals with binary variables and logic operations
- Function results in binary 0 or 1

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



F = x(y+z’)

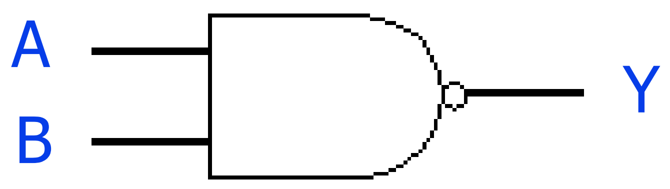
Logic functions of N variables

- Each truth table represents one possible function (AND, OR … etc)
- If there are N inputs, there are 2^{2^N}
- For example, if N is 2 then there are 16 possible truth tables
- So far, we have defined 2 of these functions
  - 14 more are possible
- Why consider new functions?
  - Cheaper hardware, more flexibility

x	y	G
0	0	0
0	1	0
1	0	0
1	1	1

The NAND Gate

- The NAND gate is a combination of an AND gate followed by an inverter
- NAND gates have several interesting properties…
  - NAND(a,a) → (aa)’ = a’ → NOT(a)
  - NAND’(a,b) → (ab)’’ = ab → AND(a,b)
  - NAND(a’,b’) → (a’b’)’ = a+b → OR(a,b)



$$Y = \overline{A\,B}$$

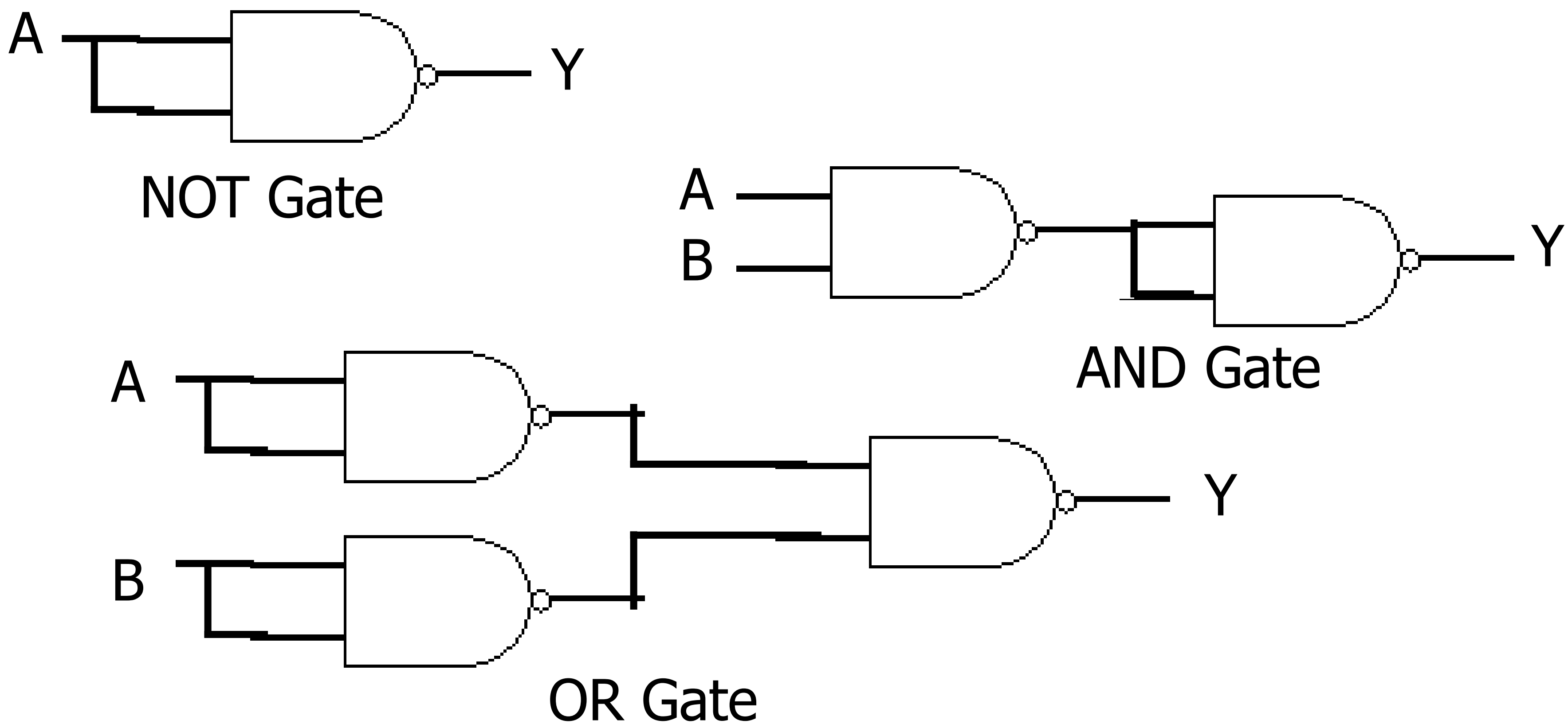
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

The NAND Gate

- Those three properties show that:
  - a NAND gate with both of its inputs driven by the same signal is equivalent to a NOT gate
  - a NAND gate whose output is complemented is equivalent to an AND gate
  - a NAND gate with complemented inputs acts as an OR gate
- Hence, we can use a NAND gate to implement all three of the *elementary* operators (AND, OR, NOT)
- Therefore, ANY switching function can be constructed using only NAND gates. Such a gate is said to be *primitive* or *functionally complete (Universal Gate)*

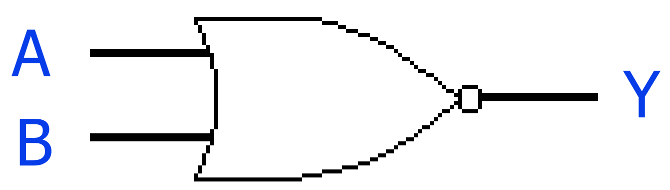
NAND Gates into Other Gates

What are these circuits?



The NOR Gate

- A NOR gate is a combination of an OR gate followed by an inverter
- NOR gates also have several interesting properties…
  - NOR(a,a) → (a+a)’ = a’ → NOT(a)
  - NOR’(a,b) → (a+b)’’ = a+b → OR(a,b)
  - NOR(a’,b’) → (a’+b’)’ = ab → AND(a,b)



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

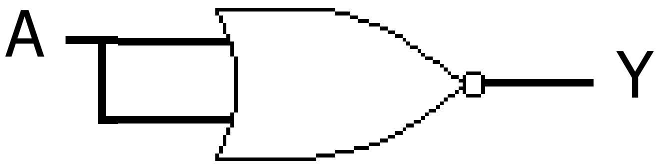
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Functionally Complete Gates

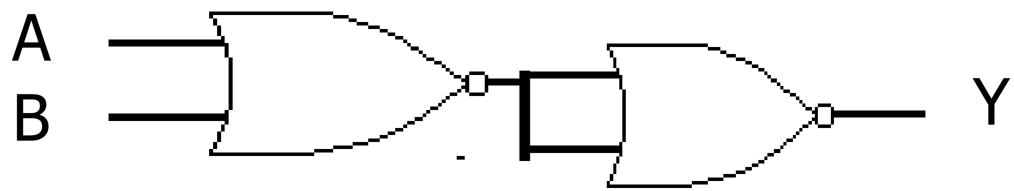
- Just like the NAND gate, the NOR gate is functionally complete…any logic function can be implemented using just NOR gates
- Both NAND and NOR gates are very valuable as any design can be realized using either one
- It is easier to build an IC chip using all NAND or NOR gates than to combine AND, OR, and NOT gates
- NAND/NOR gates are typically faster in switching and cheaper to produce

NOR Gates into Other Gates

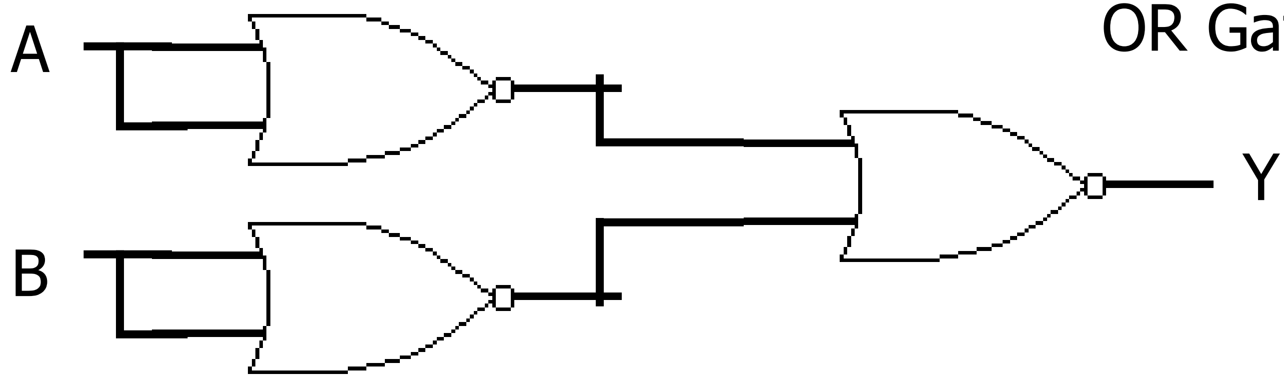
What are these circuits?



NOT Gate



OR Gate



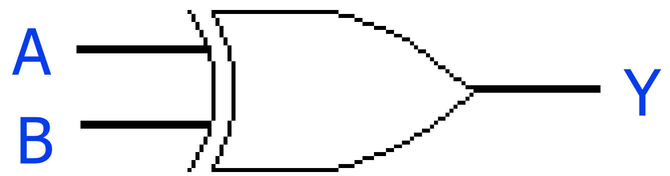
AND Gate

The XOR Gate (Exclusive-OR)

- This is a XOR gate
- XOR gates assert their output when exactly one of the inputs is asserted, hence the name
- The switching algebra symbol for this operation is ⊕:
  $1 \oplus 1 = 0$ and $1 \oplus 0 = 1$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = A \oplus B$$



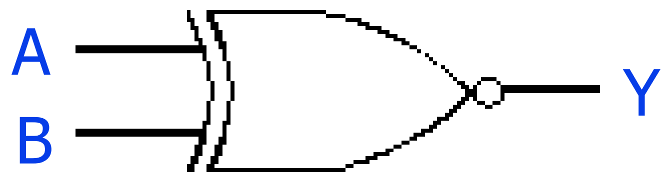
The XNOR Gate

- This is a XNOR gate
- This functions as an exclusive-NOR gate, or simply the complement of the XOR gate
- The switching algebra symbol for this operation is ⊙:

  1 ⊙ 1 = 1 and 1 ⊙ 0 = 0

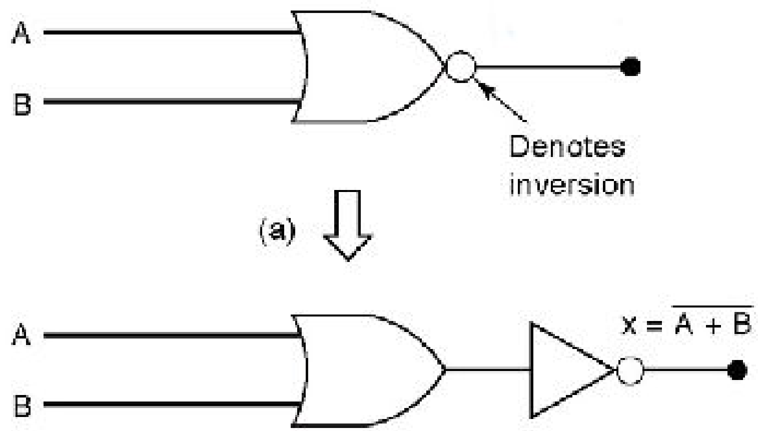
$$Y = A \odot B$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1



NOR Gate Equivalence

NOR Symbol, Equivalent Circuit, Truth Table



(a)

(b)

		OR	NOR
A	B	$A + B$	$\overline{A + B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

(c)

DeMorgan's Theorem

- A key theorem in simplifying Boolean algebra expression is DeMorgan's Theorem. It states:

  (a + b)' = a'b' $\qquad\qquad$ (ab)' = a' + b'

  $$\overline{a + b} = \overline{a} \bullet \overline{b} \qquad\qquad \overline{a \bullet b} = \overline{a} + \overline{b}$$

- Example: Complement and simplify the expression

  a(b + z(x + a'))

$$\overline{a\,(b + z\,(\,x + a'))} = \overline{a} + \overline{(b + z\,(x + a'))}$$

$$= \overline{a} + \overline{b}\,\overline{(z\,(x + a'))}$$

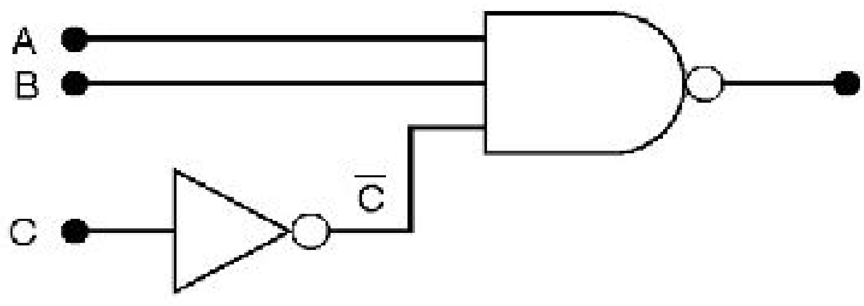
$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{(x + a')})$$

$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{x}\,\overline{\overline{a}})$$

$$= \overline{a} + \overline{b}\,(\overline{z} + \overline{x}\,a)$$

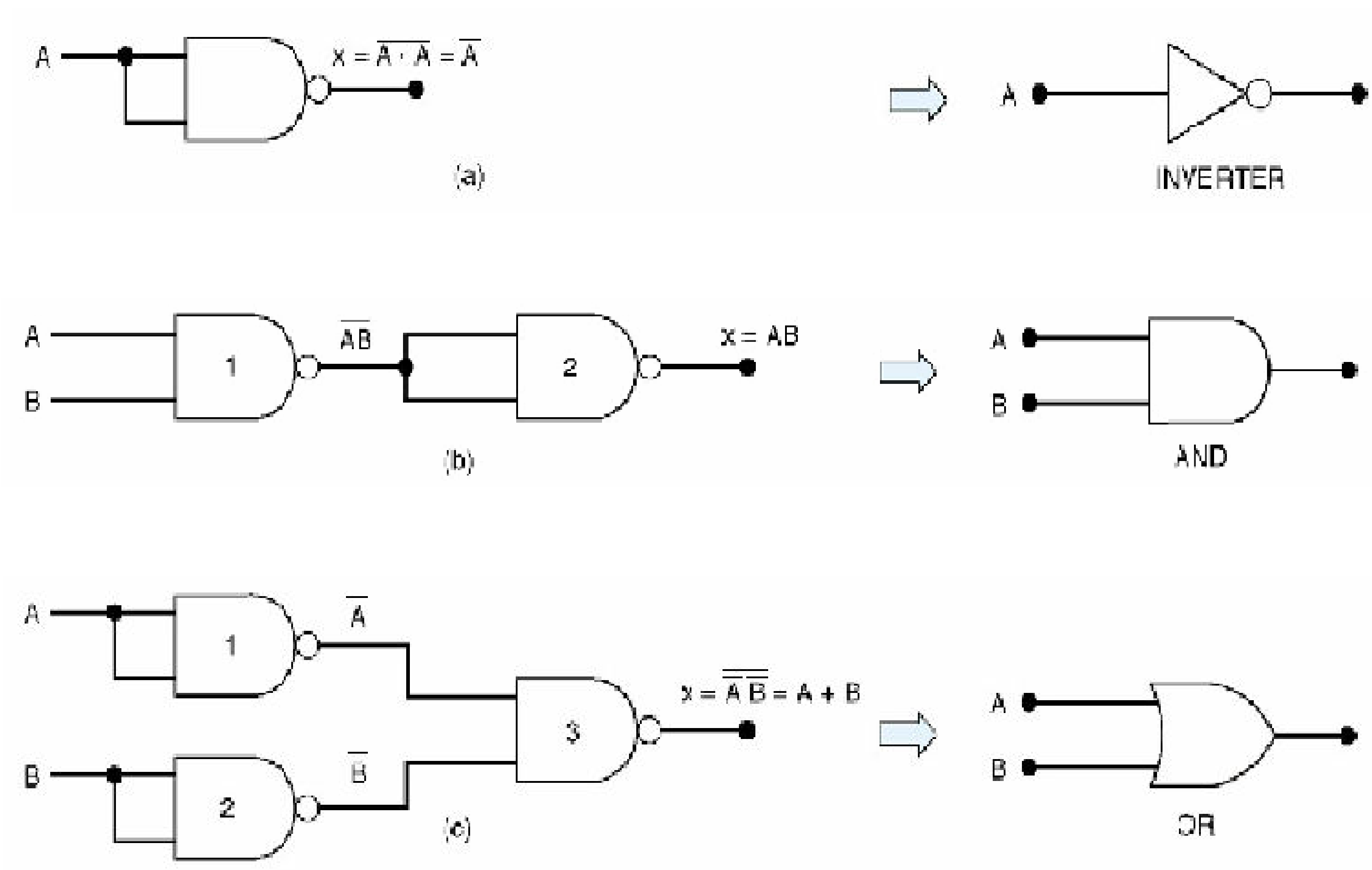
Example

Determine the output expression for the following circuit and simplify it using DeMorgan’s Theorem

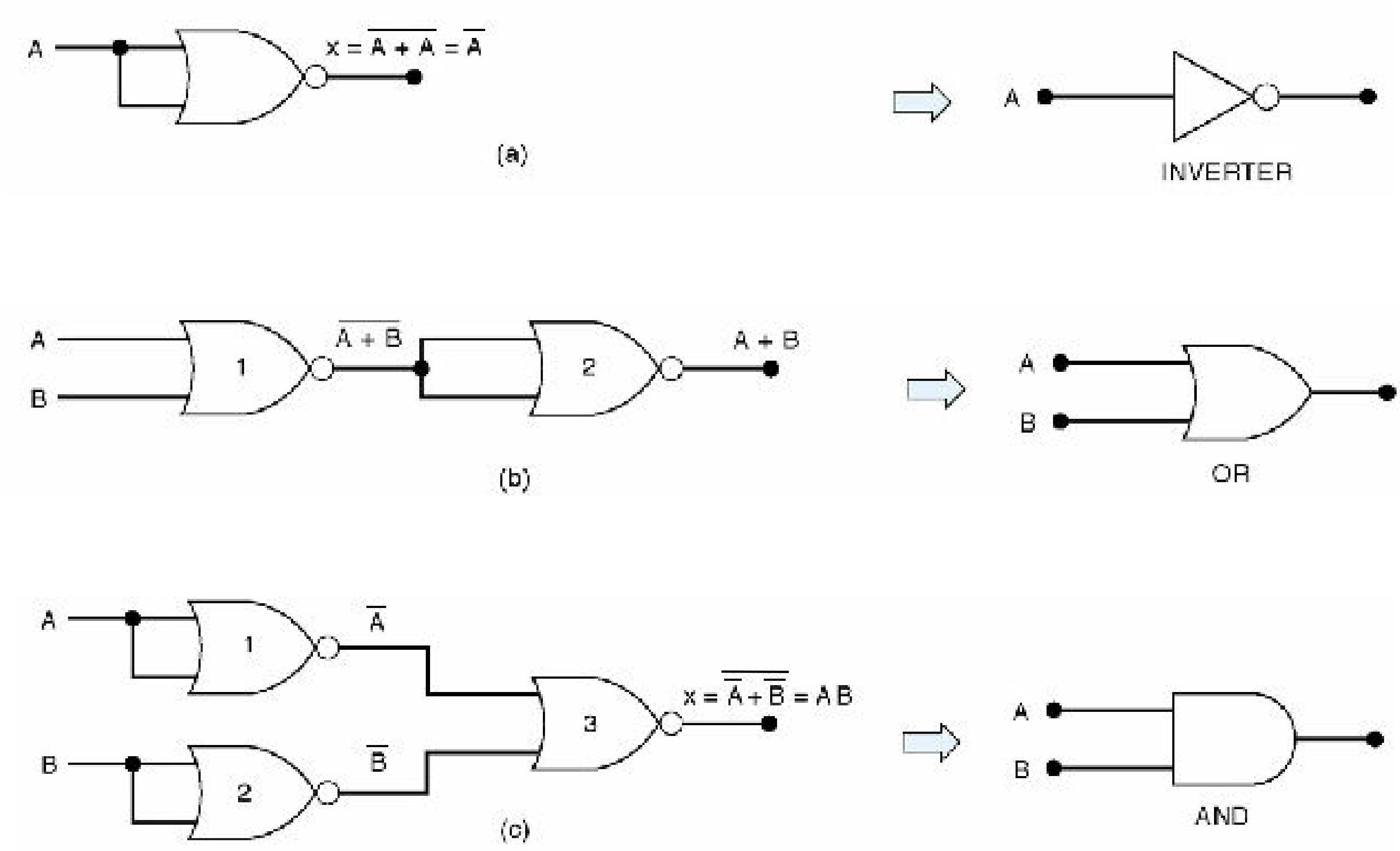


$$z = \overline{A \cdot B \cdot \overline{C}} = \overline{A} + \overline{B} + \overline{\overline{C}} = \overline{A} + \overline{B} + C$$

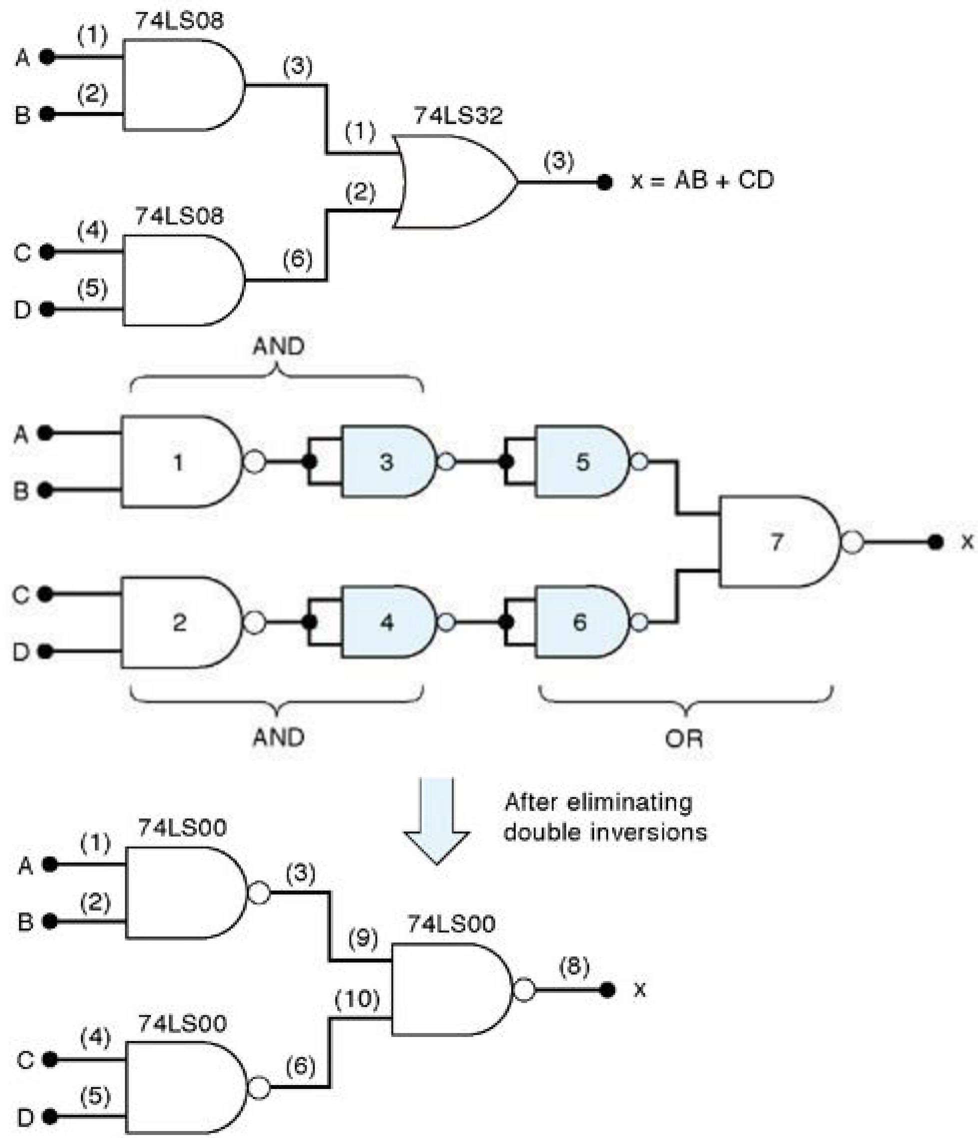
Universality of NAND gate



Universality of NOR gate



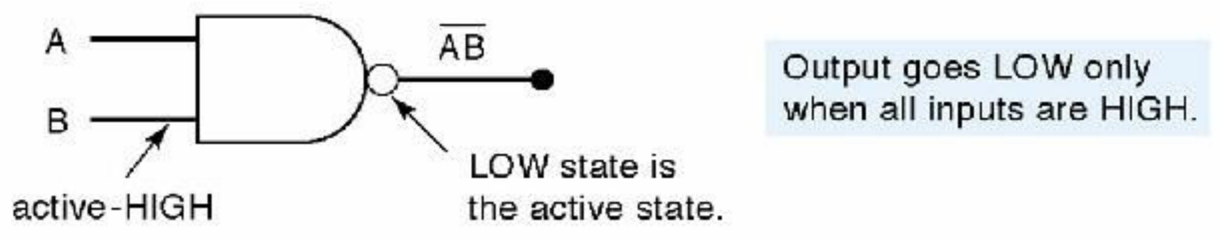
Example



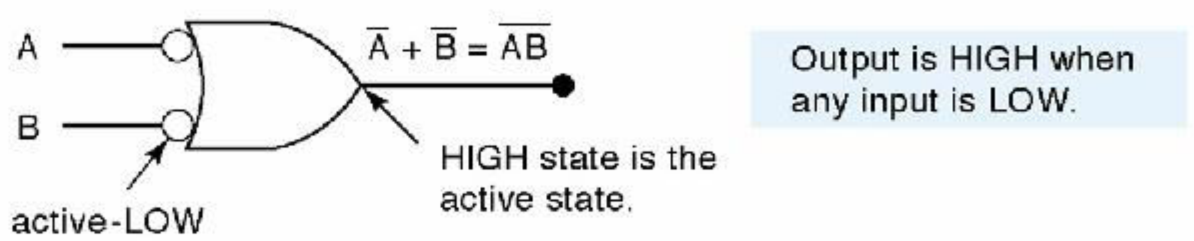
74LS08
(1)
A
(2)
B
(3)
74LS32
(1)
(2)
(3)
x = AB + CD
74LS08
(4)
C
(5)
D
(6)
AND
A
B
1
3
5
7
x
C
D
2
4
6
AND
OR
After eliminating
double inversions
74LS00
(1)
A
(2)
B
(3)
74LS00
(9)
(10)
(8)
x
74LS00
(4)
C
(5)
D
(6)

Interpretation of the two NAND gate symbols

DeMorgan's Theorem



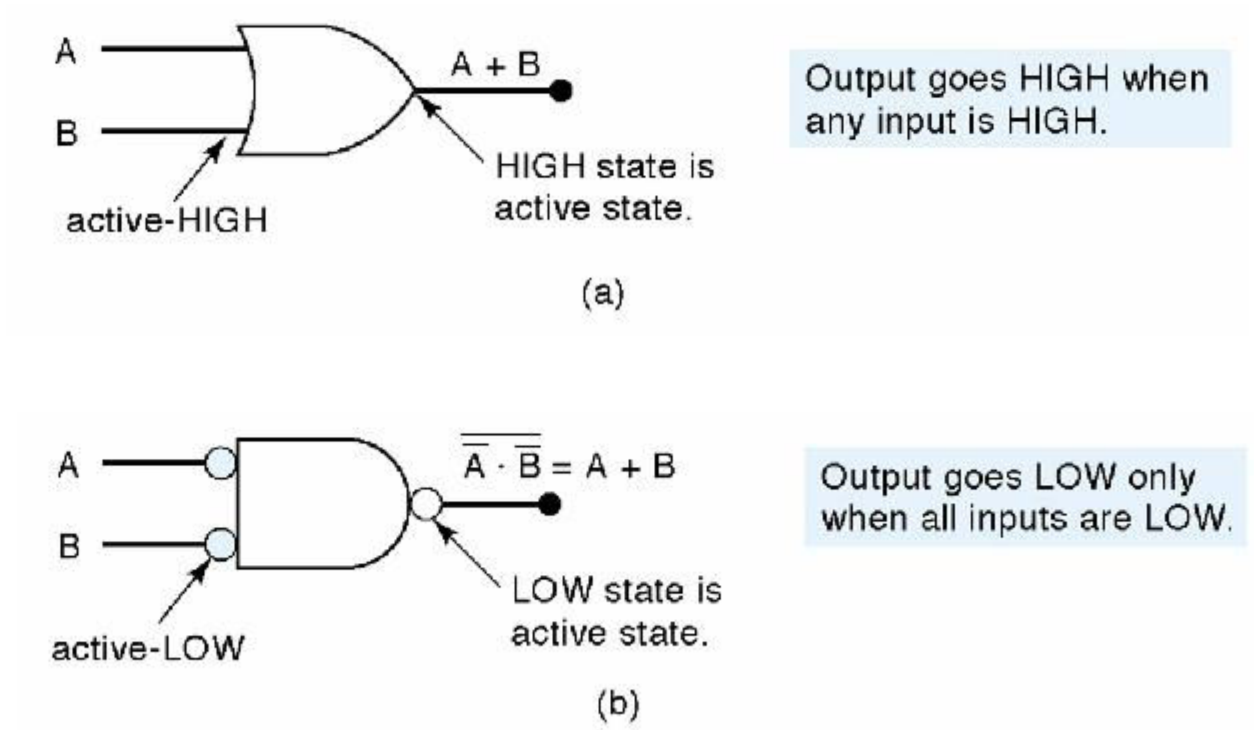
(a)



(b)

Interpretation of the two OR gate symbols

DeMorgan's Theorem



Summary

- Basic logic functions can be made from NAND, and NOR functions
- The behavior of digital circuits can be represented with waveforms, truth tables, or Boolean expressions
- Primitive gates can be combined to form larger circuits
- Boolean algebra defines how binary variables can be combined with NAND, NOR
- DeMorgan’s rules are important

Allow conversion to NAND/NOR representations

K-MAP

Karnaugh maps

- Alternate way of representing Boolean functions
- A Karnaugh map is a graphical tool for assisting in the general simplification procedure
  - Each row in the truth table is represented by a square
  - Each square represents a *minterm*

x \ y	0	1
0	$x'y'$	$x'y$
1	xy'	xy

x \ y	0	1
0	1	1
1	0	0

x	y	F
0	0	1
0	1	1
1	0	0
1	1	0

$$F = \Sigma(m_0, m_1) = x'y + x'y'$$

Karnaugh Maps

- **Two variable maps**

A \ B	0	1
0	0	1
1	1	0

$$F = A\bar{B} + \bar{A}B$$

A \ B	0	1
0	0	1
1	1	1

$$F = AB + \bar{A}B + A\bar{B}$$

- **Three variable maps**

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

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

$$F = A\bar{B}\bar{C} + A\bar{B}C + ABC + AB\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C}$$

Karnaugh maps

Numbering scheme is based on Gray code

- e.g. 00, 01, 11, 10
- Only a single bit changes in code for adjacent map cells
- Observe the variable transitions

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

$G(A,B,C) = B$

A \ BC	00	01	11	10
0	0	1	3	2
1	4	5	7	6

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

$F(A,B,C) = \Sigma m(0,2,6,7) = A'C' + AB$

Karnaugh Maps

- **Two variable maps**

A \ B	0	1
0	0	1
1	1	0

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

A \ B	0	1
0	0	1
1	1	1

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

$$F = A + B$$

- **Three variable maps**

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

$$F = A + \overline{B}C + B\overline{C}$$

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

More Karnaugh Map Examples

Examples

a \ b	0	1
0	0	1
1	0	1

f = b

a \ b	0	1
0	1	1
1	0	0

f = a'

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

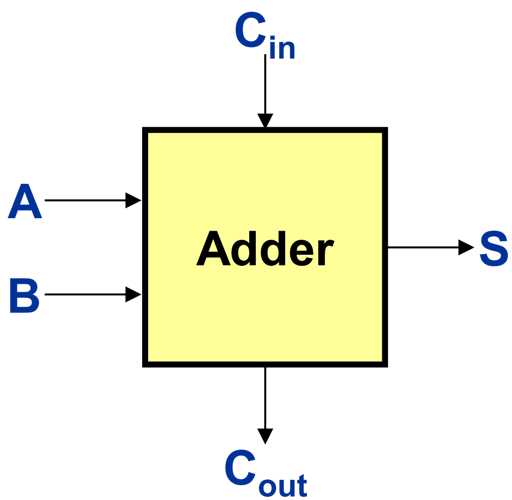
cout = ac+bc + ab

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

f = b

1. Circle the largest groups possible
2. Group dimensions must be a power of 2
3. Remember what circling means!

Application of Karnaugh Maps: The One-bit Adder



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

How to use a Karnaugh Map instead of the Algebraic simplification?

$$S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$$

$$C_{out} = A'BC_{in} + A\,B'C_{in} + ABC_{in}' + ABC_{in}$$

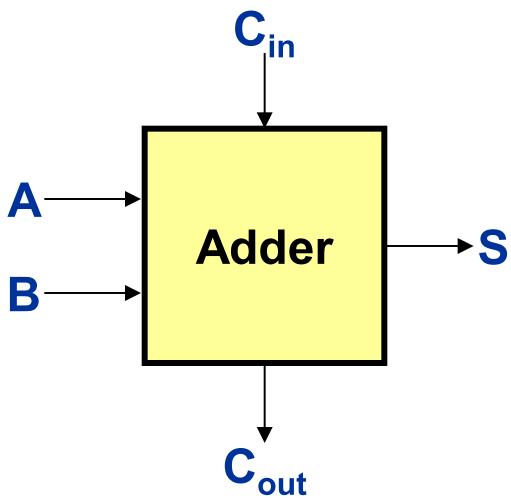
$$= A'BC_{in} + ABC_{in} + AB'C_{in} + ABC_{in} + ABC_{in}' + ABC_{in}$$

$$= (A' + A)BC_{in} + (B' + B)AC_{in} + (C_{in}' + C_{in})AB$$

$$= 1 \cdot BC_{in} + 1 \cdot AC_{in} + 1 \cdot AB$$

$$= BC_{in} + AC_{in} + AB$$

Application of Karnaugh Maps: The One-bit Adder



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

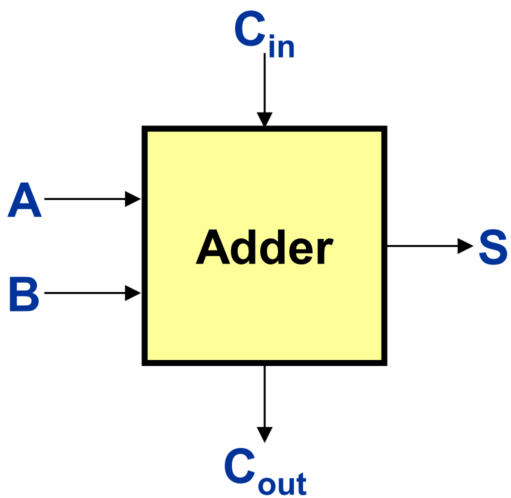
A \ BC	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Karnaugh Map for C_{out}

Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

$$C_{out} = BC_{in} + AB + AC_{in}$$

Application of Karnaugh Maps: The One-bit Adder



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

A \ BC	00	01	11	10
0	0	$\boxed{1}$	0	$\boxed{1}$
1	$\boxed{1}$	0	$\boxed{1}$	0

Karnaugh Map for S

Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

$$S = A\,B'\,C'_{in} + A'B'C_{in} + A\,B\,C_{in} + A'\,BC'_{in}$$

No Possible Reduction!

Summary

- Karnaugh map allows us to represent functions with new notation
- Representation allows for logic reduction
  - Implement same function with less logic
- Each square represents one minterm
- Each circle leads to one product term
- Not all functions can be reduced

K-MAP

Karnaugh Maps for 4–Input Functions

- **Represent functions of 4 inputs with 16 minterms**
- **Use same rules developed for 3-input functions**

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

(a)

				y	y	
	wx \ yz	00	01	11	10	
	00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$	
	01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$	x
w	11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$	x
w	10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$	
			z	z		

(b)

Fig. 3-8 Four-variable Map

$$F(A,B,C,D) = \Sigma m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$$

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

F = $C + A'BD + B'D'$

Design Examples

K-map for LT

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

F = $A'C + A'B'D + B'CD$

Design Examples

K-map for EQ

				C	C	
	AB \ CD	00	01	11	10	
	00	1	0	0	0	
	01	0	1	0	0	B
A	11	0	0	1	0	B
A	10	0	0	0	1	
			D	D		

F = $A'B'C'D' + A'BC'D + ABCD + AB'CD'$

Design Examples

K-map for GT

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

C, D, B, A

F = $AC' + BC'D' + ABD'$

Physical Implementation

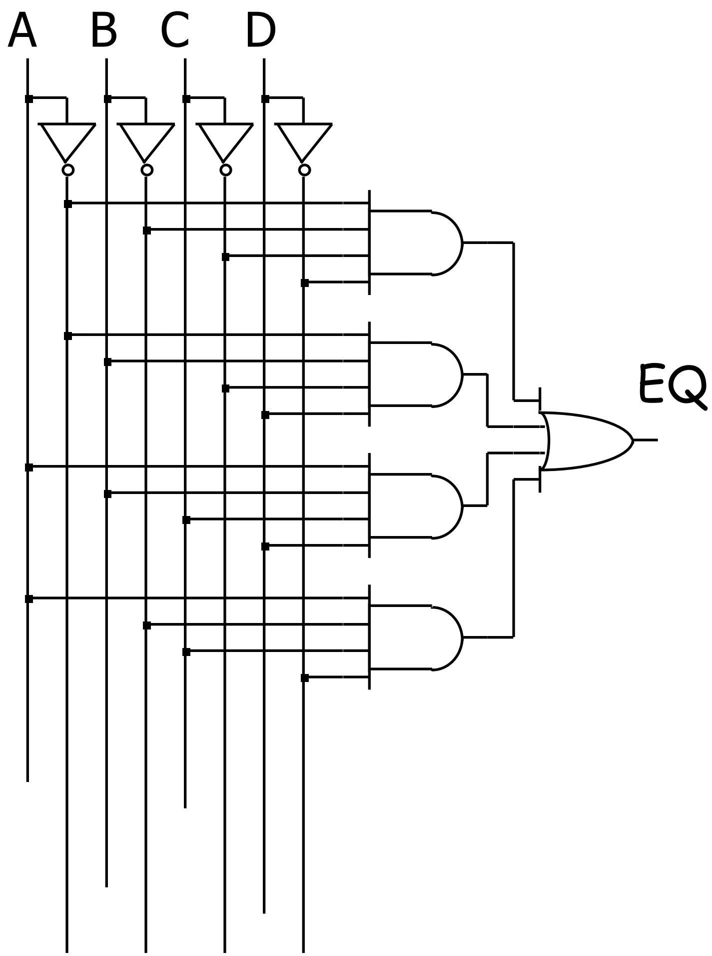
Step 1: Truth table

Step 2: K-map

Step 3: Minimized sum-of-products

Step 4: Physical implementation
with gates

Physical Implementation



K-map for EQ

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

$$F = A'B'C'D' + A'BC'D + ABCD + AB'CD'$$

Karnaugh Maps

- **Four variable maps**

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

$$F = A'BC' + A'CD' + ABC + AB'C'D' + ABC' + AB'C$$

$$F = BC' + AC + CD' + AD'$$

- **Need to make sure all 1's are covered**
- **Try to minimize total product terms**
- **Design could be implemented using NANDs and NORs**

Karnaugh Maps: Don’t Cares

- In some cases, outputs are undefined
- We “don’t care” if the circuit produces a ‘0’ or a ‘1’
- This knowledge can be used to simplify functions

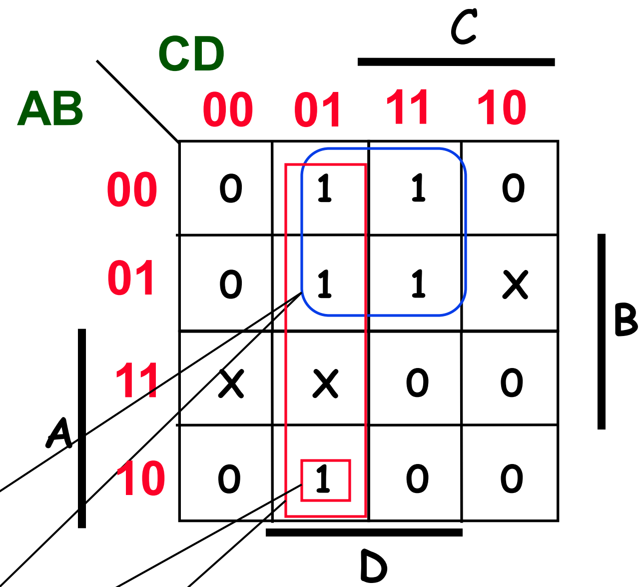
CD \ AB	00	01	11	10
00	0	0	X	0
01	1	1	X	1
11	1	1	0	0
10	0	X	0	0

A B C D

- Treat X’s like either 1’s or 0’s
- Very useful
- OK to leave some X’s uncovered

Karnaugh Maps: Don't Cares

$$F(A,B,C,D) = \Sigma\,(1,3,5,7,9) + d(6,12,13)$$



A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	X
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	X
1	1	0	1	X
1	1	1	0	0
1	1	1	1	0

$F = A'D + B'C'D$ Without don't cares

$F = A'D + C'D$ With don't cares

Don’t Care Conditions

- In some situations, we don’t care about the value of a function for certain combinations of the variables
  - these combinations may be impossible in certain contexts
  - or the value of the function may not matter when the combinations occur
- In such situations we say the function is incompletely specified and there are multiple (completely specified) logic functions that can be used in the design
  - so we can select a function that gives the simplest circuit
- When constructing the terms in the simplification procedure, we can choose to either cover or not cover the don’t care conditions

Map Simplification with Don't Cares

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

$$F = A'C'D + B + AC$$

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

$$F = A'B'C'D + BC + AC + ABC'$$

Alternative covering:

Karnaugh Maps: Product of Sums

$$F(A,B,C,D) = \Sigma\,(2,3,9,11,13) + d(6,14)$$

AB \ *CD*	00	01	11	10
00	0	0	1	1
01	0	0	0	x
11	0	1	0	x
10	0	1	1	0

$$F = AC'D + AB'D + A'B'C$$

Karnaugh Maps: Product of Sums

G(*A*,*B*,*C*,*D*) = Σ (0,1,4,5,7,8,10,12,15) + d(6,14)

AB \ *CD*	00	01	11	10
00	1	1	0	0
01	1	1	1	x
11	1	0	1	x
10	1	0	0	1

G = AD‵ + A‵C‵ + BC

Karnaugh Maps: Product of Sums

$$F(A,B,C,D) = \Sigma\,(2,3,9,11,13) + d(6,14)$$

AB \ *CD*	00	01	11	10
00	0	0	1	1
01	0	0	0	x
11	0	1	0	x
10	0	1	1	0

$F = AC'D + A'B'C + AB'D$

$F' = (B'+C')\,(A+C)(A'+D)$

Prime Implicants

Any single 1 or group of 1s in the Karnaugh map of a function F is an implicant of F.

A product term is called a prime implicant of F if it cannot be combined with another term to eliminate a variable.

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

(a) A’B’C
(b) BD
(c) A’B’C’D’
(d) A’C
(e) A’B’D’

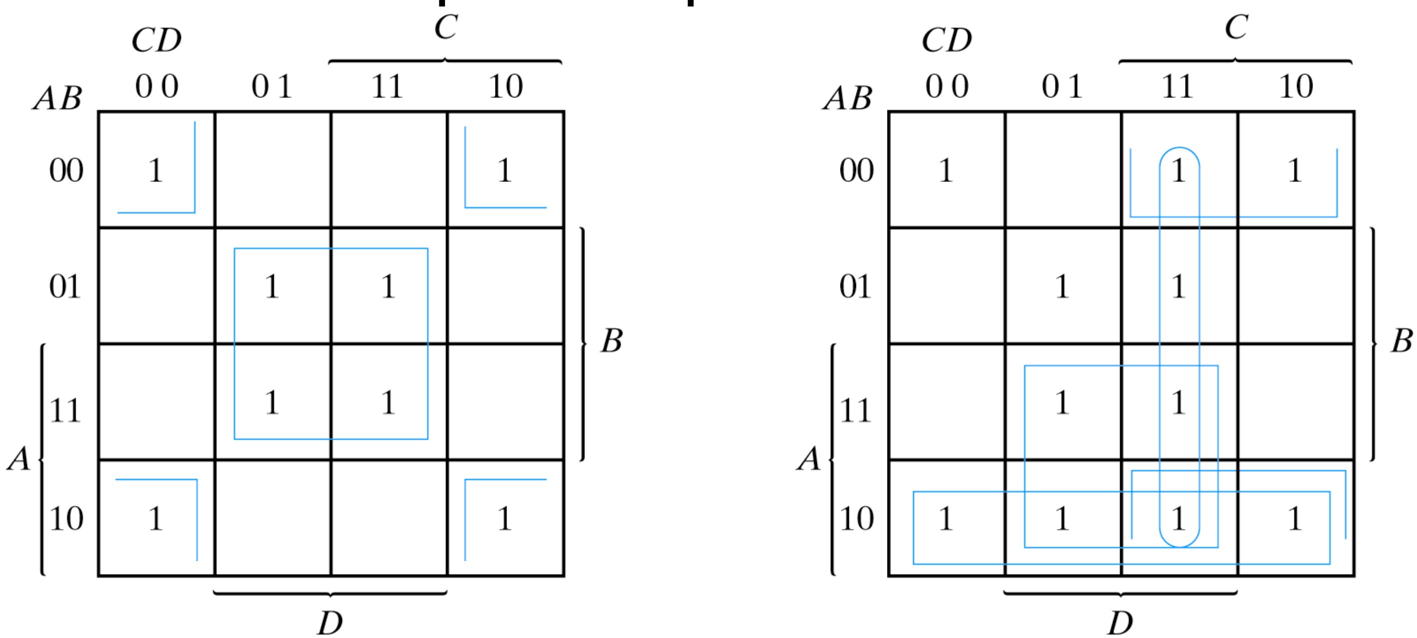
Implicants:
(a),(c),(d),(e)

Prime Implicants:
(d),(e)

Essential Prime Implicants

A product term is an essential prime implicant if there is a minterm that is only covered by that prime implicant

The minimal sum-of-products form of F must include all the essential prime implicants of F

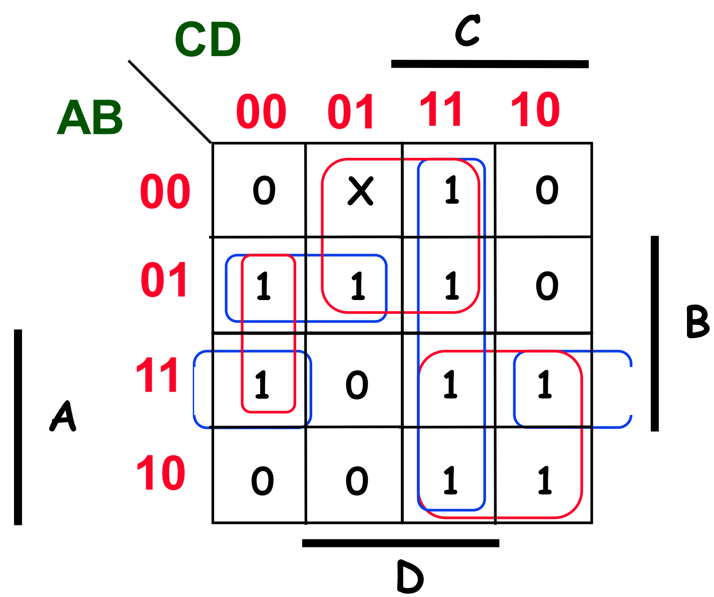


(a) Essential prime implicants BD and B′D′

(b) Prime implicants CD, B′C AD, and AB′

Fig. 3-11 Simplification Using Prime Implicants

Examples to Illustrate Terms



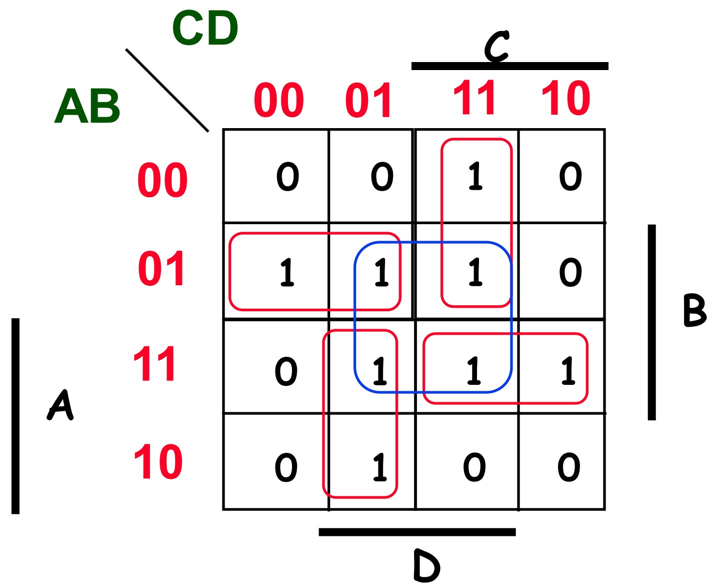
6 prime implicants:

ABD', A'D, AC, A'BC', CD, BC'D'

essential

minimum cover: AC + A'D + BC'D'

Examples to Illustrate Terms



5 prime implicants:

BD, ABC, AC'D, A'BC', A'CD

essential

minimum cover: 4 essential implicants

Summary

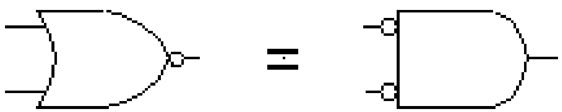
- K-maps of four literals were considered
  - Larger examples exist
- Don’t care conditions help minimize functions
  - Output for don’t cares are originally undefined
- Result of minimization is a minimal sum-of-products
- Result contains prime implicants
- Essential prime implicants are required in the implementation

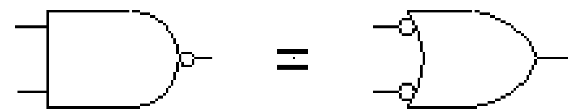
NAND-NAND & NOR-NOR Networks

DeMorgan’s Law:

$(a + b)' = a'\,b'$ $\qquad$ $(a\,b)' = a' + b'$

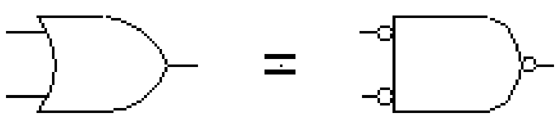
$\overline{a + b} = \overline{a}\,\overline{b}$ $\qquad$ $\overline{a\,b} = \overline{a} + \overline{b}$

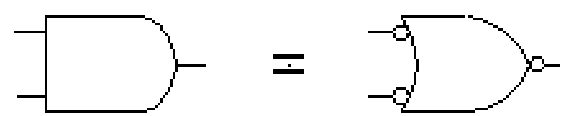




$a + b = (a'\,b')'$ $\qquad$ $(a\,b) = (a' + b')'$

$a + b = \overline{\overline{a}\;\overline{b}}$ $\qquad$ $a\,b = \overline{\overline{a} + \overline{b}}$

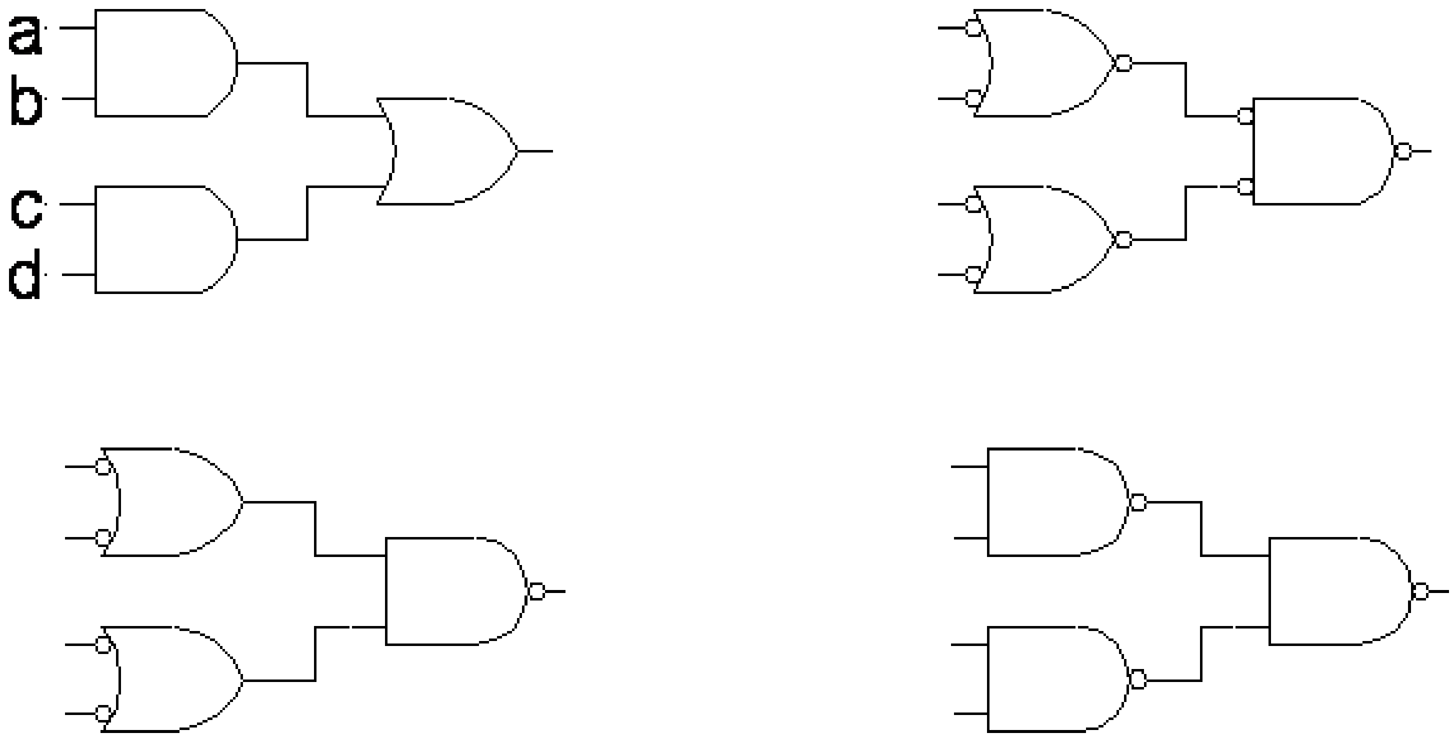




push bubbles **or** ***introduce in pairs*** **or** ***remove pairs***

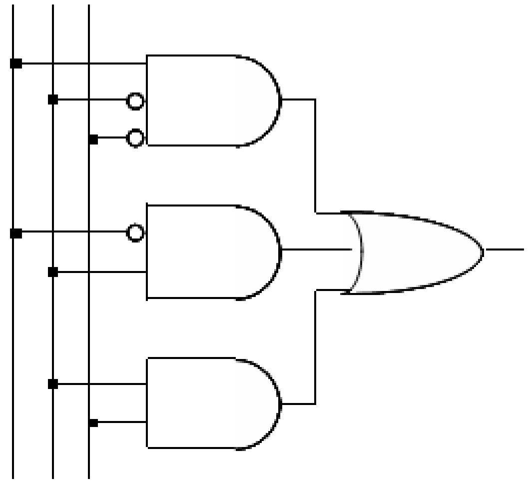
NAND-NAND Networks

Mapping from AND/OR to NAND/NAND

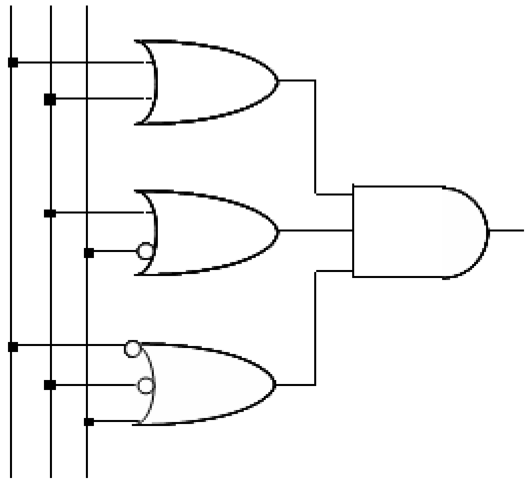


Implementations of 2-Level Logic

- **Sum-of-products**
  - **AND gates to form product terms (minterms)**
  - **OR gate to form sum**

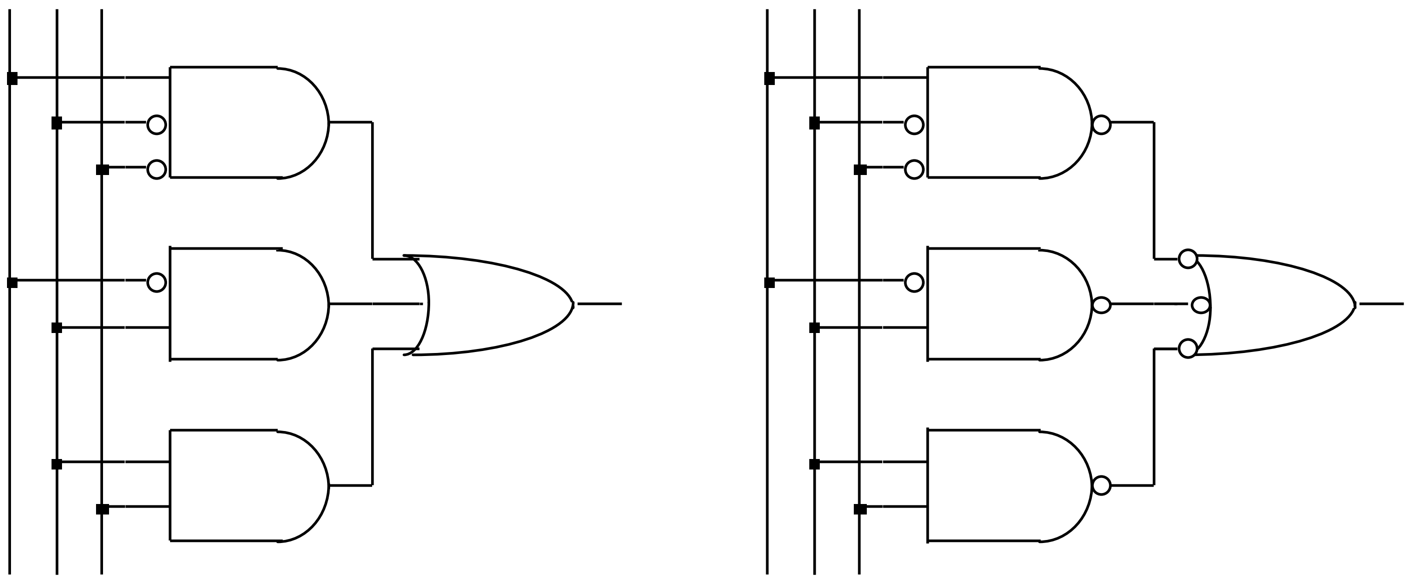


- **Product-of-sums**
  - **OR gates to form sum terms (maxterms)**
  - **AND gates to form product**



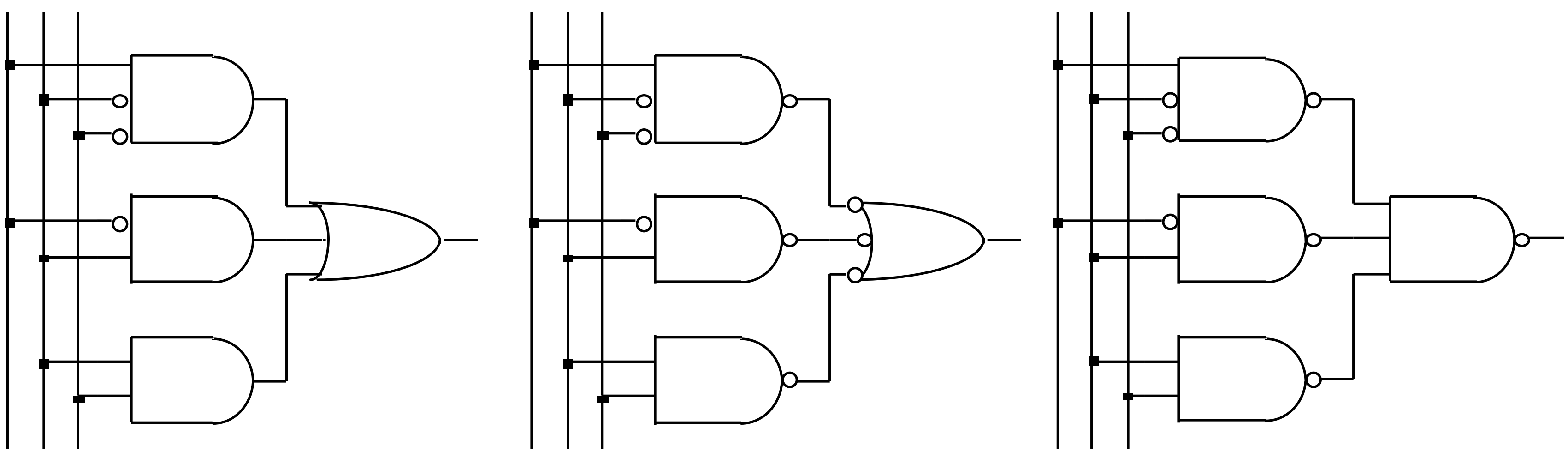
Two-level Logic using NAND Gates

- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate



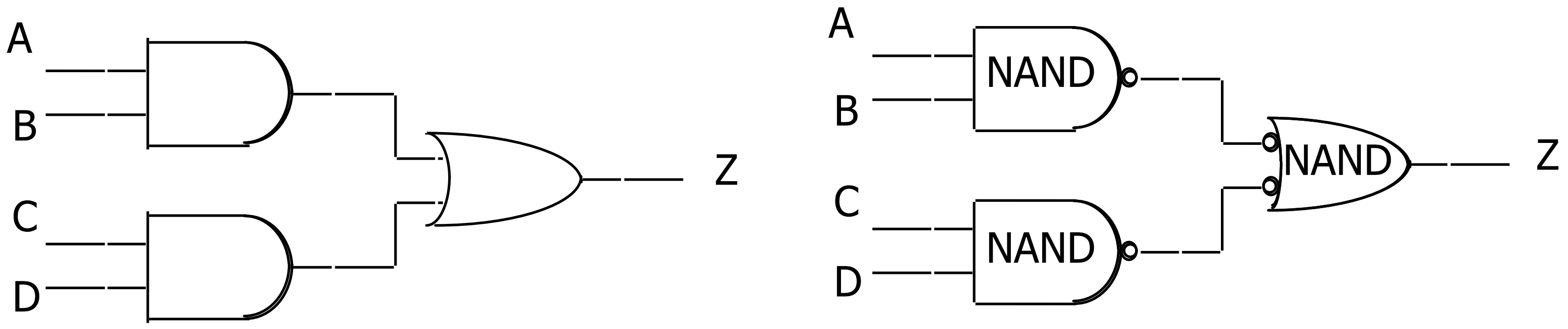
Two-level Logic using NAND Gates (cont'd)

- OR gate with inverted inputs is a NAND gate
  - DeMorgan's: A' + B' = (A • B)' $\overline{A} + \overline{B} = \overline{A \cdot B}$
- Two-level NAND-NAND network
  - Inverted inputs are not counted
  - In a typical circuit, inversion is done once and signal is then distributed



Conversion Between Forms (cont'd)

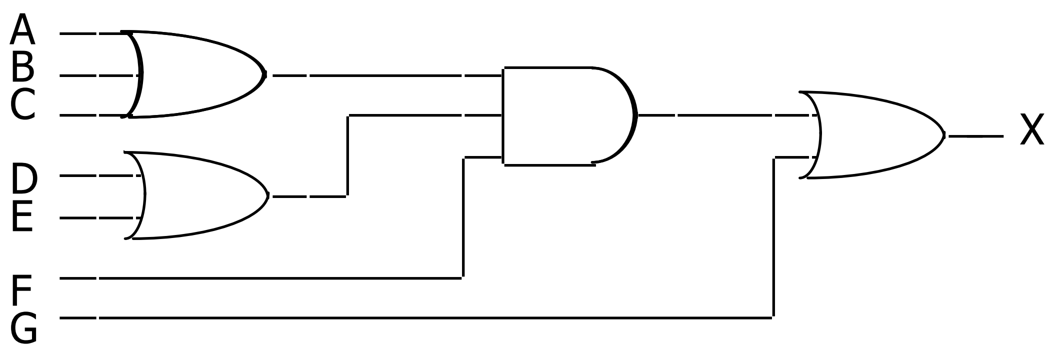
Example: verify equivalence of two forms



$$
\begin{aligned}
Z &= [\,(A \cdot B)' \cdot (C \cdot D)'\,]' \\
&= [\,(A' + B') \cdot (C' + D')\,]' \\
&= [\,(A' + B')' + (C' + D')'\,] \\
&= (A \cdot B) + (C \cdot D) \checkmark
\end{aligned}
$$

Multi-level Logic

- $x = (A + B + C)(D + E)F + G$
  - Factored form – not written as two-level S-o-P
  - 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
  - 10 wires (7 literals plus 3 internal wires)



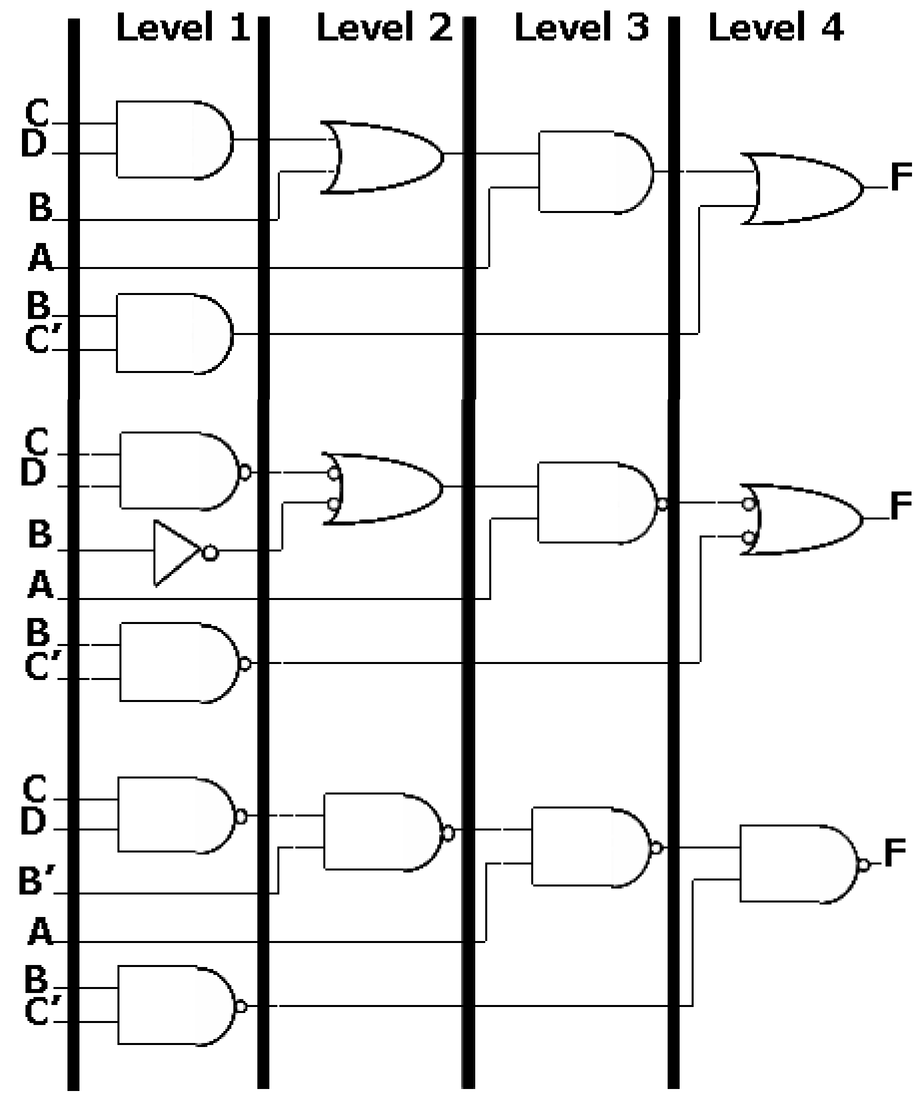
Conversion of Multi-level Logic to NAND Gates

$F = A\ (B + C\ D) + B\ C'$

original AND-OR network

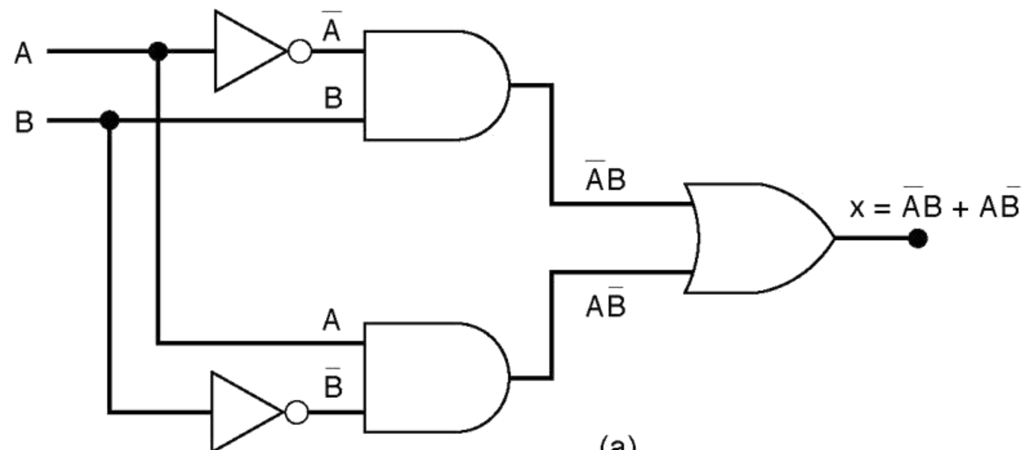
introduction and conservation of bubbles

redrawn in terms of conventional NAND gates



Exclusive-OR Circuits

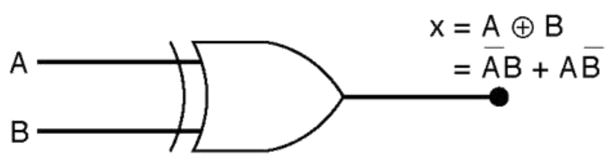
Exclusive-OR (XOR) produces a HIGH output whenever the two inputs are at opposite levels



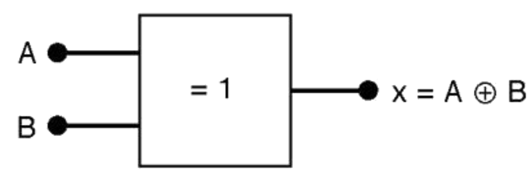
(a)

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate symbols



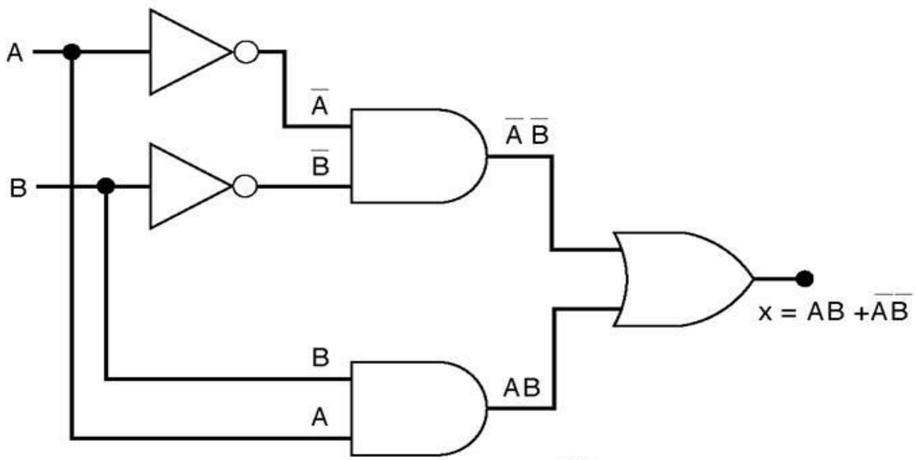
(b)



(c)

Exclusive-NOR Circuits

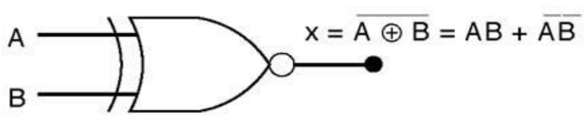
Exclusive-NOR (XNOR) produces a HIGH output whenever the two inputs are at the same level



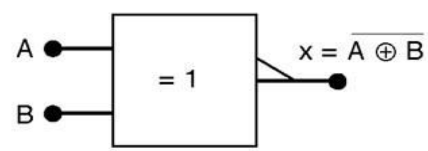
A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

(a)

XNOR gate symbols



(b)



(c)

XOR Function

XOR function can also be implemented with AND/OR gates (also NANDs)

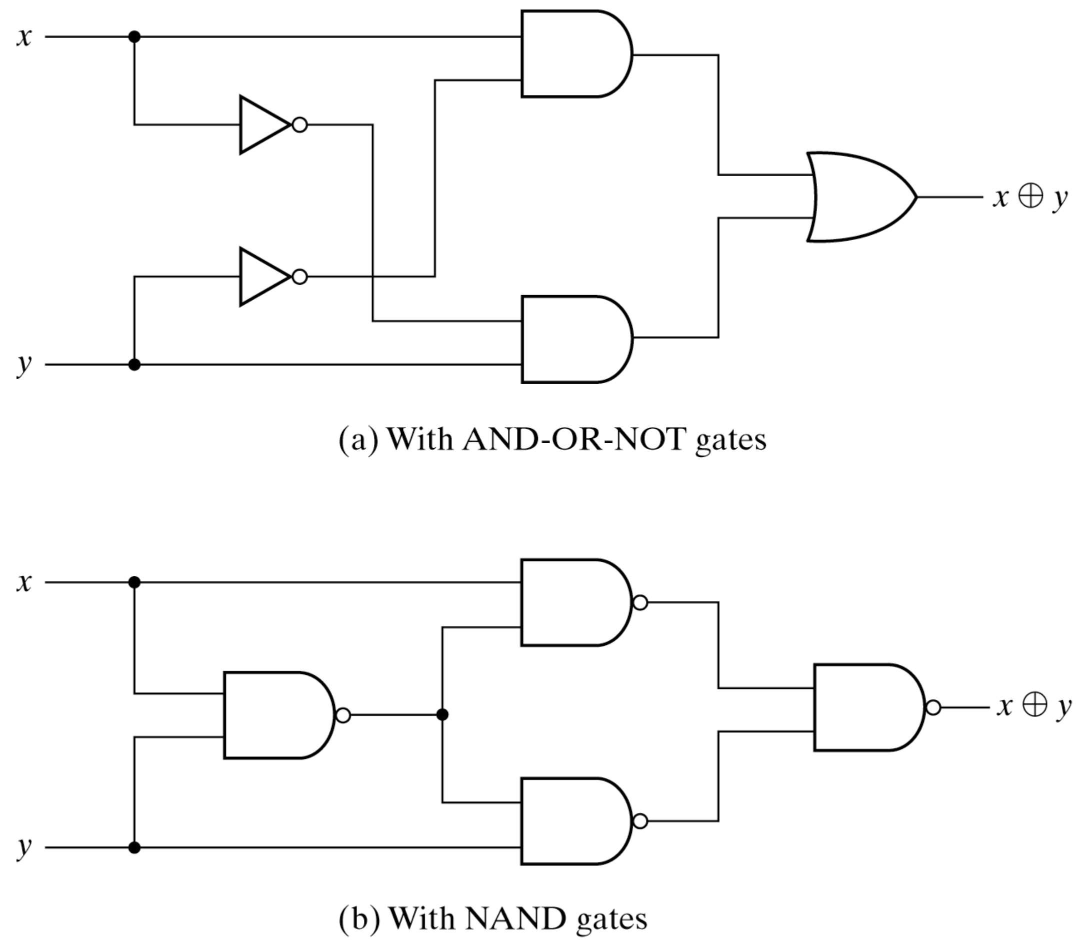
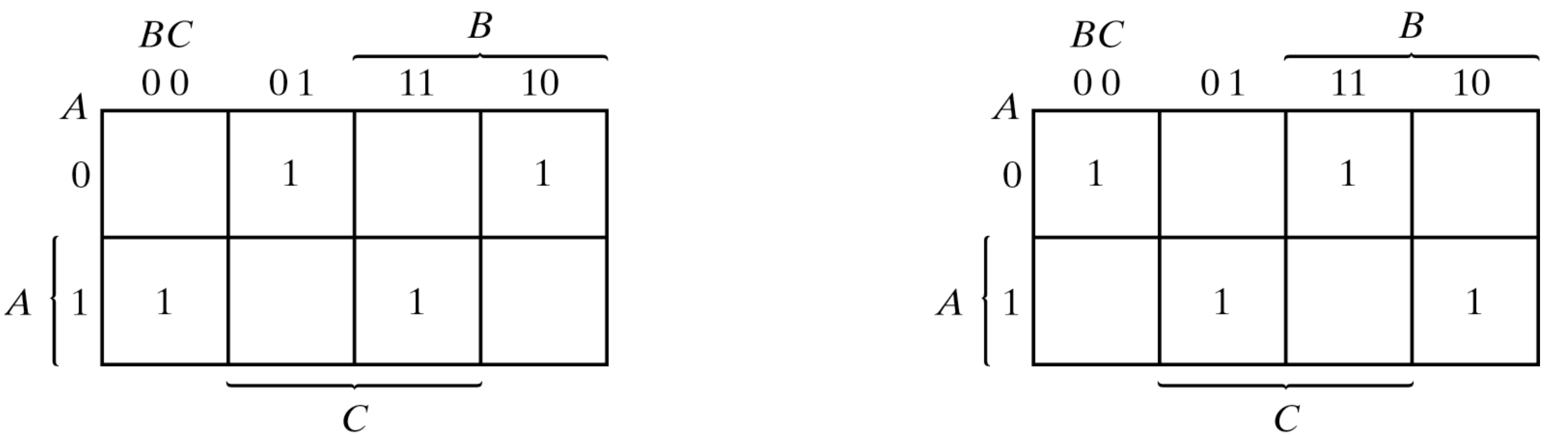


Fig. 3-32 Exclusive-OR Implementations

XOR Function

- **Even function – even number of inputs are 1**
- **Odd function – odd number of inputs are 1**

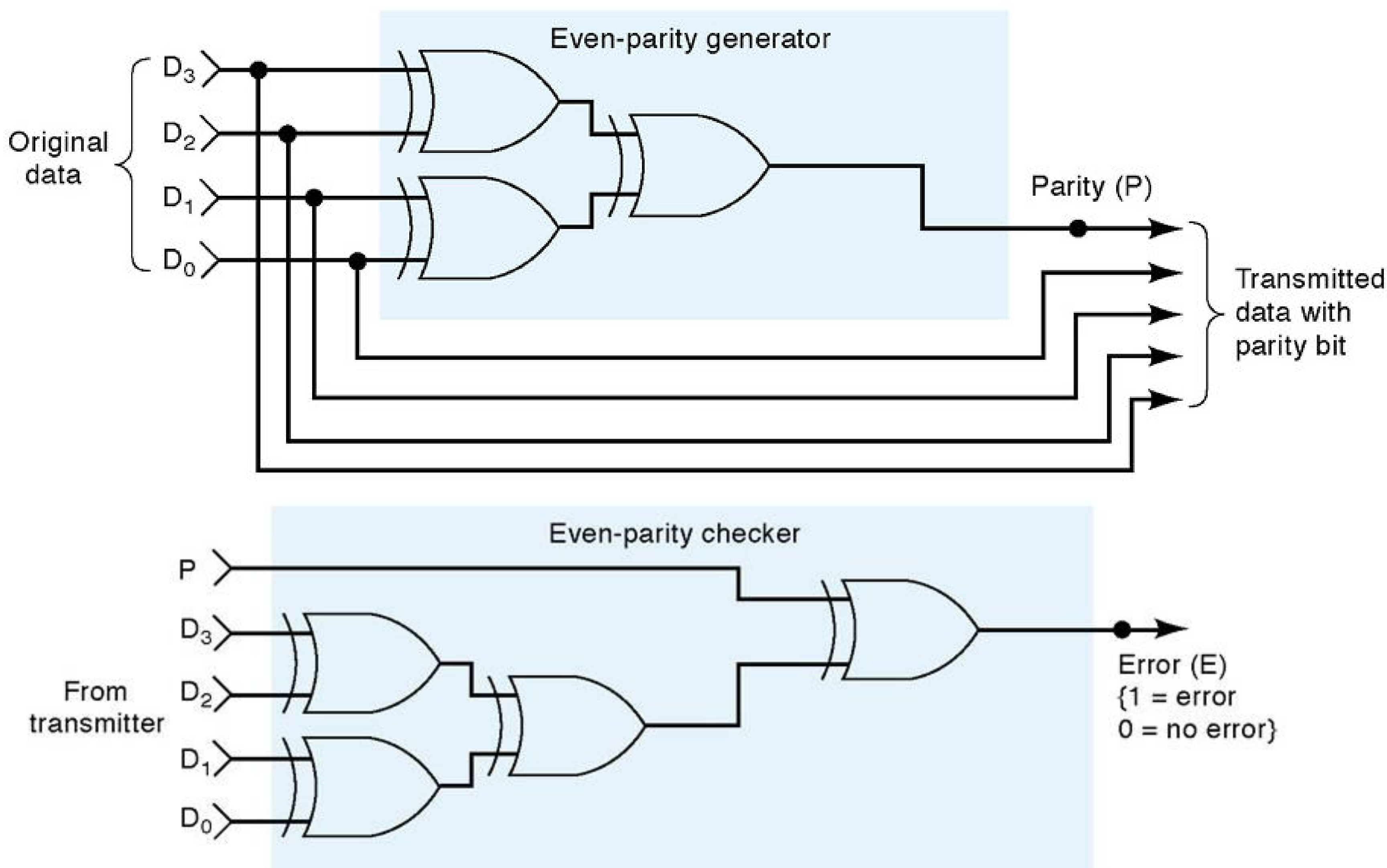


(a) Odd function
$F = A \oplus B \oplus C$

(a) Even function
$F = (A \oplus B \oplus C)'$

Fig. 3-33 Map for a Three-variable Exclusive-OR Function

Parity Generation and Checking

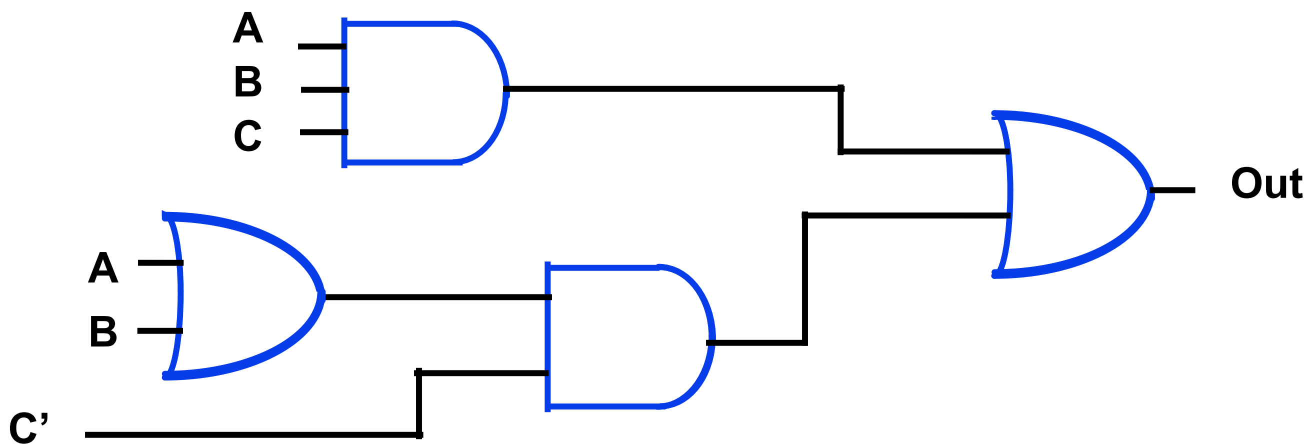


Summary

- Follow rules to convert between AND/OR representation and symbols
- Conversions are based on DeMorgan's Law
- NOR gate implementations are also possible
- XORs provide straightforward implementation for some functions
    - Used for parity generation and checking
- XOR circuits can also be implemented using AND/ORs

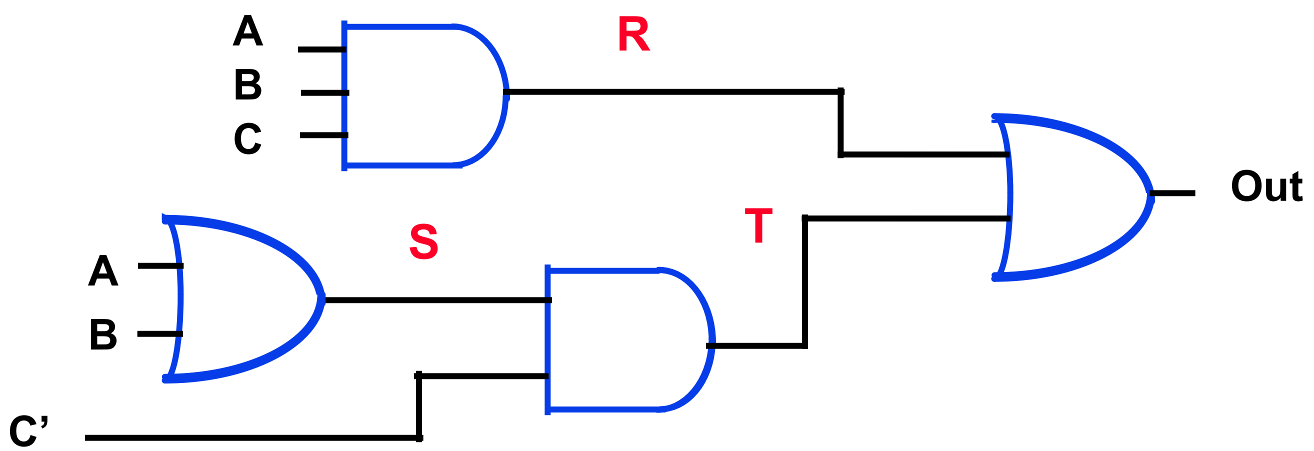
The Problem

- How can we convert from a circuit drawing to an equation or truth table?
- Two approaches
  - Create intermediate equations
  - Create intermediate truth tables



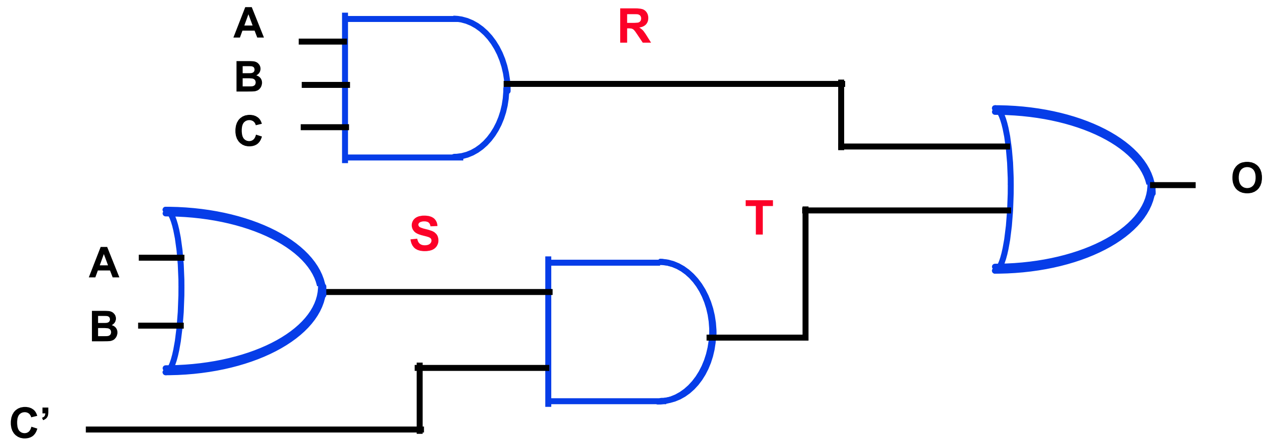
Label Gate Outputs

1. Label all gate outputs that are functions of input variables
2. Label gates that are functions of input variables and previously labeled gates
3. Repeat process until all outputs are labeled



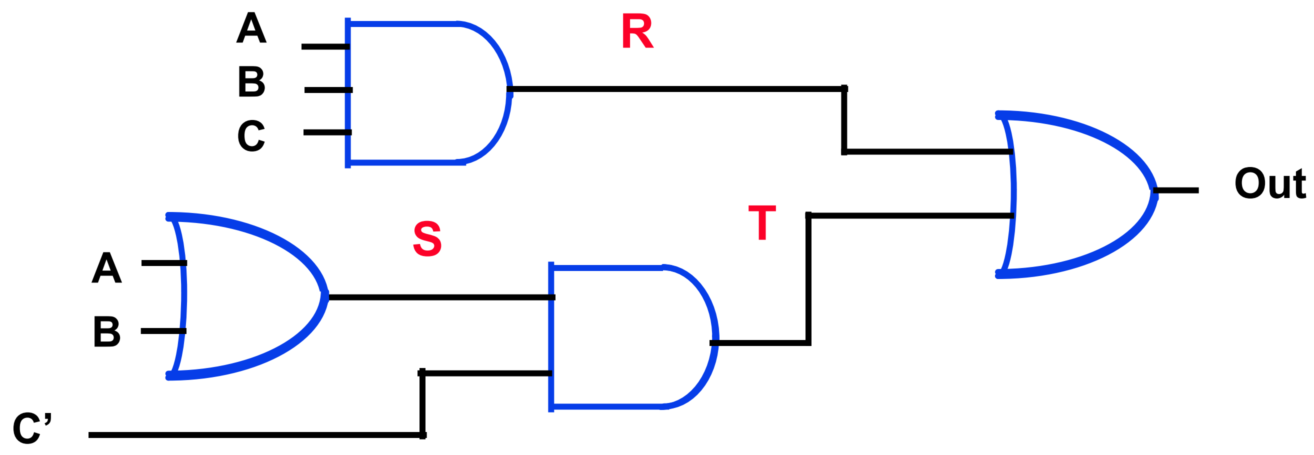
Approach 1: Create Intermediate Equations

- **Step 1**: Create an equation for each gate output based on its inputs
  - R = ABC
  - S = A + B
  - T = C'S
  - Out = R + T



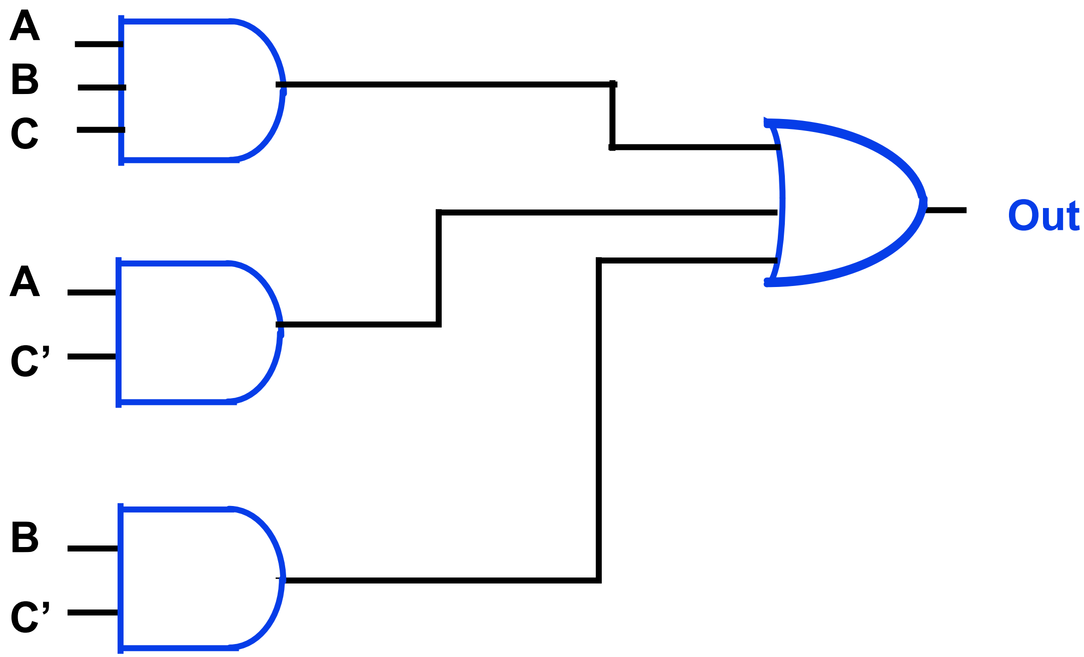
Approach 1: Substitute in subexpressions

- **Step 2**: Form a relationship based on input variables
  - R = ABC
  - S = A + B
  - T = C'S = C' (A + B)
  - Out = R+T = ABC + C'(A+B)



Approach 1: Substitute in subexpressions

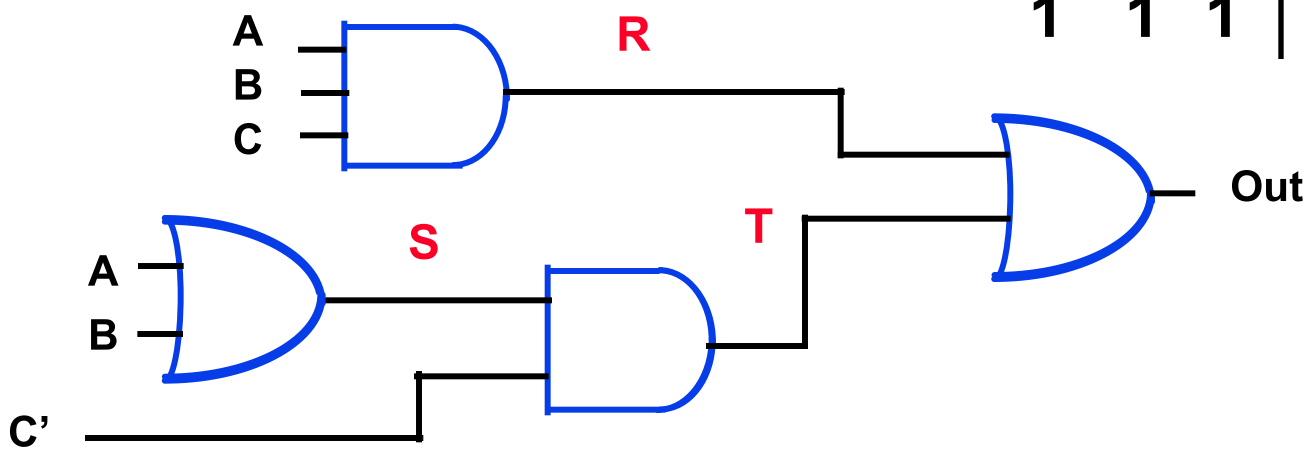
- **Step 3**: Expand equation to SOP
  - Out = ABC + C'(A+B) = ABC + AC' + BC'



Approach 2: Truth Table

- Step 1: Determine outputs for functions of input variables

A	B	C	R	S
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

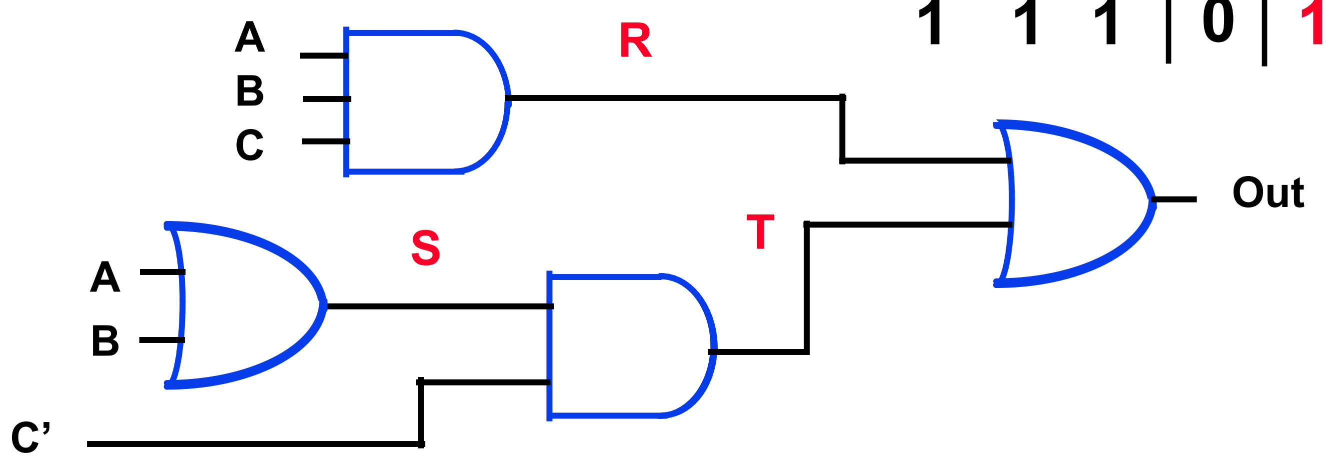


Approach 2: Truth Table

- **Step 2**: Determine outputs for functions of intermediate variables.

$$T = S \cdot C'$$

A	B	C	C'	R	S	T
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	0	1	0
1	1	0	1	0	1	1
1	1	1	0	1	1	0

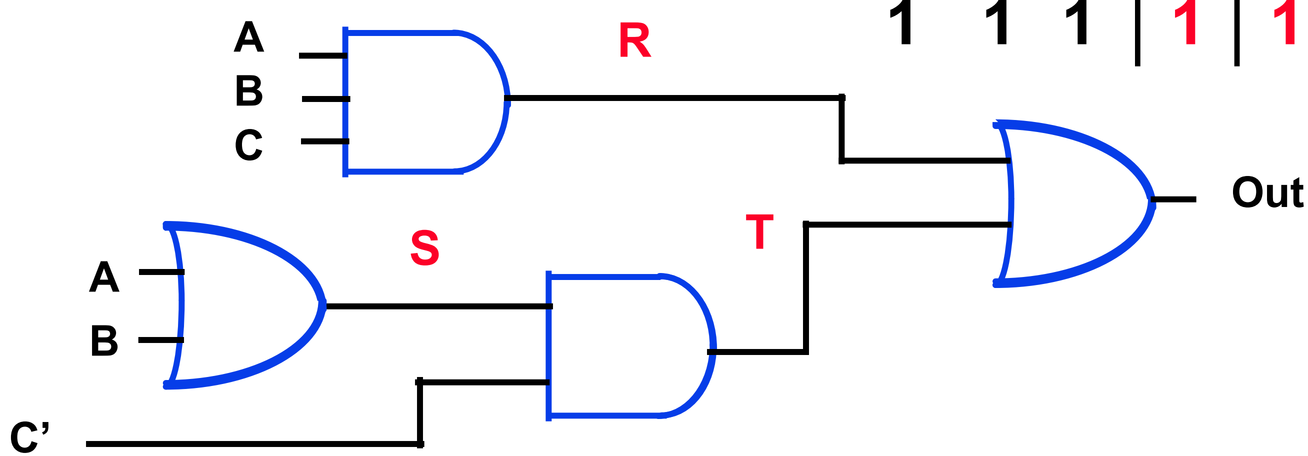


Approach 2: Truth Table

- **Step 3**: Determine outputs for function.

Out = R + T

A	B	C	R	S	T	Out
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	1	1	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	0	0
1	1	0	0	1	1	1
1	1	1	1	1	0	1



More Difficult Example

Note labels on interior nodes

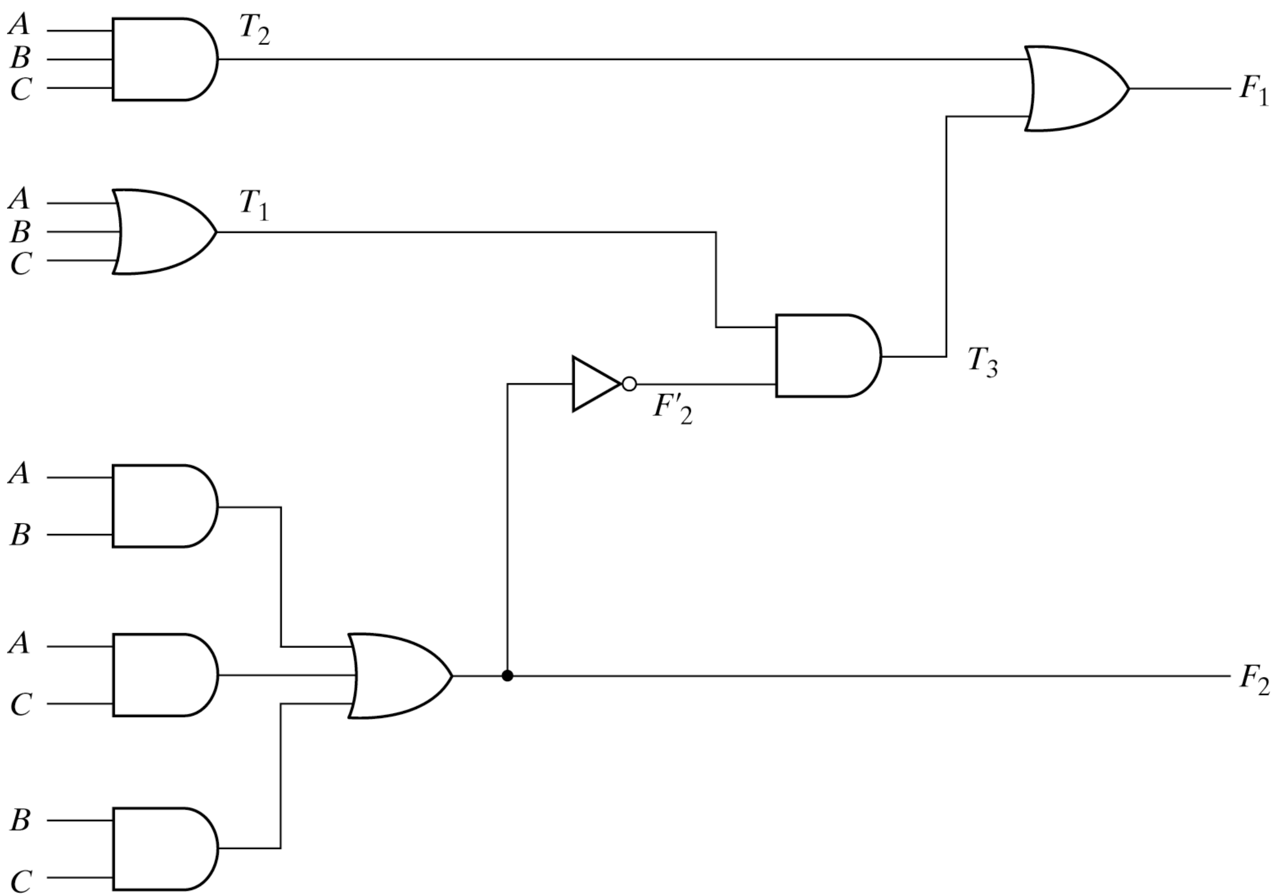


Fig. 4-2 Logic Diagram for Analysis Example

More Difficult Example: Truth Table

- Remember to determine intermediate variables starting from the inputs
- When all inputs are determined for a gate, determine its output
- The truth table can be reduced using K-maps

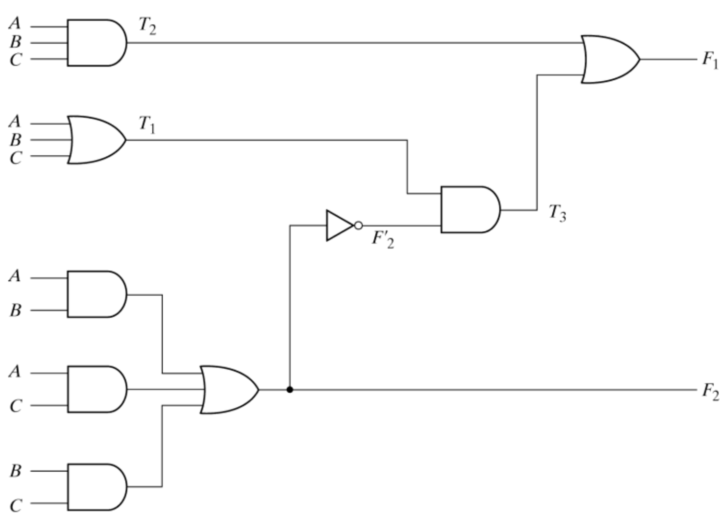


Fig. 4-2 Logic Diagram for Analysis Example

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Summary

- Important to be able to convert circuits into truth table and equation form
  - WHY? Leads to minimized sum of products representation
- Two approaches illustrated
  - Approach 1: Create an equation with circuit outputs dependent on circuit inputs
  - Approach 2: Create a truth table which shows relationship between circuit inputs and circuit outputs
- Both results can then be minimized using K-maps