LECTURE NOTES
ON
DIGITAL ELECTRONICS

GGS COLLEGE OF MODERN TECHNOLOGY

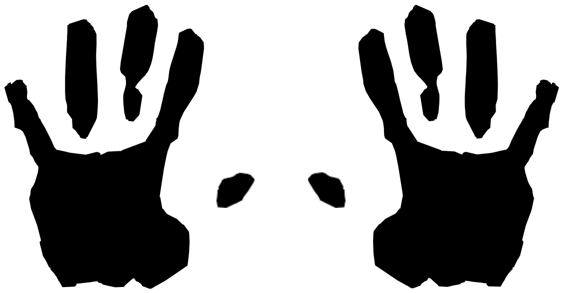
KHARAR

DEPT. OF ELECTRICAL ENGINEERING

MODULE - 1
Number Systems

Understanding Decimal Numbers

- Decimal numbers are made of decimal digits: (0,1,2,3,4,5,6,7,8,9)
- Decimal number representation:
  - $8653 = 8x10^3 + 6x10^2 + 5x10^1 + 3x10^0$
- What about fractions?
  - $97654.35 = 9x10^4 + 7x10^3 + 6x10^2 + 5x10^1 + 4x10^0 + 3x10^{-1} + 5x10^{-2}$
  - In formal notation -> $(97654.35)_{10}$
- Why do we use 10 digits?



Understanding Binary Numbers

- Binary numbers are made of binary digits (bits):
  - 0 and 1
- How many items does an binary number represent?
- $(1011)_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = (11)_{10}$
- 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*.
- $(1101)_2$

Why Use Binary Numbers?

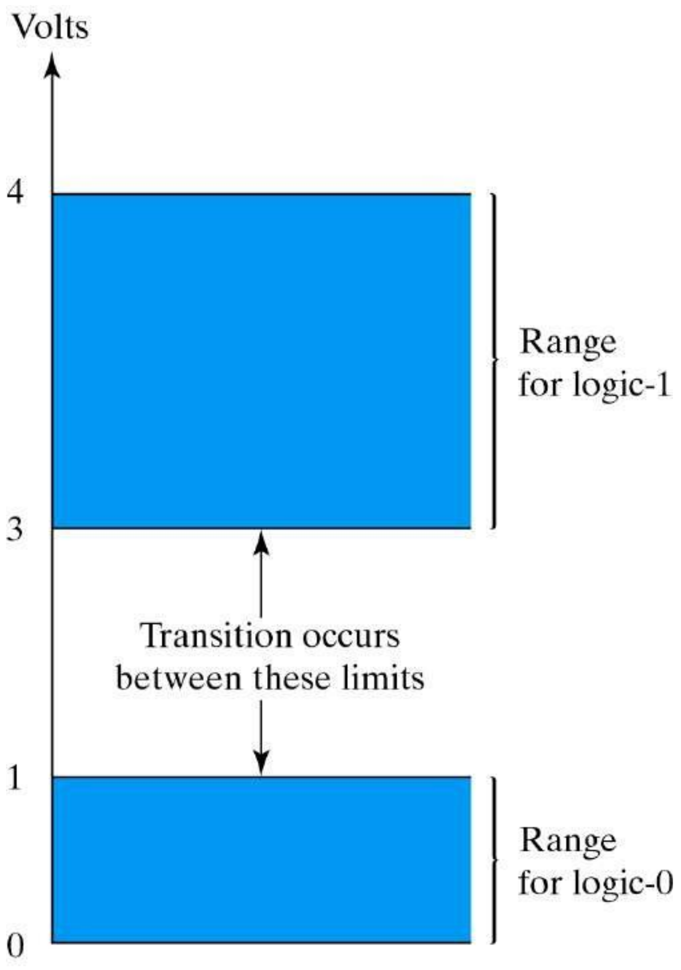
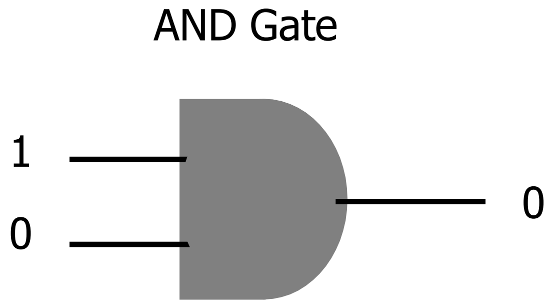
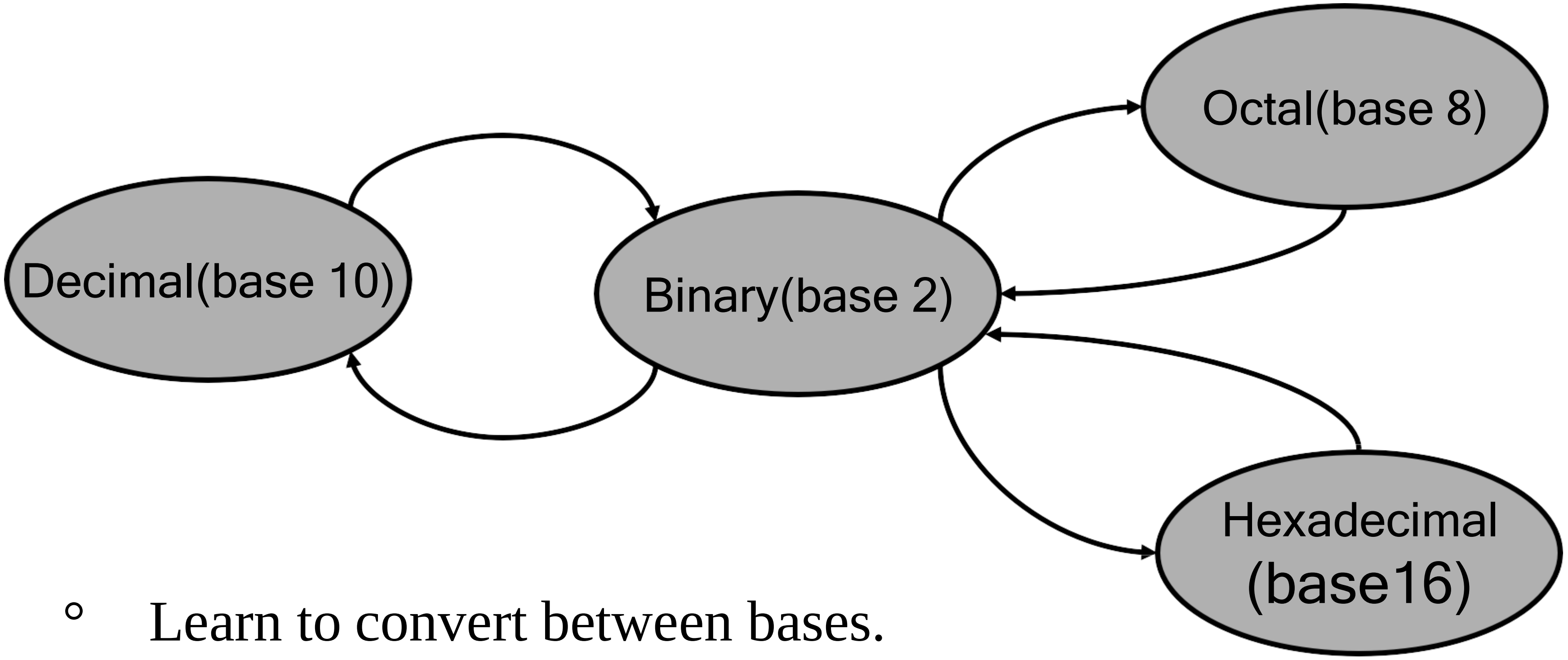


Fig. 1-3 Example of binary signals

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



Conversion Between Number Bases



- Learn to convert between bases.
- Conversion demonstrated in next slides

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 $(13)_{10}$:

	Integer Quotient		Remainder	Coefficient
13/2 =	6	+	½	$a_0 = 1$
6/2 =	3	+	0	$a_1 = 0$
3/2 =	1	+	½	$a_2 = 1$
1/2 =	0	+	½	$a_3 = 1$

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

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

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	$2^{10}=1024$	
11	$2^{11}=2048$	
12	$2^{12}=4096$	
20	$2^{20}=1M$	Mega
30	$2^{30}=1G$	Giga
40	$2^{40}=1T$	Tera

Binary Addition

- Binary addition is very simple.
- This is best shown in an example of adding two binary numbers…

```
    1   1   1   1   1   1 ←———————— carries

        1   1   1   1   0   1
+           1   0   1   1   1
-----------------------------
    1   0   1   0   1   0   0
```

Binary Subtraction

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

$$\begin{array}{cccccccc|l}
 & & 1 & & & 10 & & & \longleftarrow \text{borrows} \\
 & 0 & \cancel{10} & 10 & 0 & \cancel{0} & 10 & & \\
 & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{1} & \cancel{1} & \cancel{0} & 1 & \\
- & & & 1 & 0 & 1 & 1 & 1 & \\ \hline
 & & 1 & 1 & 0 & 1 & 1 & 0 &
\end{array}$$

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
```

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 $(175)_{10}$:

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

Answer $(175)_{10} = (a_2\, a_1\, a_0)_2 = (257)_8$

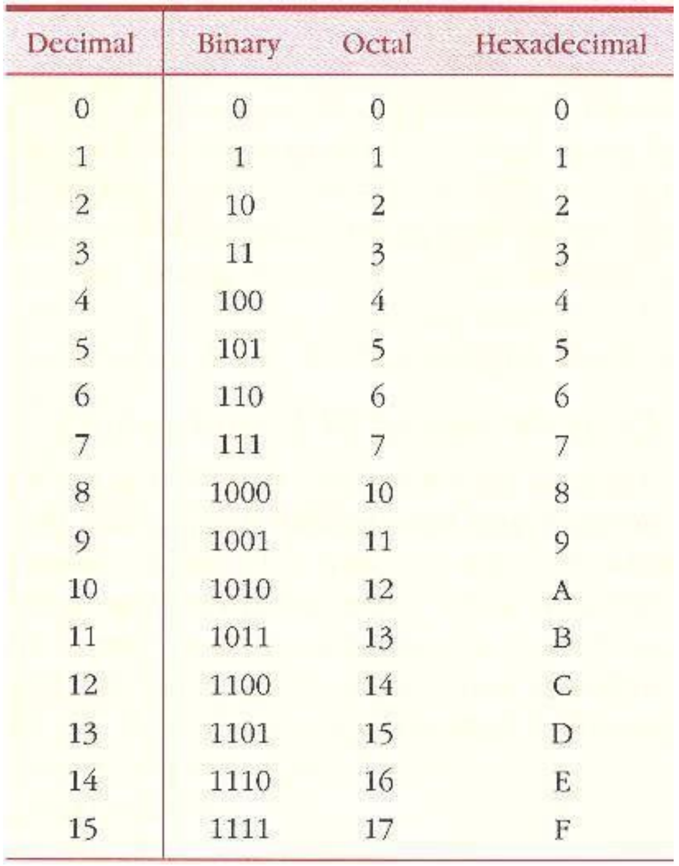
Understanding Octal Numbers

- Octal numbers are made of octal digits: (0,1,2,3,4,5,6,7)
- Octal number representation:
  - $(4536)_8 = 4x8^3 + 5x8^{2\,+}\,3x8^{1\,+}\,6x8^0 = (1362)_{10}$
- What about fractions?
  - $(465.27)_8 = 4x8^{2\,+}\,6x8^{1\,+}\,5x8^0 + 2x8^{-1\,+}\,7x8^{-2}$
- 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)
- hex number representation:
    - $(3A9F)_{16} = 3x16^3 + 10x16^2 + 9x16^1 + 15x16^0 = 14999_{10}$
- 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*.
    - $(1110)_2$

Putting It All Together



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

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

Converting Between Base 16 and Base 2

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

- Determine 4-bit value for each hex digit
- Note that there are $2^4 = 16$ different values of four bits
- Easier to read and write in hexadecimal.
- Representations are equivalent!

Converting Between Base 16 and Base 8

$$\boxed{3A9F_{16} = \underbrace{0011}_{3}\ \underbrace{1010}_{A}\ \underbrace{1001}_{9}\ \underbrace{1111}_{F}{}_{2}}$$

$$\downarrow$$

$$35237_{8} = \underbrace{011}_{3}\ \underbrace{101}_{5}\ \underbrace{010}_{2}\ \underbrace{011}_{3}\ \underbrace{111}_{7}{}_{2}$$

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.

How To Represent Signed Numbers

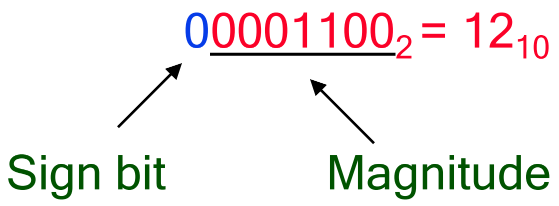
- Plus and minus sign used for decimal numbers: 25 (or +25), -16, etc.
- For computers, desirable to represent everything as *bits*.
- Three types of signed binary number representations: signed magnitude, 1’s complement, 2’s complement.
- In each case: left-most bit indicates sign: positive (0) or negative (1).

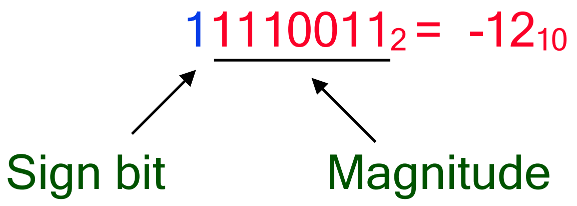
Consider signed magnitude:

$$\underbrace{0}_{\text{Sign bit}}\underbrace{0001100}_{\text{Magnitude}}{}_2 = 12_{10} \qquad \underbrace{1}_{\text{Sign bit}}\underbrace{0001100}_{\text{Magnitude}}{}_2 = -12_{10}$$

One’s Complement Representation

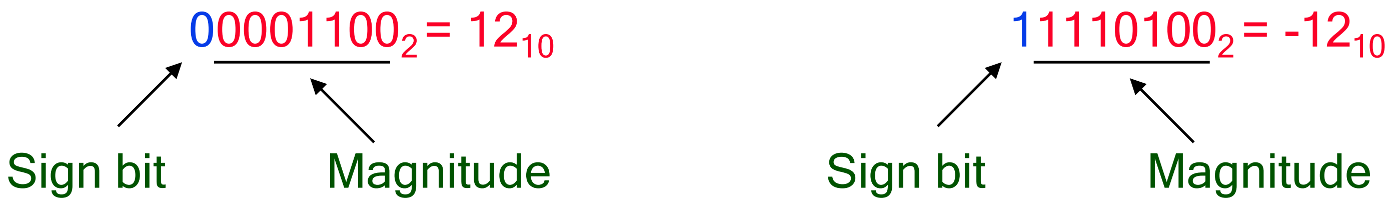
- The one’s complement of a binary number involves inverting all bits.
    - 1’s comp of 00110011 is 11001100
    - 1’s comp of 10101010 is 01010101
- For an n bit number N the 1’s complement is $(2^n-1) - N$.
- Called diminished radix complement by Mano since 1’s complement for base (radix 2).
- To find negative of 1’s complement number take the 1’s complement.





Two’s Complement Representation

- The two’s complement of a binary number involves inverting all bits and 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 since 2’s complement for base (radix 2).
- To find negative of 2’s complement number take the 2’s complement.



Two's Complement Shortcuts

- Algorithm 1 – Simply complement each bit and then add 1 to the result.
  - Finding the 2's complement of $(01100101)_2$ and of its 2's complement…

```
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 and then complementing the remaining bits.
  - N = 0 1 1 0 0 1 0 1

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

1’s Complement Addition

- Using 1’s complement numbers, adding numbers is easy.
- For example, suppose we wish to add $+(1100)_2$ and $+(0001)_2$.
- Let’s compute $(12)_{10} + (1)_{10}$.
  - $(12)_{10} = +(1100)_2 = 01100_2$ in 1’s comp.
  - $(1)_{10} = +(0001)_2 = 00001_2$ in 1’s comp.

Step 1: Add binary numbers
Step 2: Add carry to low-order bit

```
                          0  1  1  0  0
Add         +             0  0  0  0  1
            -------------------------------
                   0      0  1  1  0  1
Add carry          └───────────────────> 0
            -------------------------------
Final
Result                    0  1  1  0  1
```

1’s Complement Subtraction

- Using 1’s complement numbers, subtracting numbers is also easy.
- For example, suppose we wish to subtract $+(0001)_2$ from $+(1100)_2$.
- Let’s compute $(12)_{10} - (1)_{10}$.
  - $(12)_{10} = +(1100)_2 = 01100_2$ in 1’s comp.
  - $(-1)_{10} = -(0001)_2 = 11110_2$ in 1’s comp.

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

1’s comp
                      0   1   1   0   0
Add         +         1   1   1   1   0
            ----------------------------
                  1   0   1   0   1   0
Add carry         └───────────────────→ 1
            ----------------------------
Final
Result                0   1   0   1   1
```

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

Step 2: Add binary numbers

Step 3: Add carry to low order bit

2’s Complement Addition

- Using 2’s complement numbers, adding numbers is easy.
- For example, suppose we wish to add $+(1100)_2$ and $+(0001)_2$.
- Let’s compute $(12)_{10} + (1)_{10}$.
  - $(12)_{10} = +(1100)_2 = 01100_2$ in 2’s comp.
  - $(1)_{10} = +(0001)_2 = 00001_2$ in 2’s comp.

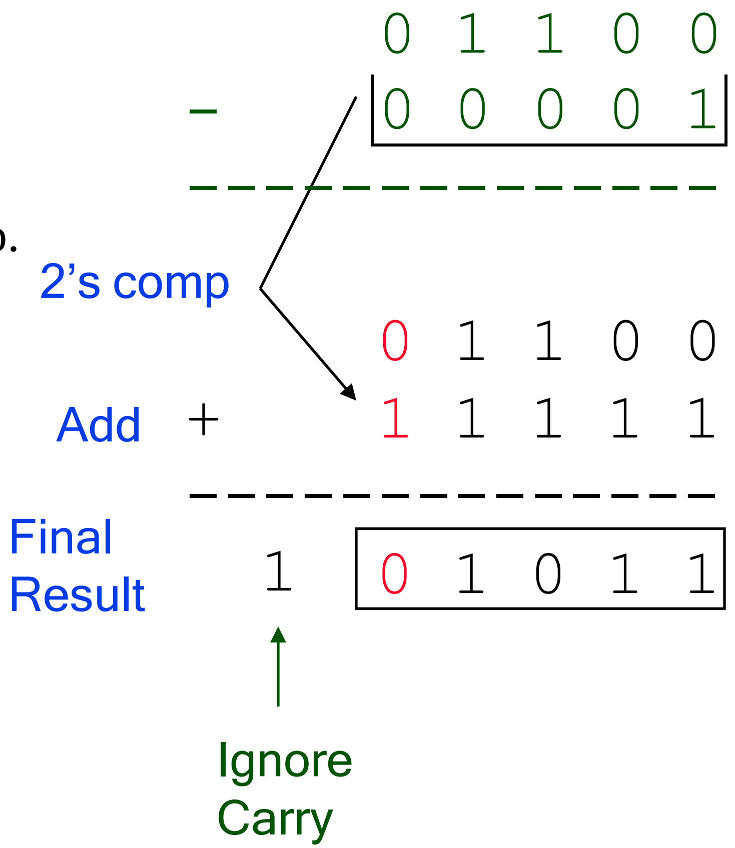
Step 1: Add binary numbers
Step 2: Ignore carry bit

$$\begin{array}{llcccccc}
 & & & 0 & 1 & 1 & 0 & 0 \\
\text{Add} & + & & 0 & 0 & 0 & 0 & 1 \\
\hline
\text{Final Result} & & 0 & \multicolumn{5}{c}{\boxed{0\quad 1\quad 1\quad 0\quad 1}} \\
 & & \uparrow & & & & & \\
 & & \text{Ignore} & & & & &
\end{array}$$

2's Complement Subtraction

- Using 2's complement numbers, follow steps for subtraction
- For example, suppose we wish to subtract $+(0001)_2$ from $+(1100)_2$.
- Let's compute $(12)_{10} - (1)_{10}$.
  - $(12)_{10} = +(1100)_2 = 01100_2$ in 2's comp.
  - $(-1)_{10} = -(0001)_2 = 11111_2$ in 2's comp.

Step 1: Take 2's complement of 2nd operand
Step 2: Add binary numbers
Step 3: Ignore carry bit



2’s Complement Subtraction: Example #2

- Let’s compute $(13)_{10} - (5)_{10}$.
  - $(13)_{10} = +(1101)_2 = (01101)_2$
  - $(-5)_{10} = -(0101)_2 = (11011)_2$
- Adding these two 5-bit codes…

```
                        0 1 1 0 1
carry              +    1 1 0 1 1
                   ------------------
                     1  0 1 0 0 0
```

- Discarding the carry bit, the sign bit is seen to be zero, indicating a correct result. Indeed,

  $(01000)_2 = +(1000)_2 = +(8)_{10}$.

2’s Complement Subtraction: Example #3

- Let’s compute $(5)_{10} – (12)_{10}$.
  - $(-12)_{10}$ = $-(1100)_2$ = $(10100)_2$
  - $(5)_{10}$ = $+(0101)_2$ = $(00101)_2$
- Adding these two 5-bit codes…

```
           0 0 1 0 1
 +         1 0 1 0 0
 ---------------------
           1 1 0 0 1
```

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

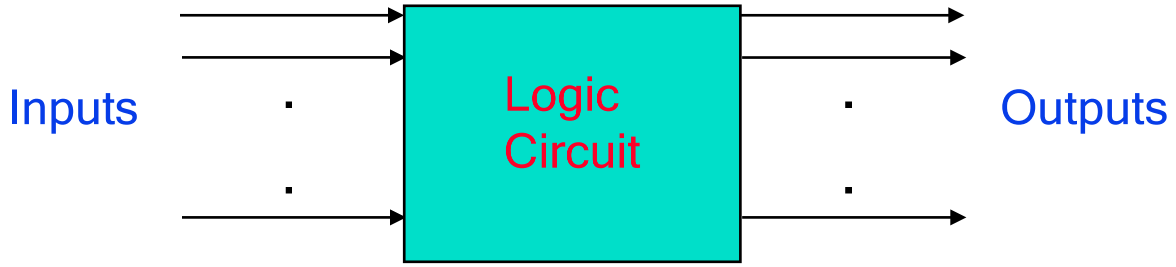
Boolean Algebra

Overview

- Logic functions with 1’s and 0’s
    - Building digital circuitry
- Truth tables
- Logic symbols and waveforms
- Boolean algebra
- Properties of Boolean Algebra
    - Reducing functions
    - Transforming functions

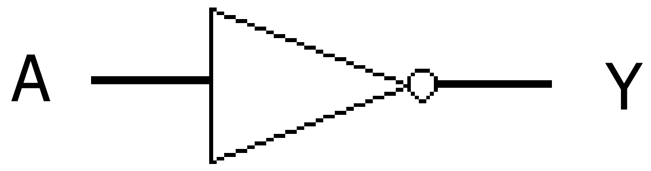
Digital Systems

- Analysis problem:



  - Determine binary outputs for each combination of inputs
- Design problem: given a task, develop a circuit that accomplishes the task
  - Many possible implementation
  - Try to develop "best" circuit based on some criterion (size, power, performance, etc.)

Describing Circuit Functionality: Inverter



Symbol

Truth Table

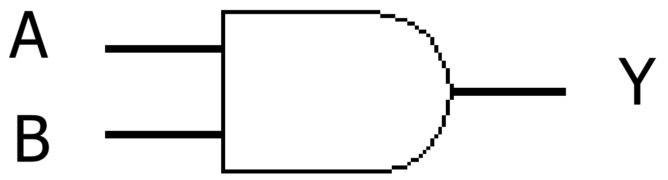
A	Y
0	1
1	0

Input

Output

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

The AND Gate

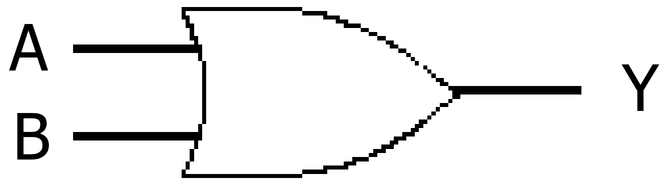


- This is an AND gate.
- So, if the two inputs signals are asserted (high) the output will also be asserted. Otherwise, the output will be deasserted (low).

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.
- So, if either of the two input signals are 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

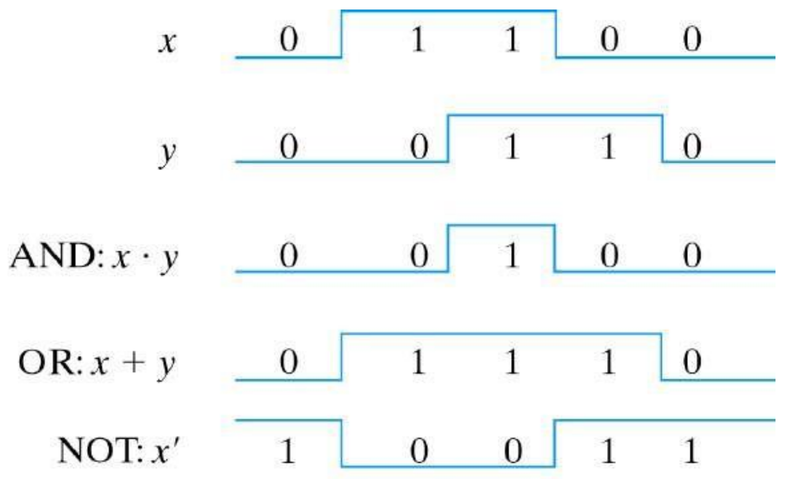


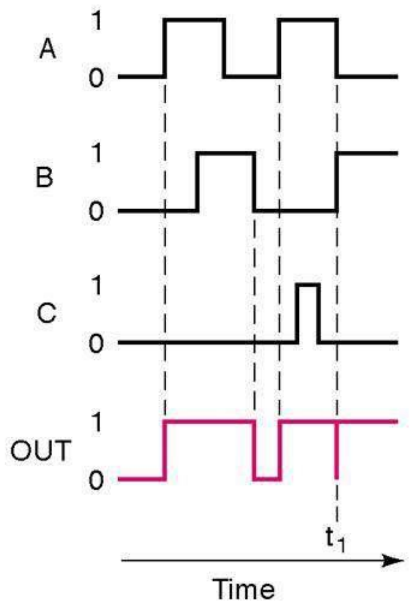
Fig. 1-5 Input-output signals for gates

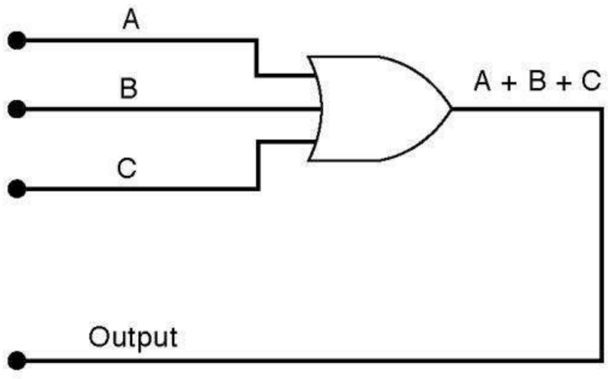
AND Gate

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

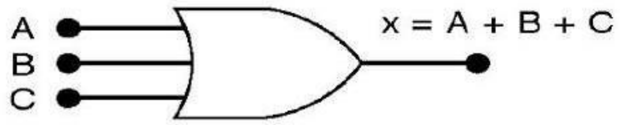
- Waveforms provide another approach for representing functionality.
- Values are either high (logic 1) or low (logic 0).
- Can you create a truth table from the waveforms?

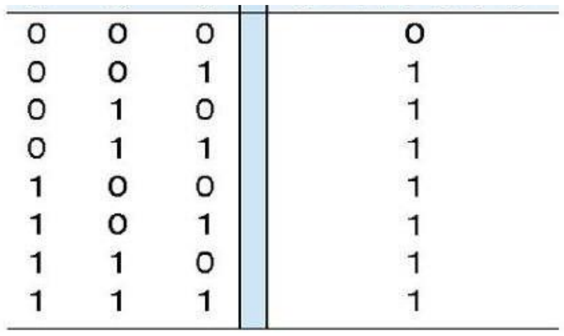
Consider three-input gates





3 Input OR Gate

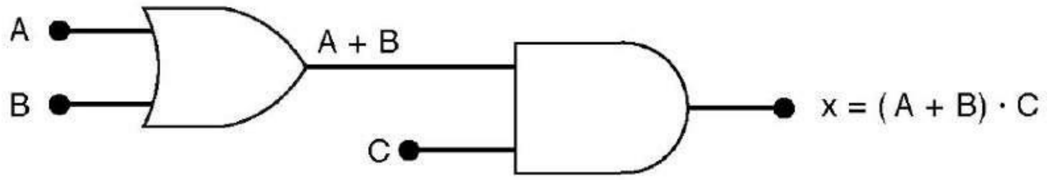




0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

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 or two or more elements and the two operators, . and +.
- Useful for identifying and minimizing circuit functionality
- Identity elements
  - a + 0 = a
  - a . 1 = a
- 0 is the identity element for the + operation.
- 1 is the identity element for the . operation.

Commutativity and Associativity of the Operators

- The Commutative Property:

  For every a and b in K,

  - a + b = b + a
  - a . b = b . a

- The Associative Property:

  For every a, b, and c in K,

  - a + (b + c) = (a + b) + c
  - a . (b . c) = (a . b) . c

Distributivity of the Operators and Complements

- The Distributive Property:

  For every a, b, and c in K,

  - a + (b . c) = (a + b) . (a + c)
  - a . (b + c) = (a . b) + (a . c)

- The Existence of the Complement:

  For every a in K there exists a unique element called a' (*complement of a)* such that,

  - a + a' = 1
  - a . 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 . c = (a + b) . (a + c)
  - a + bc = (a + b)(a + c)

Duality

- The principle of duality is an important concept. This says that 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 expression

  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
- Remember that aa’ = 0 and a+a’=1.
  - Therefore, a’ is the complement of a and a is also the complement of a’.
  - As the complement of a’ is unique, it follows that a’’=a.
- Taking the double inverse of a value will give the initial value.

Absorption

- This theorem states:

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

- To prove the first half of this theorem:

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

DeMorgan's Theorem

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

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

- Complement the expression

  $a(b + z(x + a'))$ and simplify.

$$
\begin{aligned}
(a(b+z(x + a')))' \quad &= a' + (b + z(x + a'))' \\
&= a' + b'(z(x + a'))' \\
&= a' + b'(z' + (x + a')') \\
&= a' + b'(z' + x'a'') \\
&= a' + b'(z' + x'a)
\end{aligned}
$$

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.

More Logic Functions: NAND, NOR, XOR

Overview

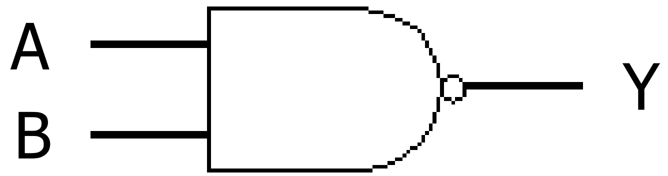
- More 2-input logic gates (NAND, NOR, XOR)
- Extensions to 3-input gates
- Converting between sum-of-products and NANDs
  - SOP to NANDs
  - NANDs to SOP
- Converting between sum-of-products and NORs
  - SOP to NORs
  - NORs to SOP
- Positive and negative logic
  - We use primarily positive logic in this course.

Logic functions of N variables

- Each truth table represents one possible function (e.g. AND, OR)
- If there are N inputs, there are 2^{2^N}
- For example, is 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



- This is a NAND gate. It is a combination of an AND gate followed by an inverter. Its truth table shows this…
- 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)

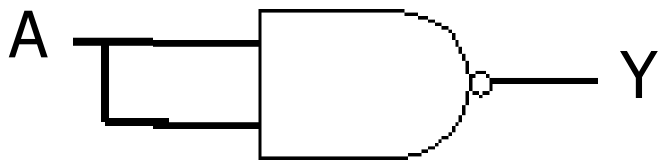
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

The NAND Gate

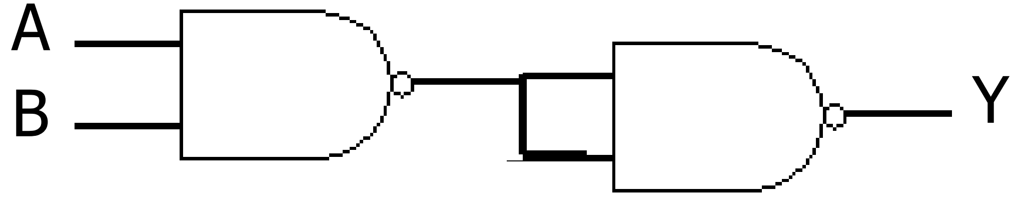
- These three properties show that a NAND gate with both of its inputs driven by the same signal is equivalent to a NOT gate
- A NAND gate whose output is complemented is equivalent to an AND gate, and a NAND gate with complemented inputs acts as an OR gate.
- Therefore, we can use a NAND gate to implement 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*.

NAND Gates into Other Gates

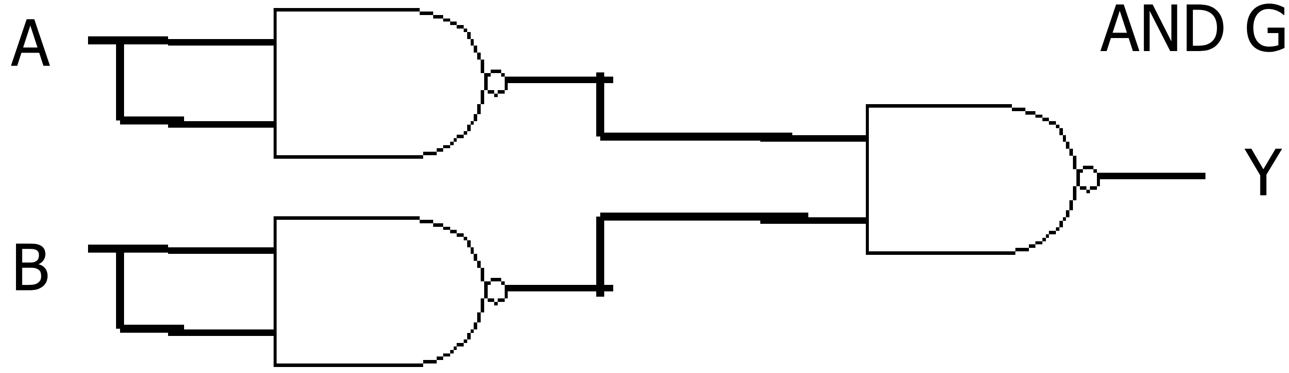
(what are these circuits?)



NOT Gate

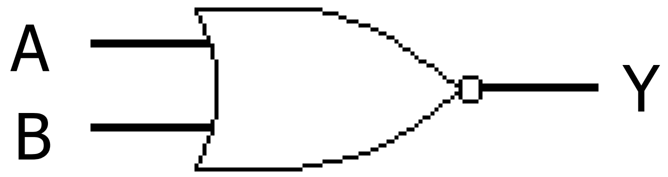


AND Gate



OR Gate

The NOR Gate



- This is a NOR gate. It is a combination of an OR gate followed by an inverter. It's truth table shows this…
- 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)

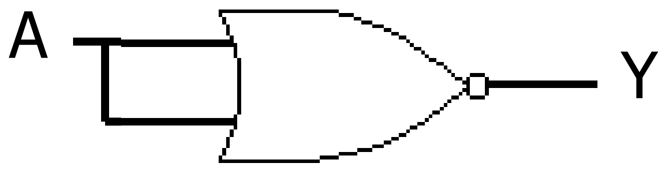
A	B	Y
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**0**

Functionally Complete Gates

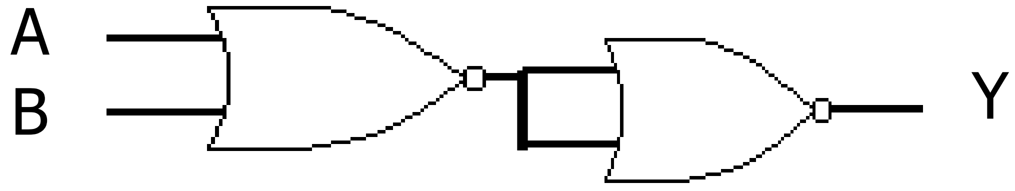
- Just like the NAND gate, the NOR gate is functionally complete…any logic function can be implemented using just NOR gates.
- Both NAND and NOR gates are very valuable as any design can be realized using either one.
- It is easier to build an IC chip using all NAND or NOR gates than to combine AND,OR,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

NOR Gates into Other Gates

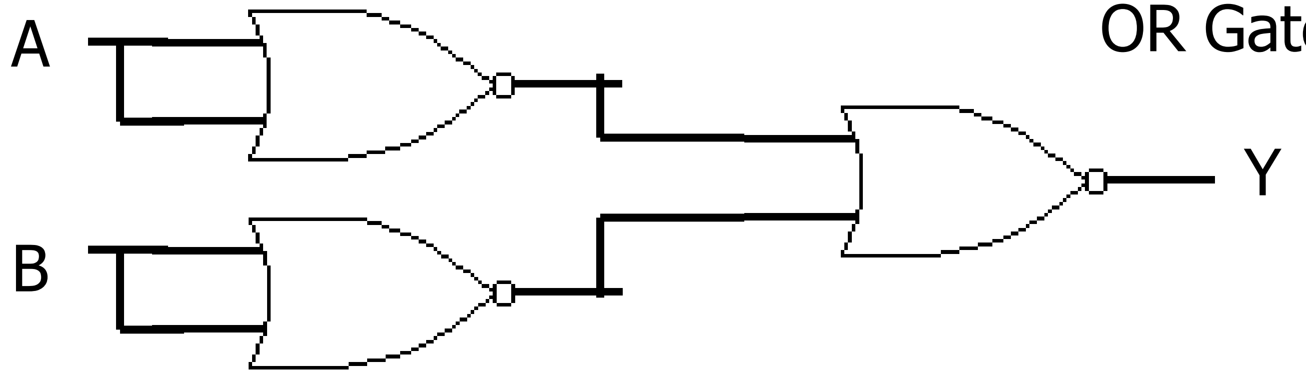
(what are these circuits?)



NOT Gate

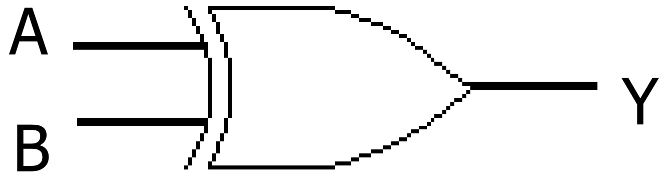


OR Gate



AND Gate

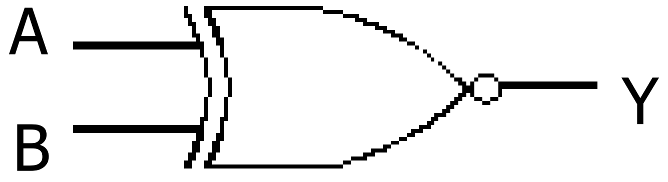
The XOR Gate (Exclusive-OR)



- 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 ⊕, i.e. $1 \oplus 1 = 0$ and $1 \oplus 0 = 1$.

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

The XNOR Gate

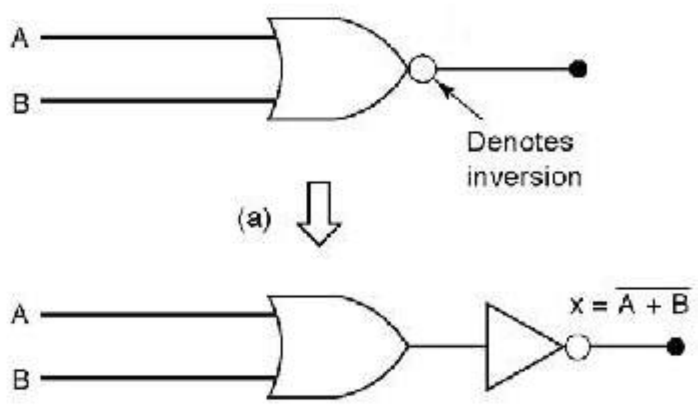


- This is a XNOR gate.
- This functions as an exclusive-NOR gate, or simply the complement of the XOR gate.
- The switching algebra symbol for this operation is ⊙, i.e. 1 ⊙ 1 = 1 and 1 ⊙ 0 = 0.

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

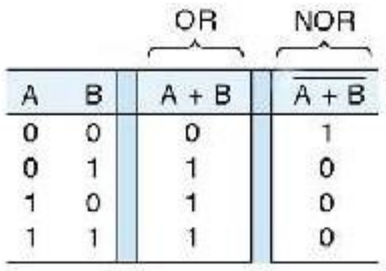
NOR Gate Equivalence

- 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:

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

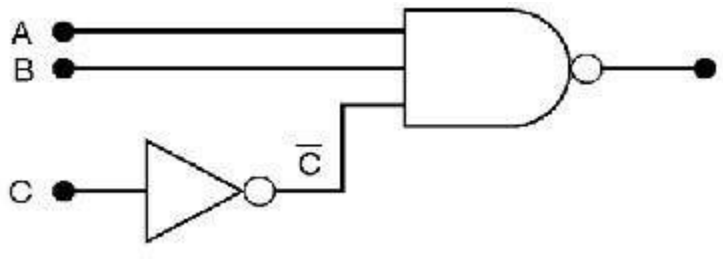
- Complement the expression

  $a(b + z(x + a'))$ and simplify.

$$
\begin{aligned}
(a(b+z(x + a')))' &= a' + (b + z(x + a'))' \\
&= a' + b'(z(x + a'))' \\
&= a' + b'(z' + (x + a')') \\
&= a' + b'(z' + x'a'') \\
&= a' + b'(z' + x'a)
\end{aligned}
$$

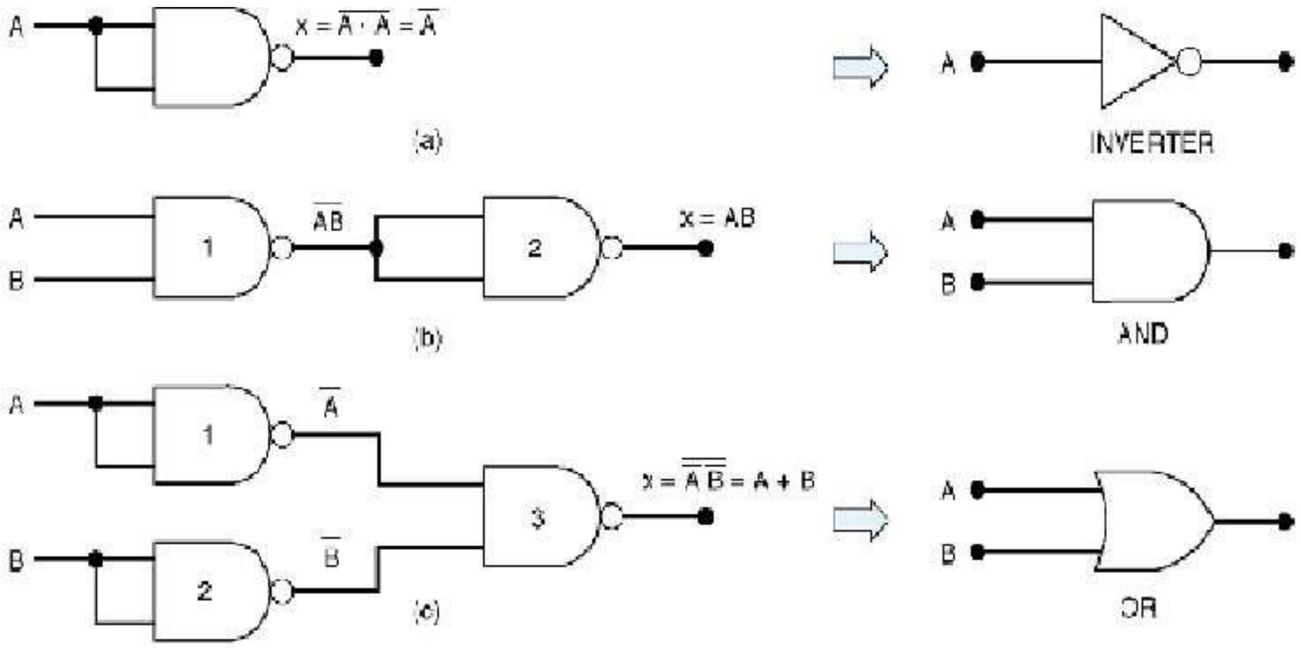
Example

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



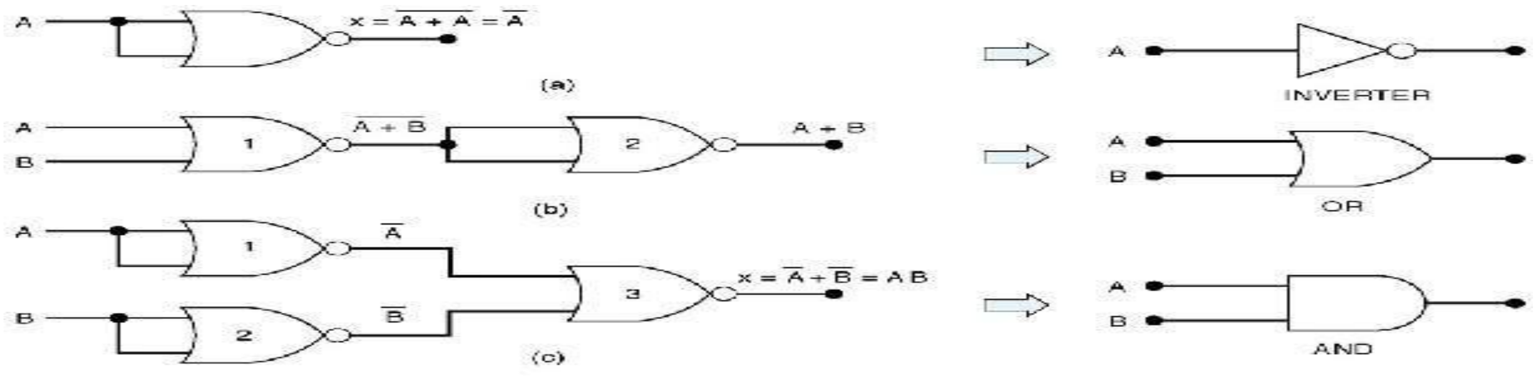
$$z = \overline{A \cdot B \cdot \overline{C}} = \overline{A} + \overline{B} + \overline{\overline{C}} = \overline{A} + \overline{B} + C$$

Universality of NAND and NOR gates



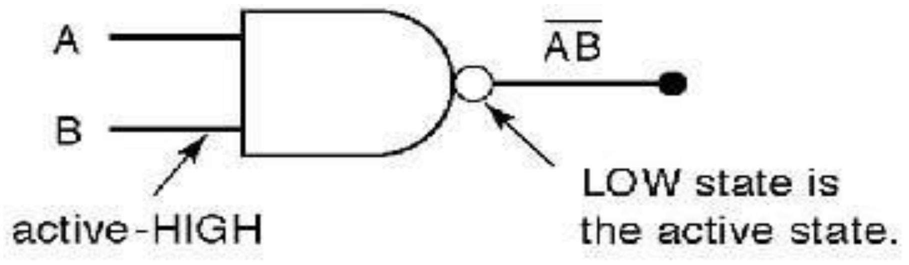
A
$x = \overline{A \cdot A} = \overline{A}$
(a)
A
INVERTER
A
B
1
$\overline{AB}$
2
$x = AB$
(b)
A
B
AND
A
1
$\overline{A}$
B
2
$\overline{B}$
3
$x = \overline{\overline{A}\,\overline{B}} = A + B$
(c)
A
B
OR

Universality of NOR gate



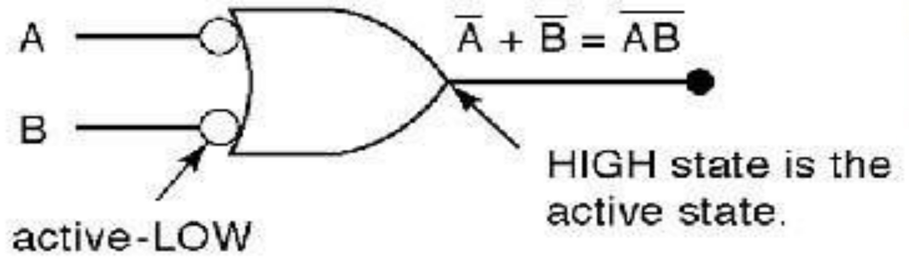
- Equivalent representations of the AND, OR, and NOT gates

Interpretation of the two NAND gate symbols



Output goes LOW only when all inputs are HIGH.

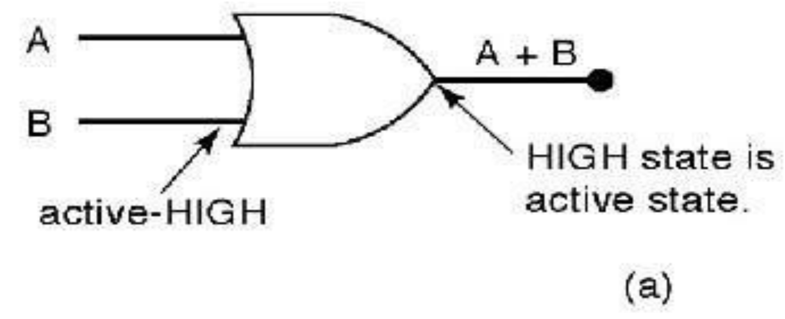
(a)


A
B
$\overline{A} + \overline{B} = \overline{AB}$
active-LOW
HIGH state is the active state.

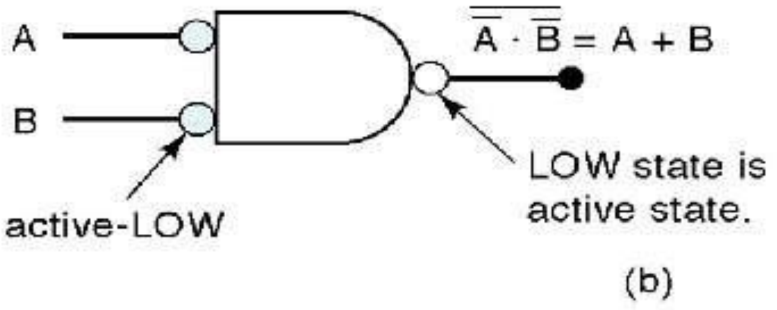
Output is HIGH when any input is LOW.

(b)

Interpretation of the two OR gate symbols



Output goes HIGH when any input is HIGH.


A
B
$\overline{\overline{A} \cdot \overline{B}} = A + B$
active-LOW
LOW state is active state.
(b)

Output goes LOW only when all inputs are LOW.

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 symbols
- Primitive gates can be combined to form larger circuits
- Boolean algebra defines how binary variables with NAND, NOR can be combined
- DeMorgan’s rules are important.
  - Allow conversion to NAND/NOR representations

More Boolean Algebra

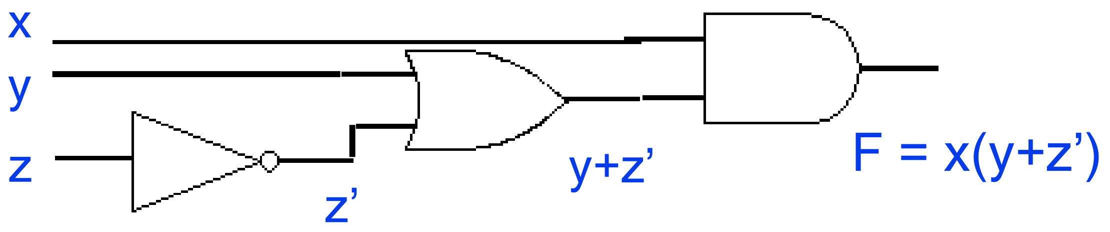
Overview

- Expressing Boolean functions
- Relationships between algebraic equations, symbols, and truth tables
- Simplification of Boolean expressions
- Minterms and Maxterms
- AND-OR representations
    - Product of sums
    - Sum of products

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

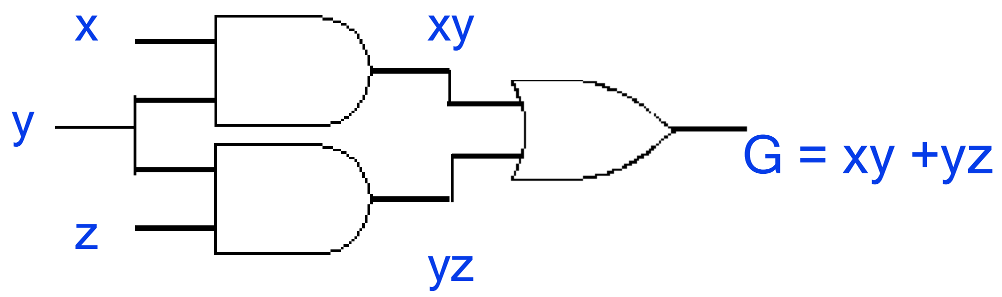


F = x(y+z')

Boolean Functions

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

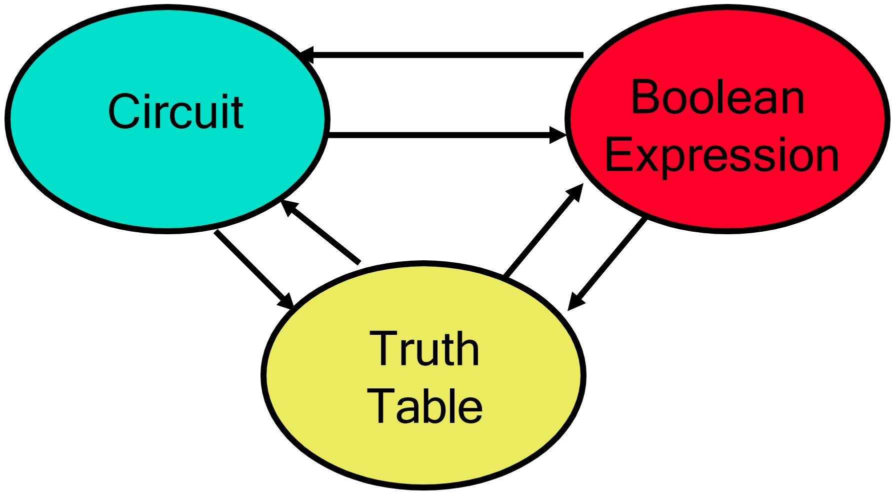
x	y	z	xy	yz	G
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	1	1



We will learn how to transition between equation, symbols, and truth table.

Representation Conversion

- Need to transition between boolean expression, truth table, and circuit (symbols).
- Converting between truth table and expression is easy.
- Converting between expression and circuit is easy.
- More difficult to convert to truth table.



Truth Table to Expression

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

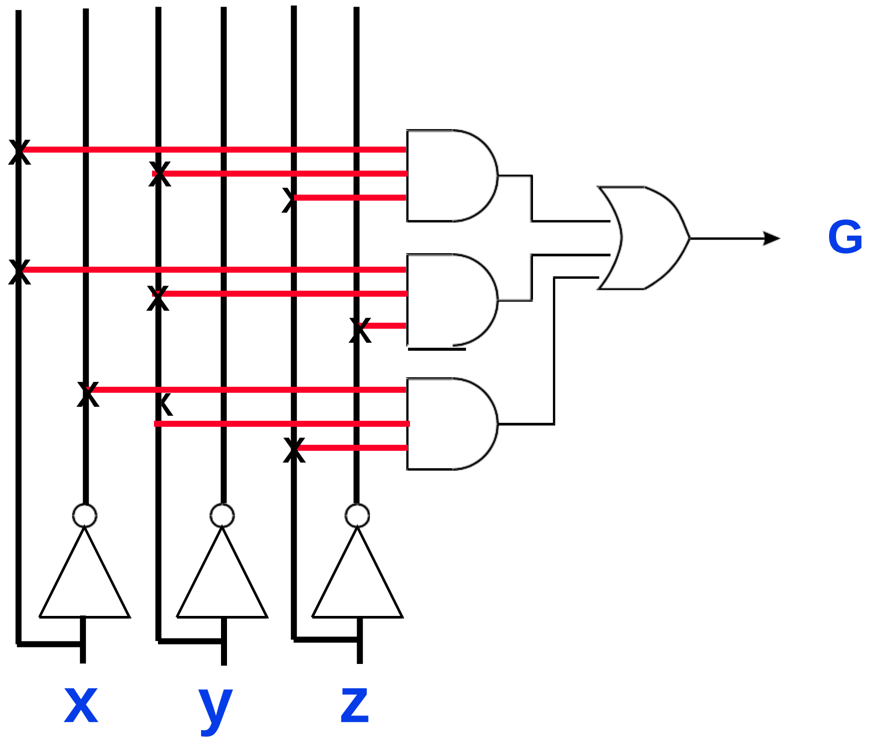
Any Boolean Expression can be represented in sum of products form!

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!

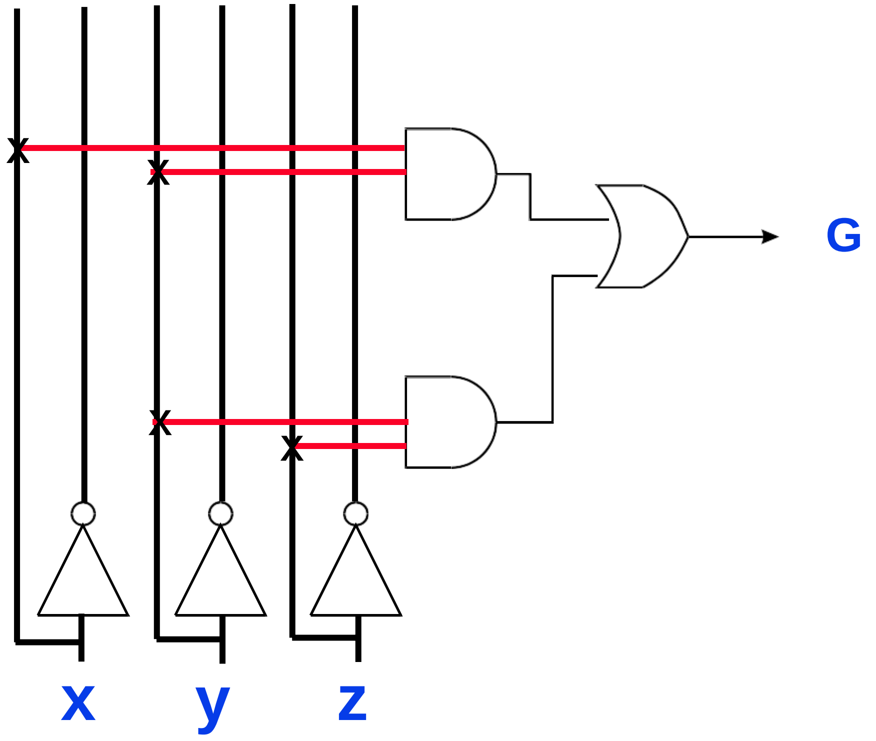
G = xyz + xyz’ + x’yz

- Use Boolean Algebra rules to reduce complexity while preserving functionality.
- Step 1: Use Theorum 1 (a + a = a)
  - So xyz + xyz’ + x’yz = xyz + xyz + xyz’ + x’yz
- Step 2: Use distributive rule a(b + c) = ab + ac
  - So xyz + xyz + xyz’ + x’yz = xy(z + z’) + yz(x + x’)
- Step 3: Use Postulate 3 (a + a’ = 1)
  - So xy(z + z’) + yz(x + x’) = xy.1 + yz.1
- Step 4: Use Postulate 2 (a . 1 = a)
  - So xy.1 + yz.1 = xy + yz = xyz + xyz’ + x’yz

Reduced Hardware Implementation

- Reduced equation requires less hardware!
- Same function 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 complement form (x')
- Each AND combination of terms is a minterm
- Each OR combination of terms is a maxterm

For example:
Minterms

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:
Maxterms

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 same as row position in truth table (starting from top from 0)
- 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 same as row position in truth table (starting from top from 0)
- 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
  - $b + c \rightarrow z$
  - $(a + z)' = G'$
- Step 2: Use DeMorgans' Law
  - $(a + z)' = a' . z'$
- Step 3: Resubstitute (b+c) for z
  - $a' . z' = a' . (b + c)'$
- Step 4: Use DeMorgans' Law
- $a' . (b + c)' = a' . (b'. c')$
- Step 5: Associative rule
  - $a' . (b'. c') = a' . b' . c'$

$G = a + b + c$

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

$G = a + b + c$

$G' = a' . b' . c' = a'b'c'$

Complementation Example

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

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

$$\downarrow$$

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

$$\downarrow$$

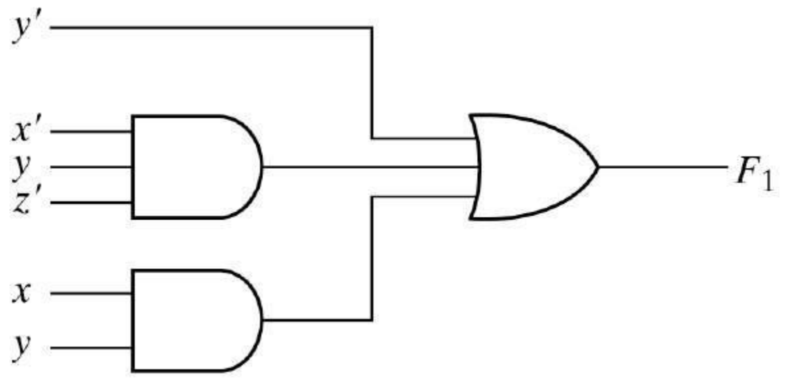
$$G = M_0M_1M_2M_4M_5 = \Pi(0,1,2,4,5)$$

$$\downarrow$$

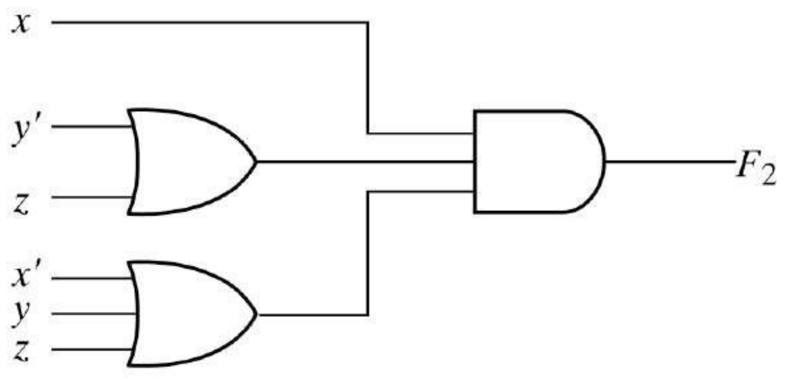
$$G = (x+y+z)(x+y+z')(x+y'+z)(x'+y+z)(x'+y+z')$$

Representation of Circuits

- All logic expressions can be represented in 2-level format
- Circuits can be reduced to minimal 2-level representation
- Sum of products representation most common in industry.



(a) Sum of Products



(b) Product of Sums

Fig. 2-3 Two-level implementation

Summary

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

Minimization with Karnaugh Maps

Overview

- K-maps: an alternate approach to representing Boolean functions
- K-map representation can be used to minimize Boolean functions
- Easy conversion from truth table to K-map to minimized SOP representation.
- Simple rules (steps) used to perform minimization
- Leads to minimized SOP representation.
    - Much faster and more more efficient than previous minimization techniques with Boolean algebra.

Karnaugh maps

- Alternate way of representing Boolean function
  - All rows of truth table represented with a square
  - Each square represents a minterm
- Easy to convert between truth table, K-map, and SOP
  - Unoptimized form: number of 1's in K-map equals number of minterms (products) in SOP
  - Optimized form: reduced number of minterms

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

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

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

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.

A \ B	0	1
0	0	1
1	1	0

$F = AB' + A'B$

A \ B	0	1
0	0	1
1	1	1

$F = AB + A'B + AB'$

- 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

$$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$$

Rules for K-Maps

- We can reduce functions by circling 1’s in the K-map
- Each circle represents minterm reduction
- Following circling, we can deduce minimized and-or form.

Rules to consider

ÊEvery cell containing a 1 must be included at least once.

ËThe largest possible “power of 2 rectangle” must be enclosed.

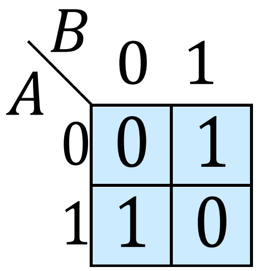
Ì The 1’s must be enclosed in the smallest possible number of rectangles.

Example

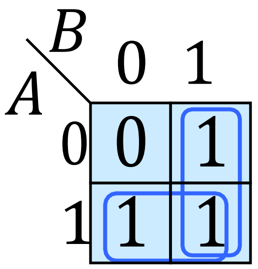
$\longrightarrow$

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.



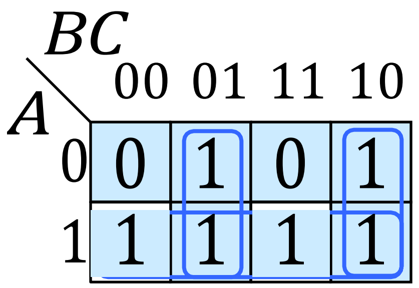
$F = AB' + A'B$



$F = AB + A'B + AB'$

$F = A + B$

- Three variable maps.

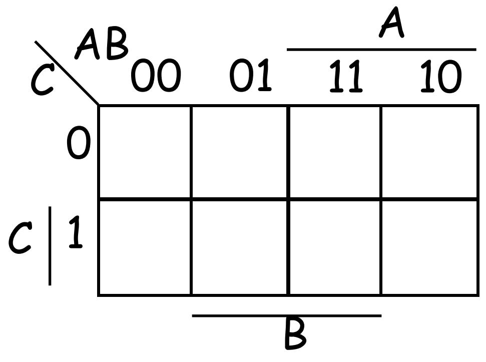


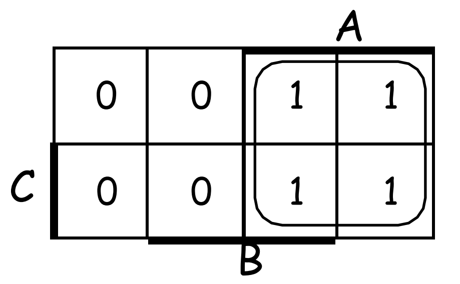
$F = A + B'C + BC'$

$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

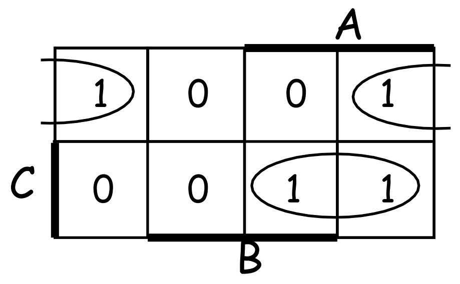
Karnaugh maps

- **Numbering scheme based on Gray–code**
  - e.g., 00, 01, 11, 10
  - Only a single bit changes in code for adjacent map cells
  - This is necessary to observe the variable transitions





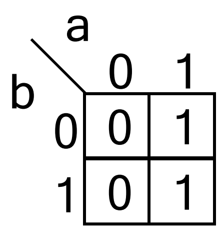
$G(A,B,C) = A$



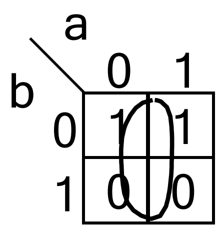
$F(A,B,C) = \Sigma m(0,4,5,7) = AC + B'C'$

More Karnaugh Map Examples

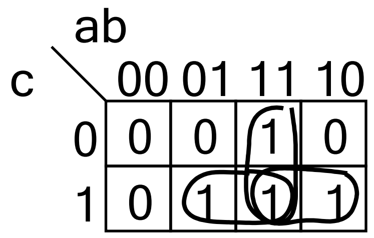
- **Examples**



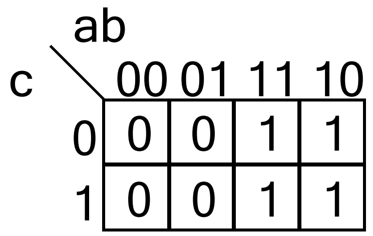
f = a



g = b'



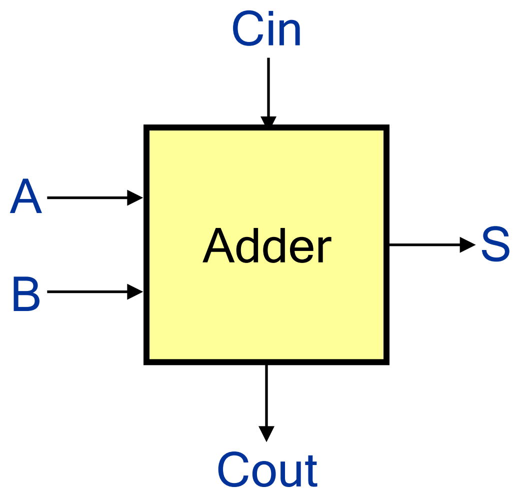
cout = ab + bc + ac



f = a

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	Cin	S	Cout
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'Cin + A'BCin' + A'BCin + ABCin

Cout = A'BCin + A B'Cin + ABCin' + ABCin

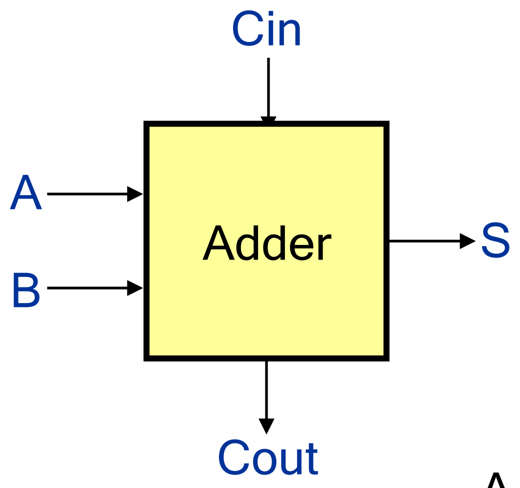
= A'BCin + ABCin + AB'Cin + ABCin + ABCin' + ABCin

= (A' + A)BCin + (B' + B)ACin + (Cin' + Cin)AB

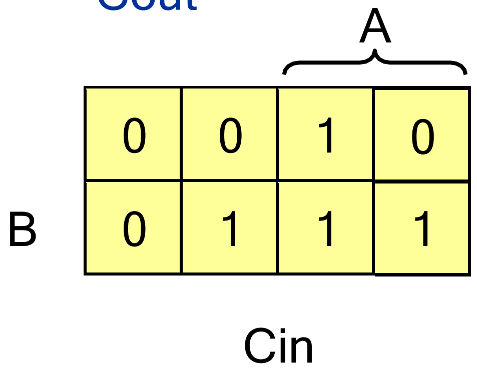
= 1·BCin + 1· ACin + 1· AB

= BCin + ACin + AB

Application of Karnaugh Maps: The One-bit Adder



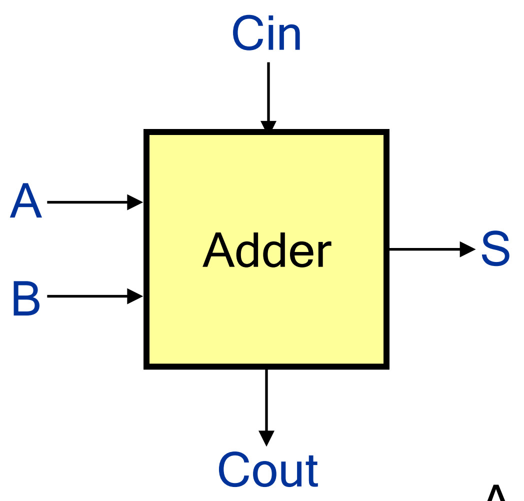
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



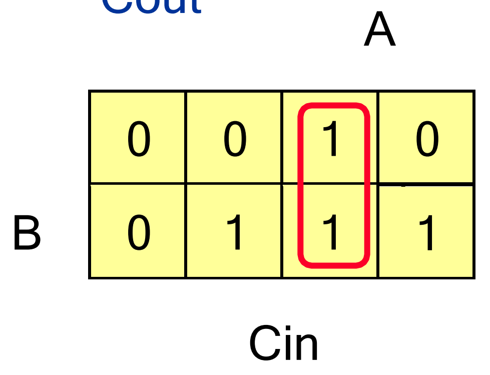
Karnaugh Map for Cout

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

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
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



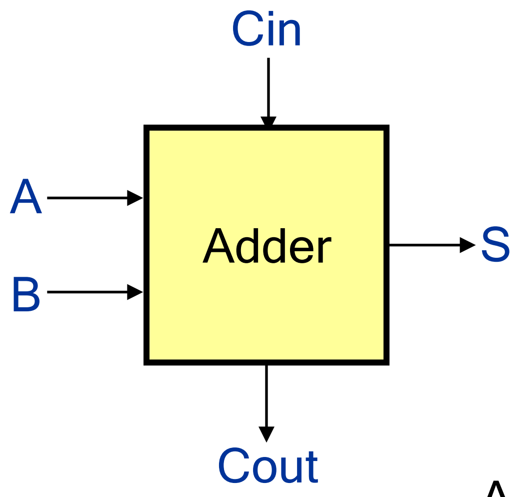
0	0	1	0
0	1	1	1

Karnaugh Map for Cout

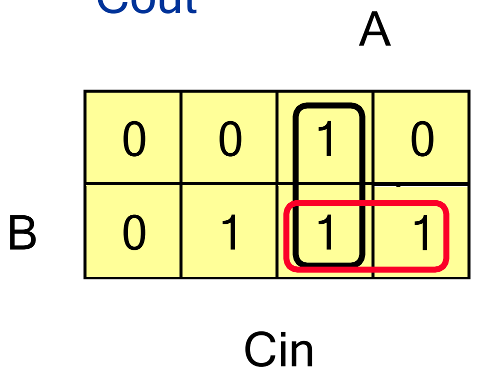
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as ~~we can~~.

Cout = ACin

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
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

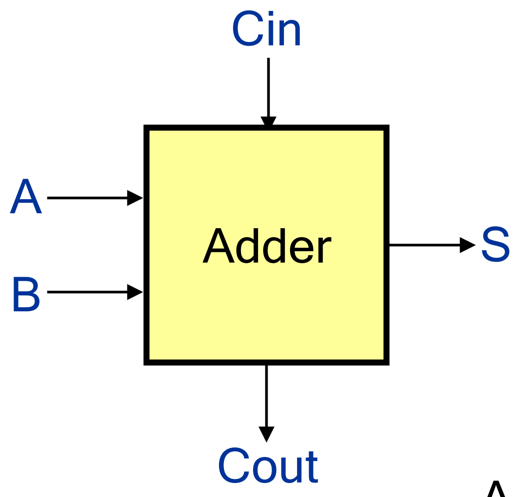


Karnaugh Map for Cout

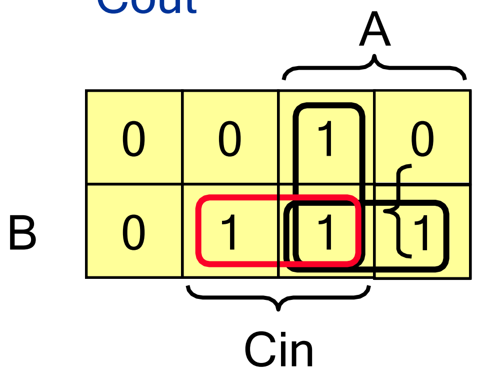
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Cout = Acin + AB

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
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

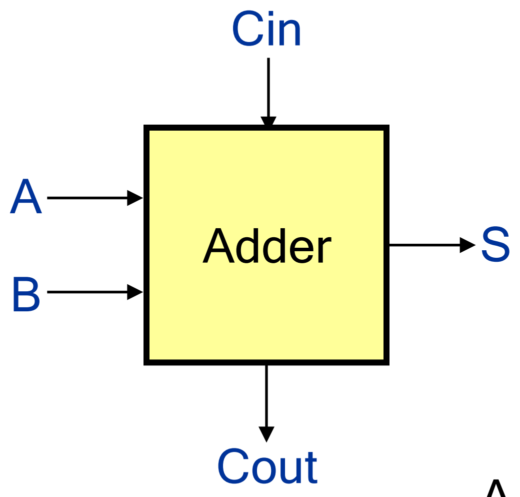


Karnaugh Map for Cout

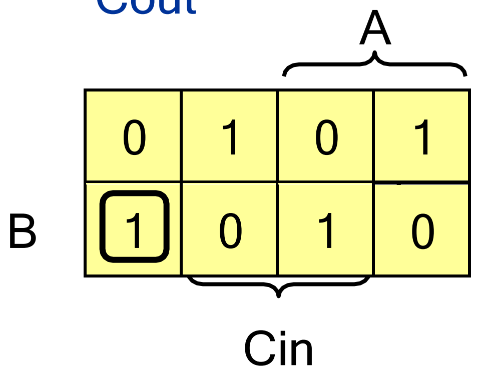
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Cout = ACin + AB + BCin

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
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

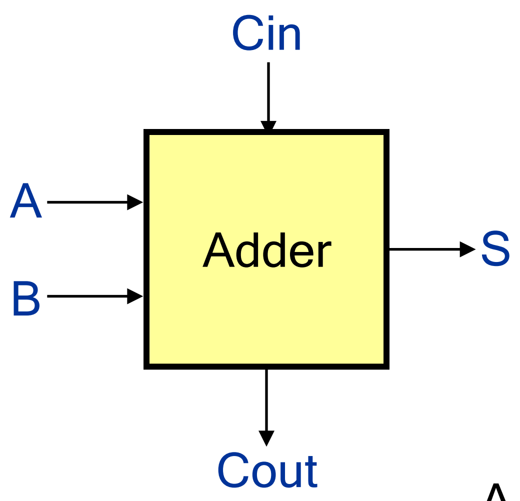


	0	1	0	1
B	$\boxed{1}$	0	1	0

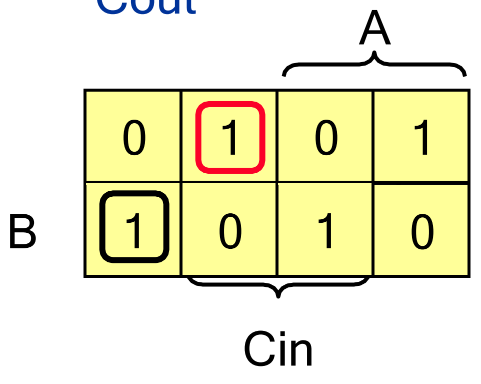
Karnaugh Map for S

S = A’BCin’

Application of Karnaugh Maps: The One-bit Adder



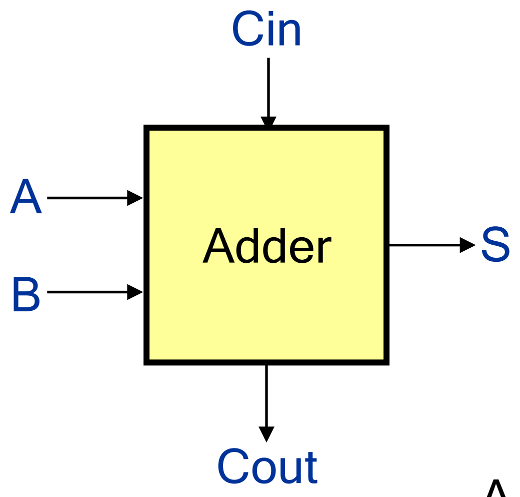
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



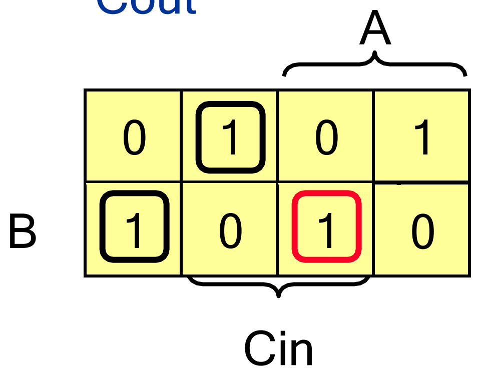
Karnaugh Map for S

S = A'BCin' + A'B'Cin

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
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

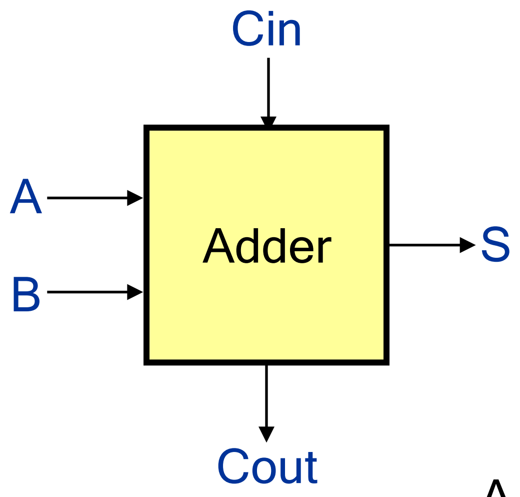


Karnaugh Map for S

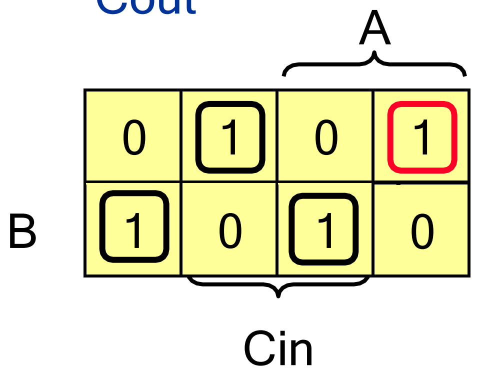
S = A’BCin’ + A’B’Cin + ABCin

Application of Karnaugh Maps: The One-bit Adder

Can you draw the circuit diagrams?



A	B	Cin	S	Cout
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



Karnaugh Map for S

S = A’BCin’ + A’B’Cin + ABCin + AB’Cin’

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
- Each circle represents an application of:
  - Distributive rule -- $x(y + z) = xy + xz$
  - Complement rule – $x + x' = 1$

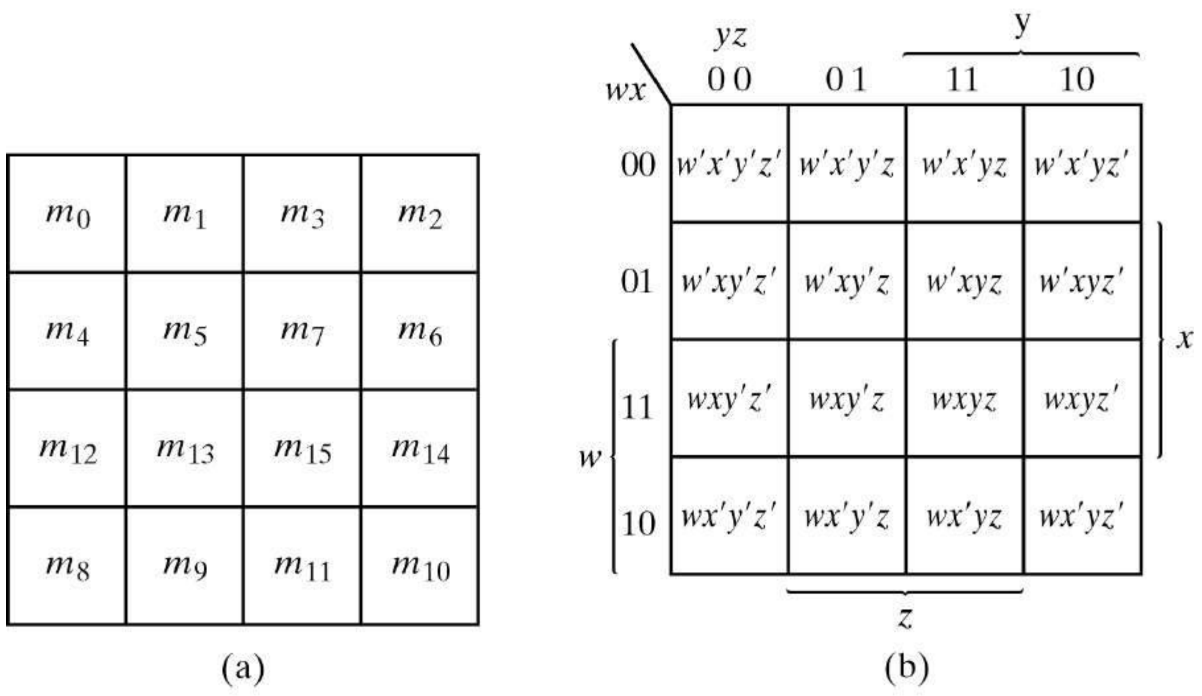
More Karnaugh Maps and Don’t Cares

Overview

- **Karnaugh maps with four inputs**
  - Same basic rules as three input K-maps
- **Understanding prime implicants**
  - Related to minterms
- **Covering all implicants**
- **Using Don’t Cares to simplify functions**
  - Don’t care outputs are undefined
- **Summarizing Karnaugh maps**

Karnaugh Maps for Four Input Functions

- Represent functions of 4 inputs with 16 minterms
- Use same rules developed for 3-input functions
- Note bracketed sections shown in example.



(a)

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}

(b)

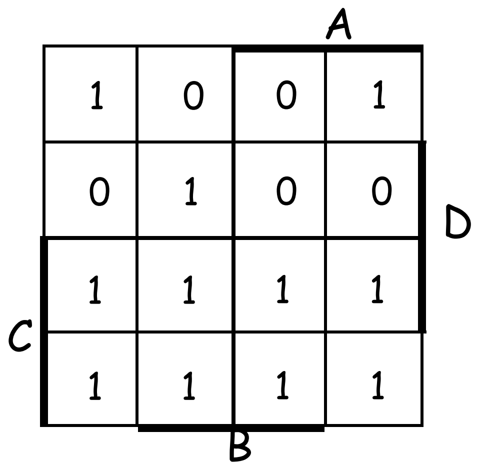
$wx \backslash yz$	00	01	11	10
00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

Fig. 3-8 Four-variable Map

Karnaugh map: 4-variable example

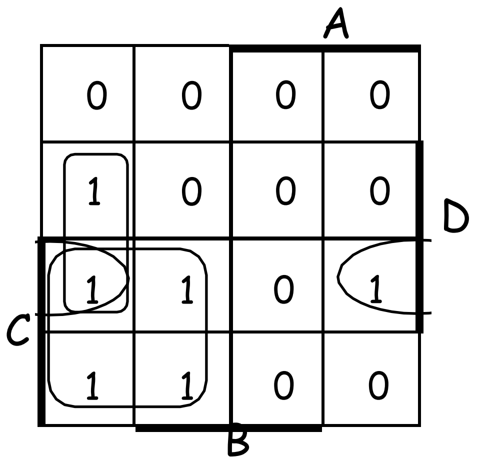
- F(A,B,C,D) = □m(0,2,3,5,6,7,8,10,11,14,15)

  F =C+A’BD+B’D’

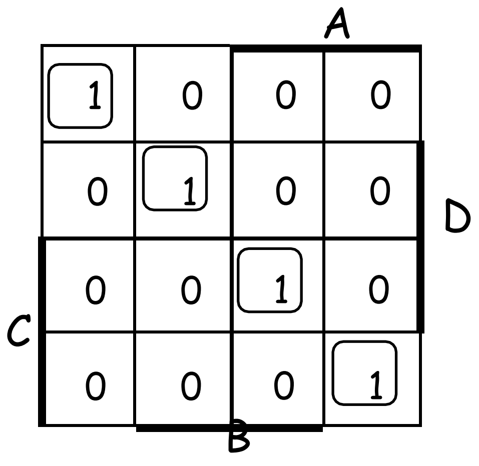


1	0	0	1
0	1	0	0
1	1	1	1
1	1	1	1

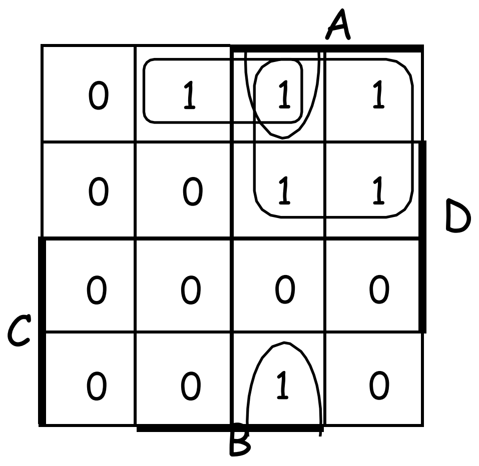
Design examples



K-map for LT



K-map for EQ



K-map for GT

LT = A' B' D + A' C + B' C D

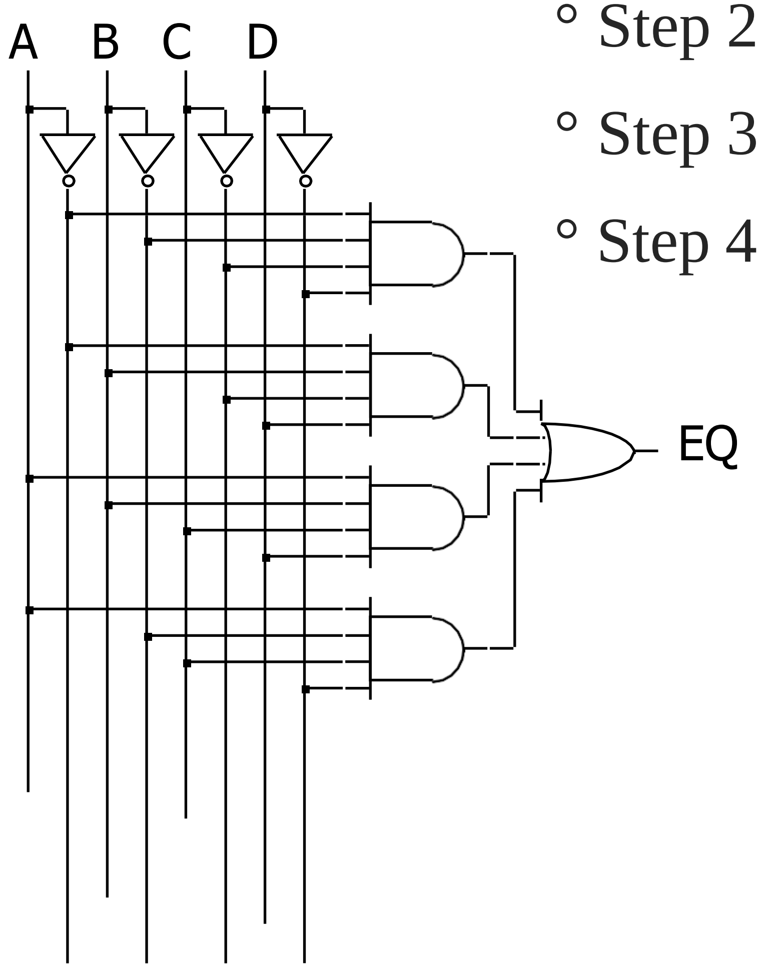
EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD'

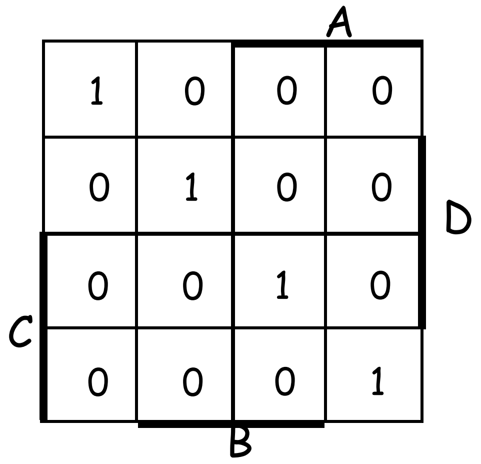
GT = B C' D' + A C' + A B D'

Can you draw the truth table for these examples?

Physical Implementation

- Step 1: Truth table
- Step 2: K-map
- Step 3: Minimized sum-of-products
- Step 4: Physical implementation with gates



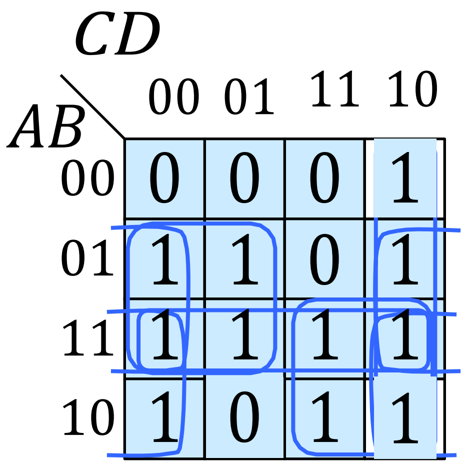


1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

K-map for EQ

Karnaugh Maps

- Four variable maps.



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

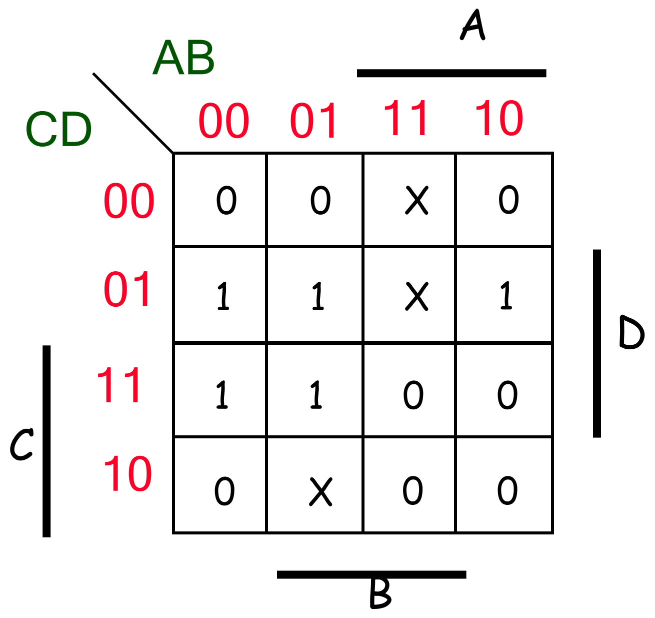
$$F = A'BC' + A'CD' + ABC + AB'C'D' + ABC' + AB'C$$

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

- 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 logic produces a 0 or a 1
- This knowledge can be used to simplify functions.



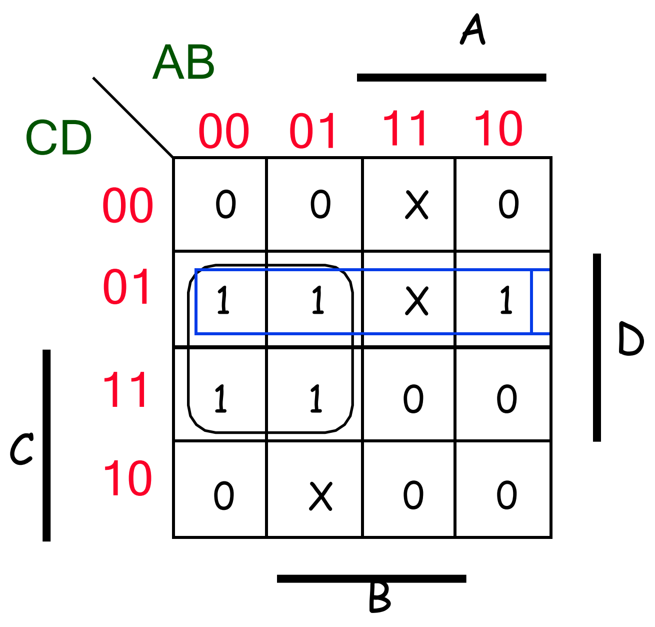
CD \ AB	00	01	11	10
00	0	0	X	0
01	1	1	X	1
11	1	1	0	0
10	0	X	0	0

- 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) = Σ m(1,3,5,7,9) + d(6,12,13)
  - without don't cares
    - f =

A'D + C'D

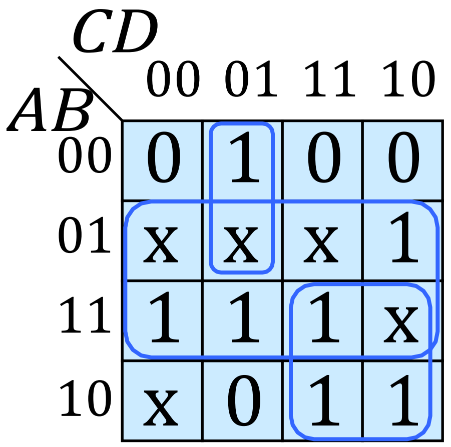


A	B	C	D	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	X
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	X
1	1	0	1	X
1	1	1	0	0
1	1	1	1	0

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 in 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.

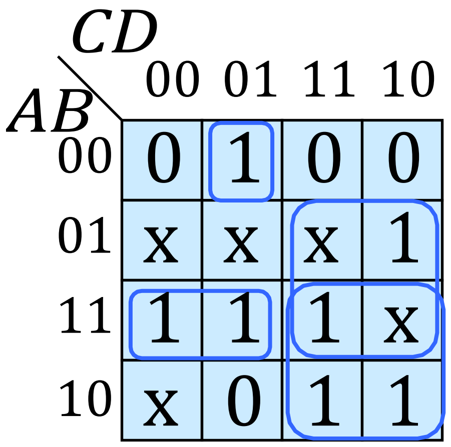
Map Simplification with Don’t Cares



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

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

- Alternative covering.



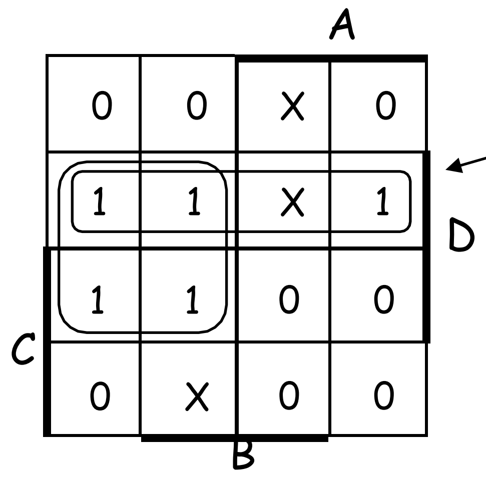
AB \ CD	00	01	11	10
00	0	1	0	0
01	x	x	x	1
11	1	1	1	x
10	x	0	1	1

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

Karnaugh maps: don’t cares (cont’d)

- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - f = A'D + B'C'D without don't cares
  - f = with don't cares

A'D + C'D



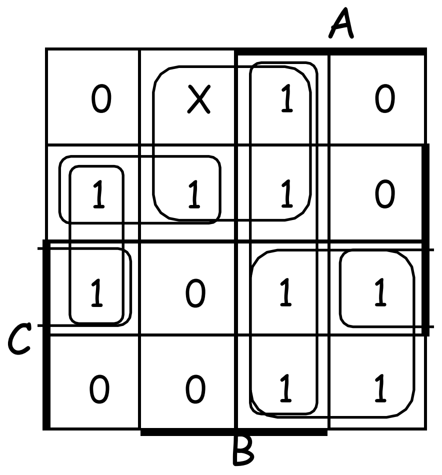
by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

<u>don't cares</u> can be treated as
1s or 0s
depending on which is more
advantageous

Definition of terms for two-level simplification

- **Implicant**
  - Single product term of the ON-set (terms that create a logic 1)
- **Prime implicant**
  - Implicant that can't be combined with another to form an implicant with fewer literals.
- **Essential prime implicant**
  - Prime implicant is essential if it alone covers a minterm in the K-map
  - Remember that all squares marked with 1 must be covered
- **Objective:**
  - Grow implicant into prime implicants (minimize literals per term)
  - Cover the K-map with as few prime implicants as possible (minimize number of product terms)

Examples to illustrate terms



6 prime implicants:

A'B'D, BC', AC, A'C'D, AB, B'CD

essential

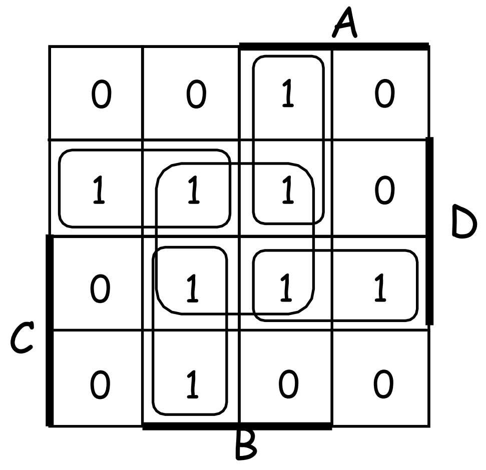
minimum cover: AC + BC' + A'B'D

5 prime implicants:

BD, ABC', ACD, A'BC, A'C'D

essential

minimum cover: 4 essential implicants

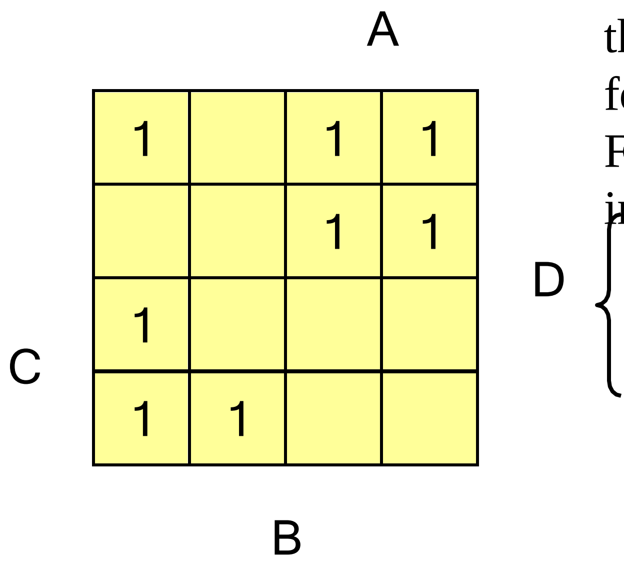


PRIME IMPLICANTS

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.

Example:



If a function F is represented by this Karnaugh Map. Which of the following terms are implicants of F, and which ones are prime implicants of F?

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

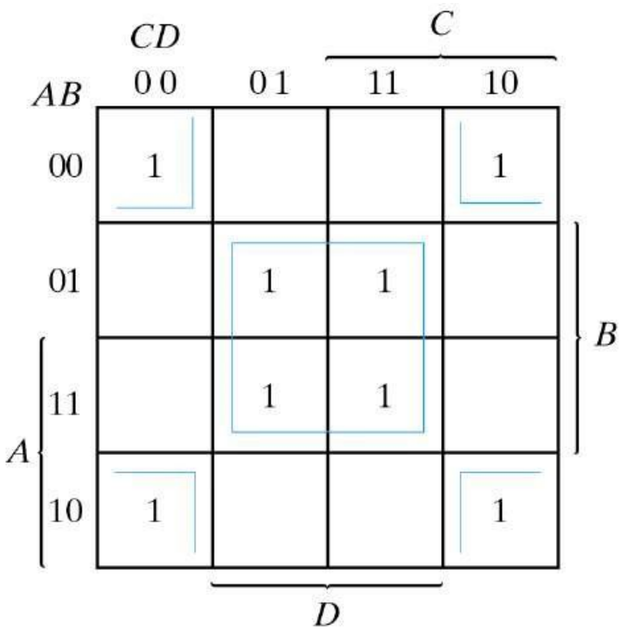
Implicants: (a),(c),(d),(e)

Prime Implicants: (d),(e)

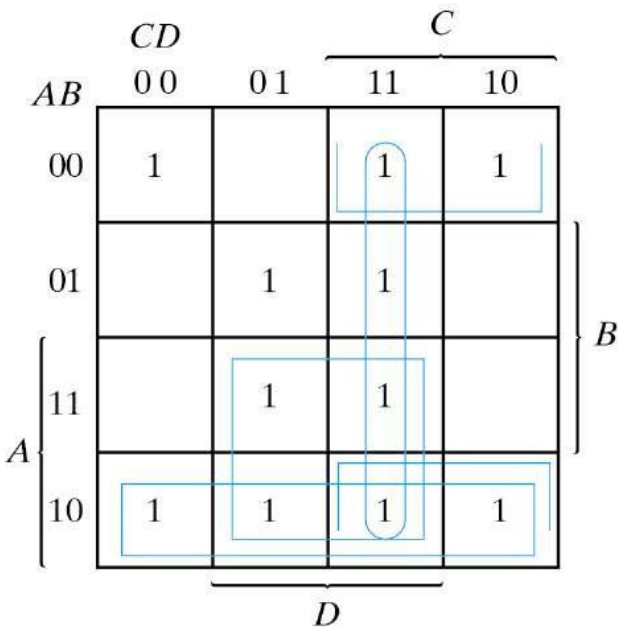
Essential Prime Implicants

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

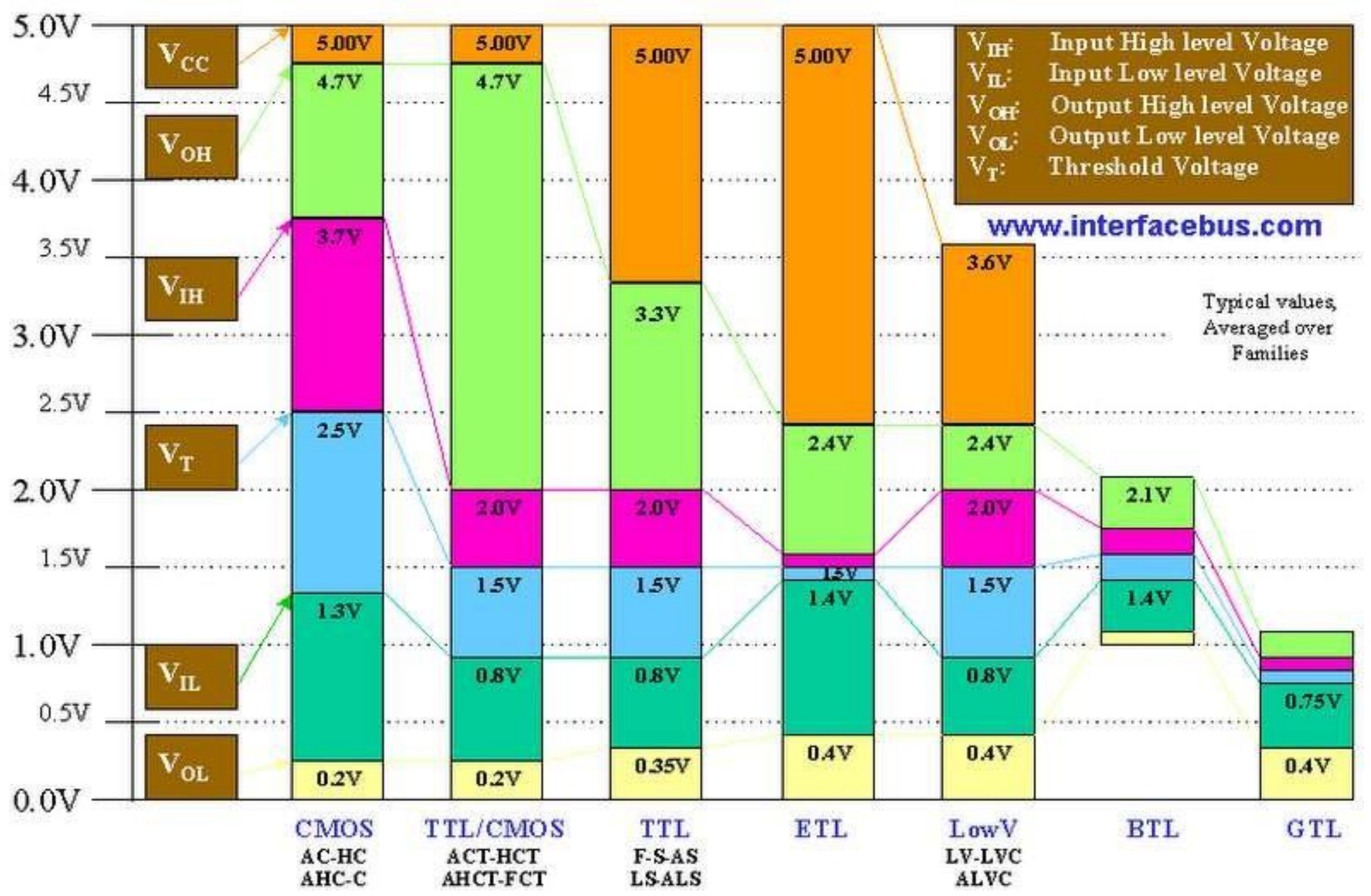
Summary

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

DIGITAL ELECTRONIC CIRCUITS

CMOS LOGIC CIRCUITS

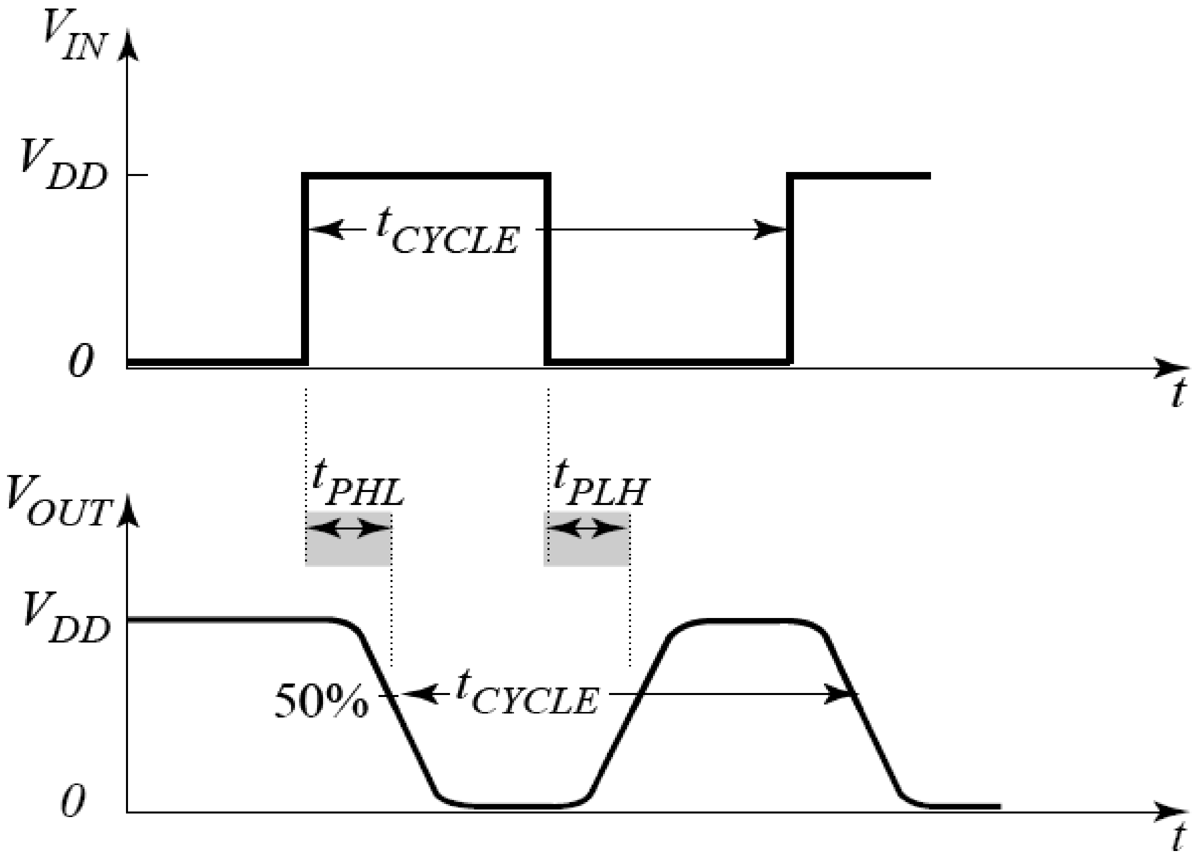
VOLTAGE AS LOGIC VARIABLE



- V_{OH}:- maximum output voltage when the output level is logic "1"
- V_{OL}:- minimum output voltage when the output level is logic "0"

LOGIC DELAY TIMES

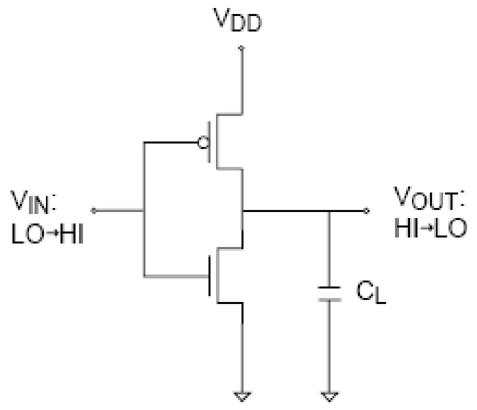
- Inverter propagation delay: time delay between input and output signals.
- Typical propagation delays: < 1 ns.
- Estimation of t_p: use of square-wave at input side.

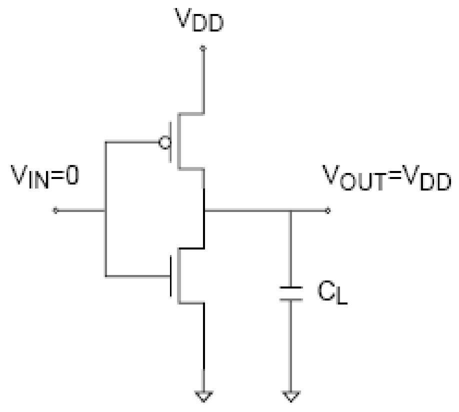


LOGIC DELAY TIMES--continued

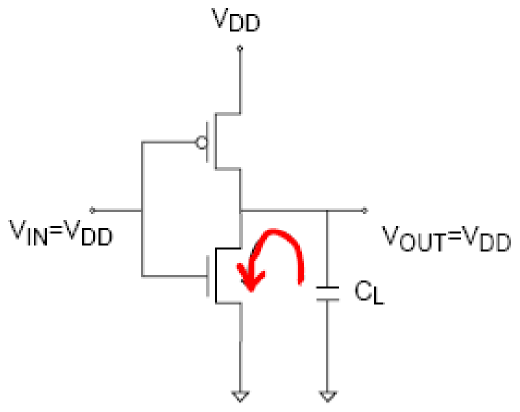
- Average propagation delay: $t_p = \frac{1}{2}(t_{PHL} + t_{PLH})$
- Where t_p = propagation delay
- t_{PHL} = time taken to drive output from high to low
- t_{PLH} = time taken to drive output from low to high

PROPAGATION DELAY HIGH-TO-LOW:

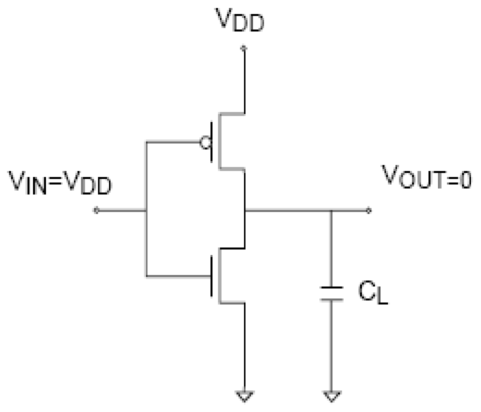




t=0-

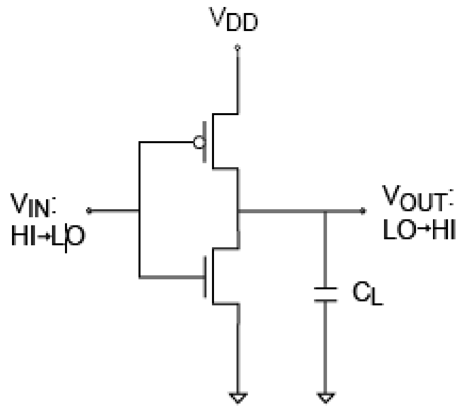


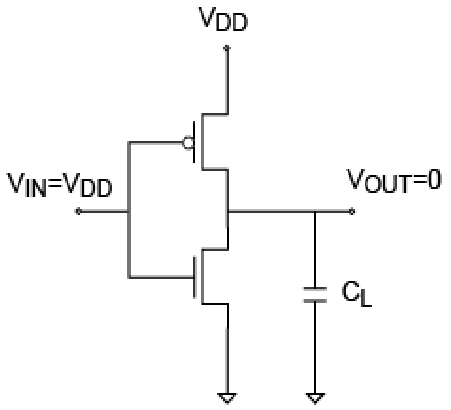
t=0+



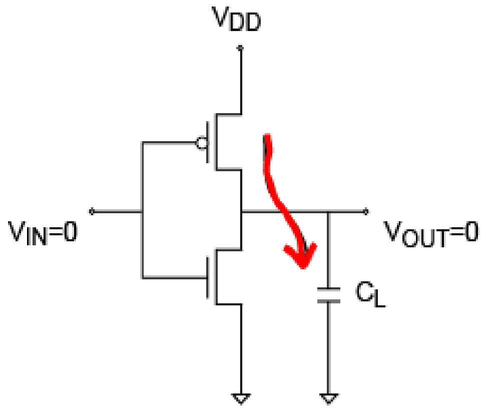
t→∞

PROPAGATION DELAY LOW-TO-HIGH:

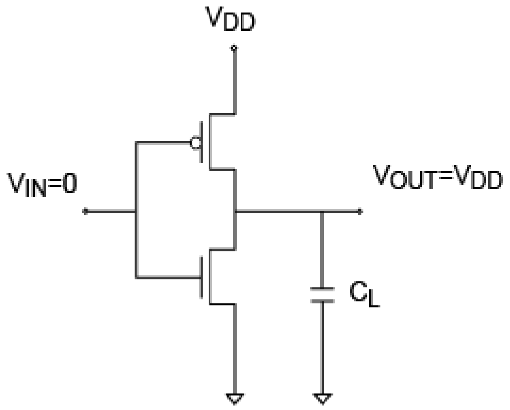




t=0-



t=0+

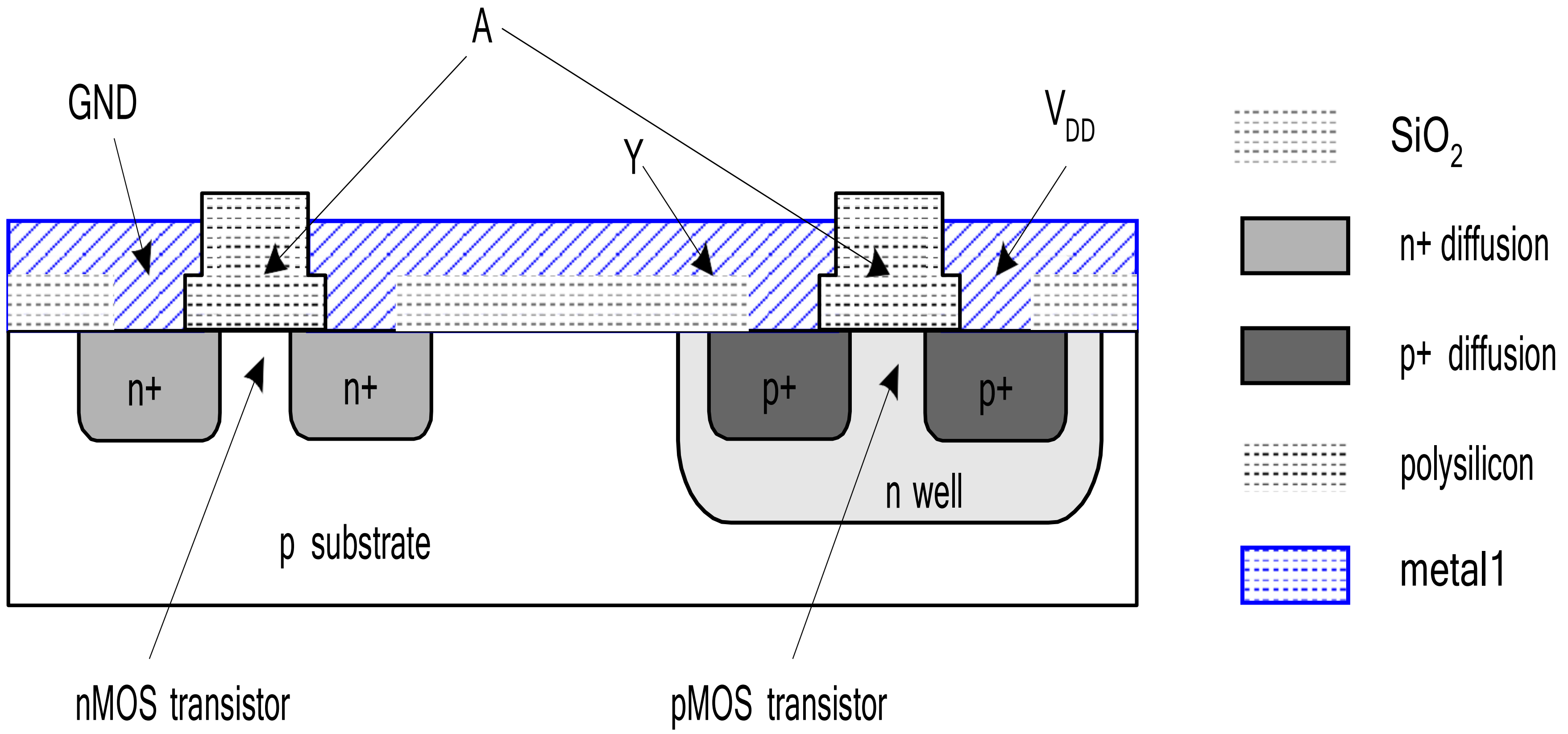


t→∞

FAN-IN AND FAN-OUT

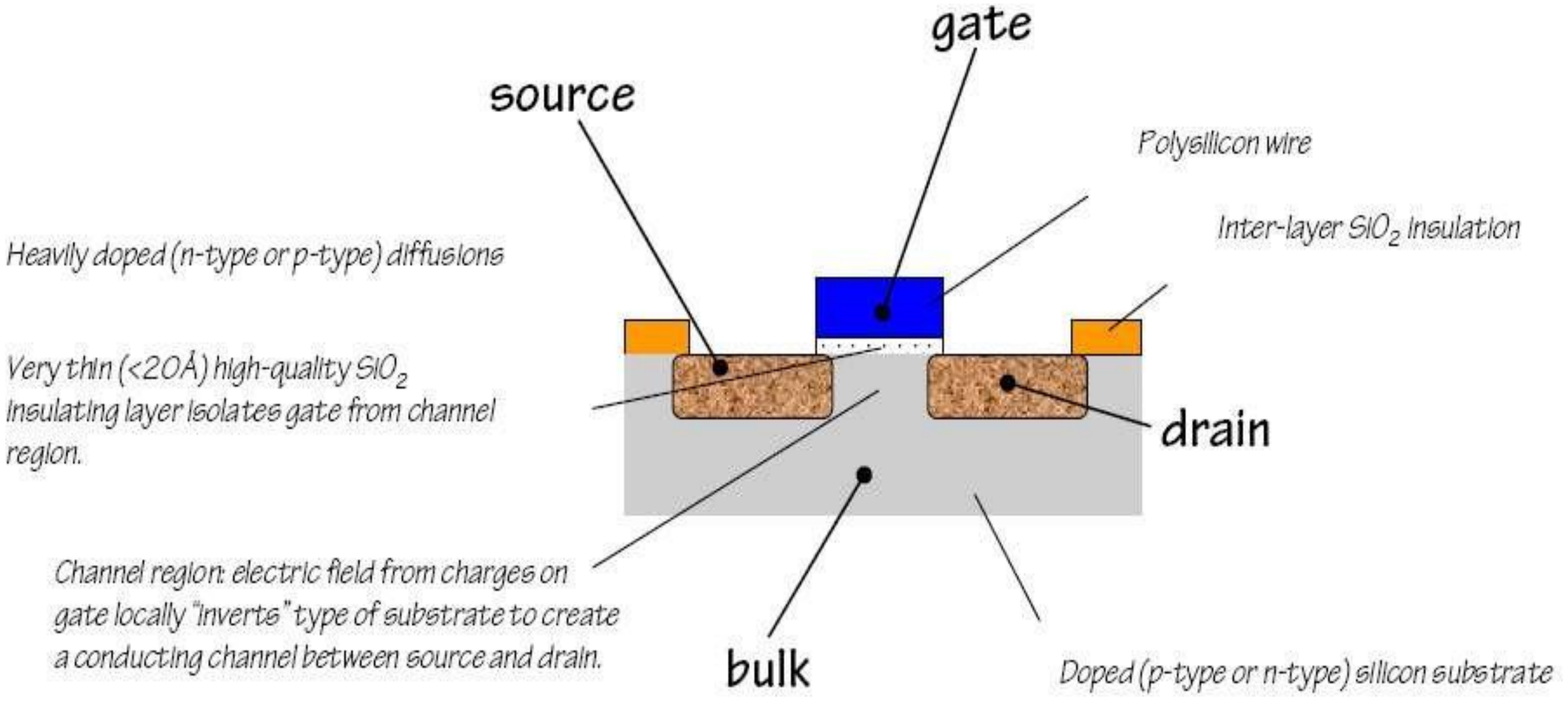
- Fan-in is the number of inputs of an electronic logic gate which it can drive at a time. For instance the 'fan-in' for the AND gate shown below is 3. Logic gates with a large fan-in tend to be slower than those with a small fan-in, because the complexity of the input circuitry increases the input capacitance of the device.
- Fan-out is a measure of the ability of a logic gate output, implemented electronically, to drive a number of inputs of other logic gates of the same type. In most designs, logic gates are connected together to form more complex circuits, and it is common for one logic gate output to be connected to several logic gate inputs.

C-MOS ELECTRONICS



- It is basically combination of a n-mosfet and a p-mosfet as shown in above figure.

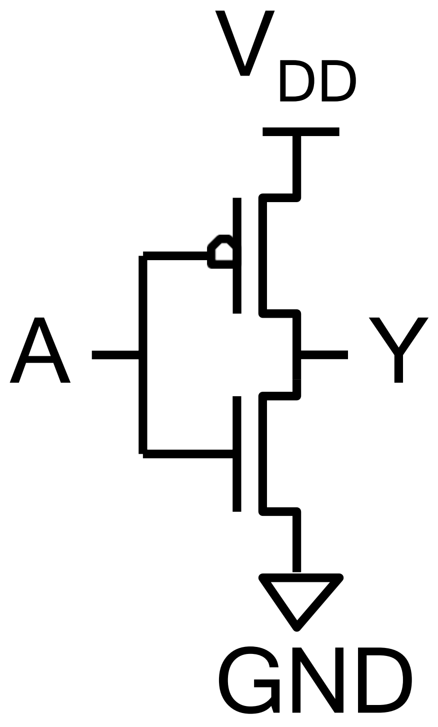
MOSFET

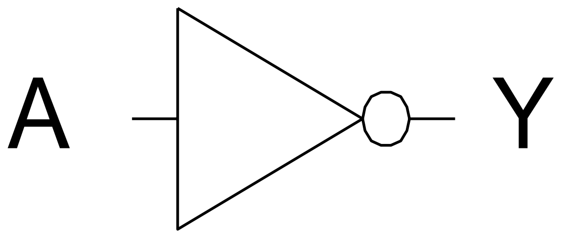

gate
source
Polysilicon wire
Inter-layer SiO_2 insulation
Heavily doped (n-type or p-type) diffusions
Very thin (<20Å) high-quality SiO_2 insulating layer isolates gate from channel region.
drain
Channel region: electric field from charges on gate locally "inverts" type of substrate to create a conducting channel between source and drain.
bulk
Doped (p-type or n-type) silicon substrate

- MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistors) are four terminal voltage-controlled switches.
- Current flows between the diffusion terminals if the voltage on the gate terminal is large enough to create a conducting "channel", otherwise the diffusion terminals are not connected.

NOT IN C-MOS

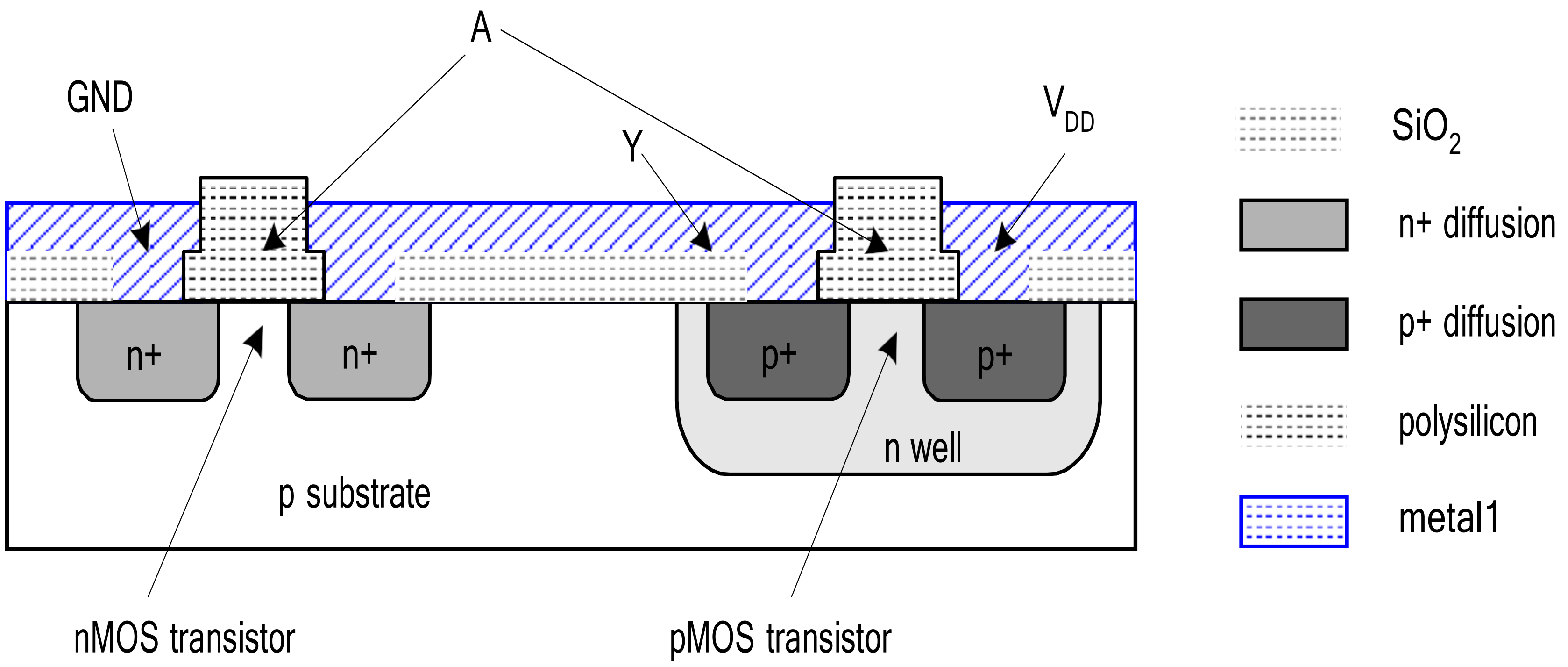
A	Y
0	**1**
1	0





C-MOS INVERTER

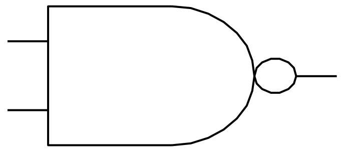
- Typically use p-type substrate for nMOS transistors
- Requires n-well for body of pMOS transistors

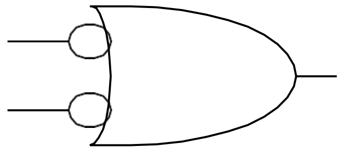


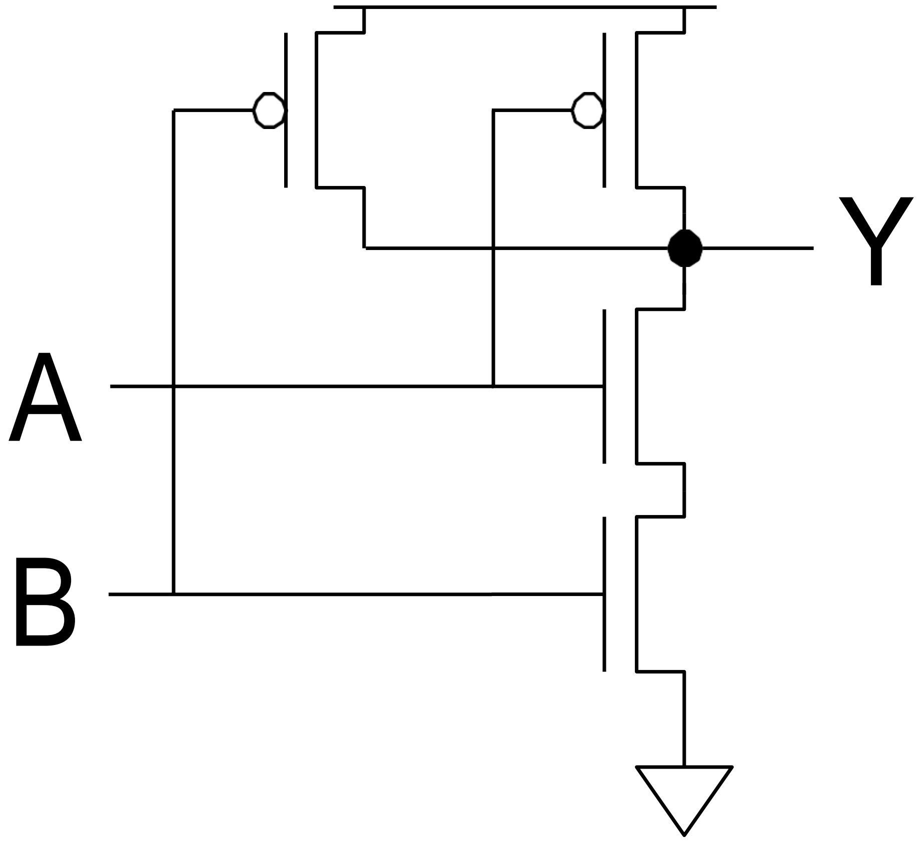
LOGIC FORMATION USING C-MOS

- **C-MOS NAND GATE**

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



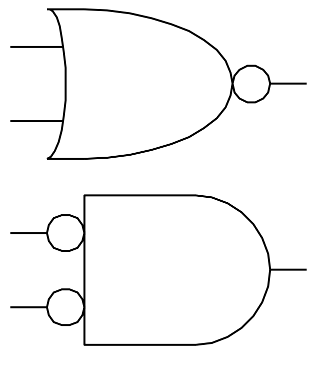


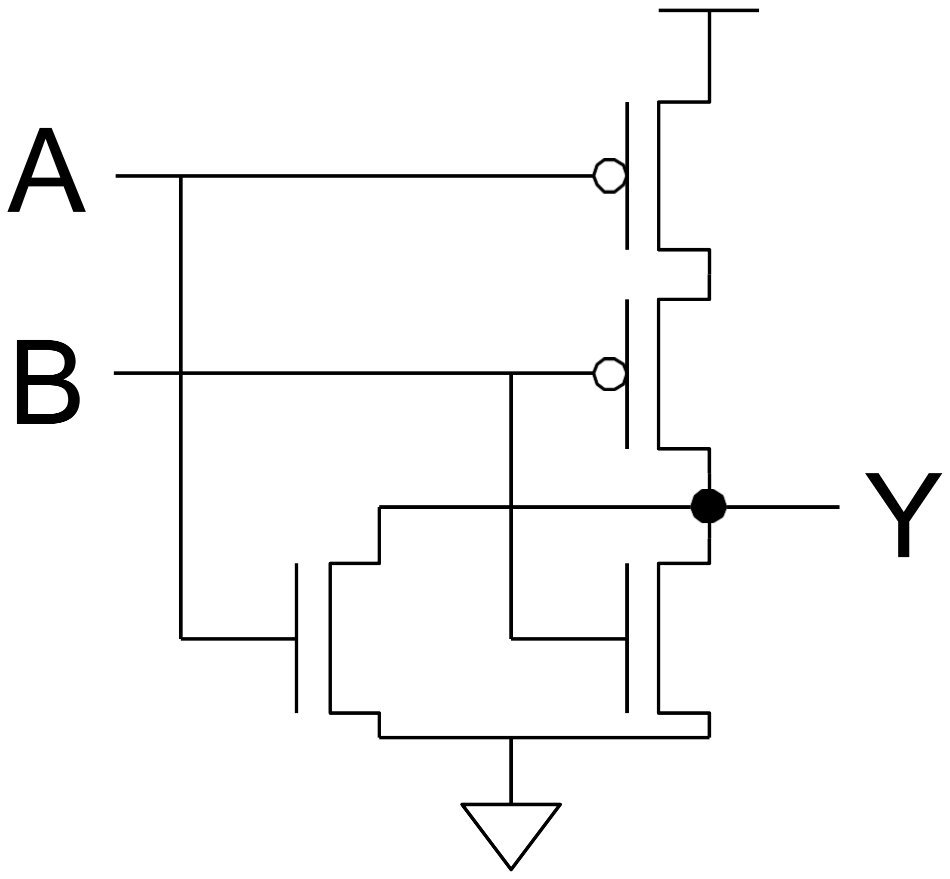


LOGIC FORMATION USING C-MOS

- **C-MOS NOR GATE**

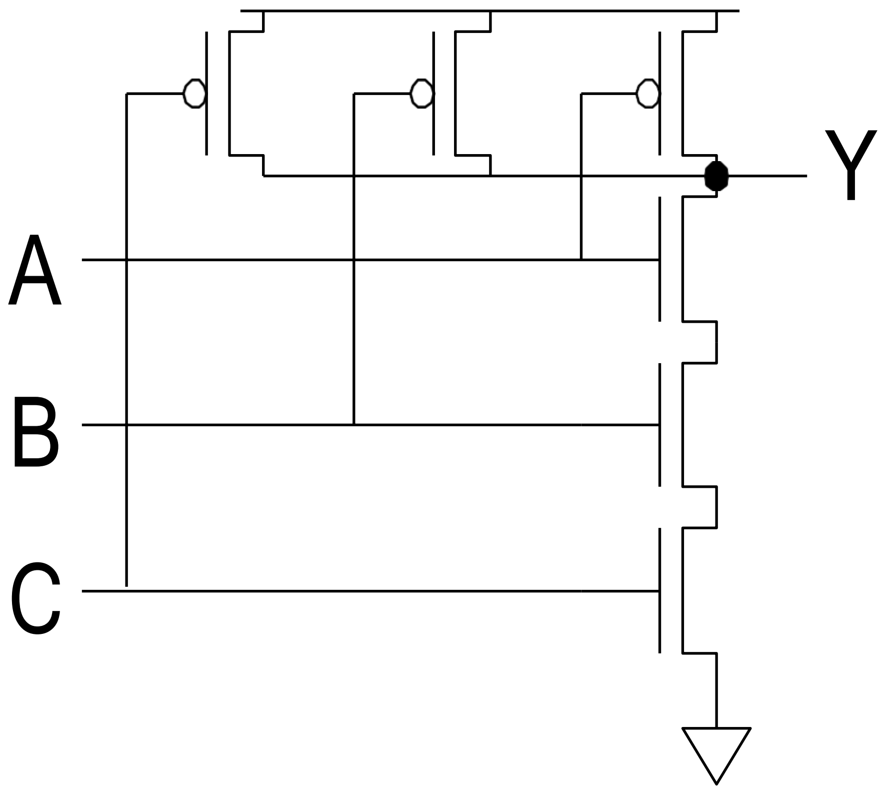
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0





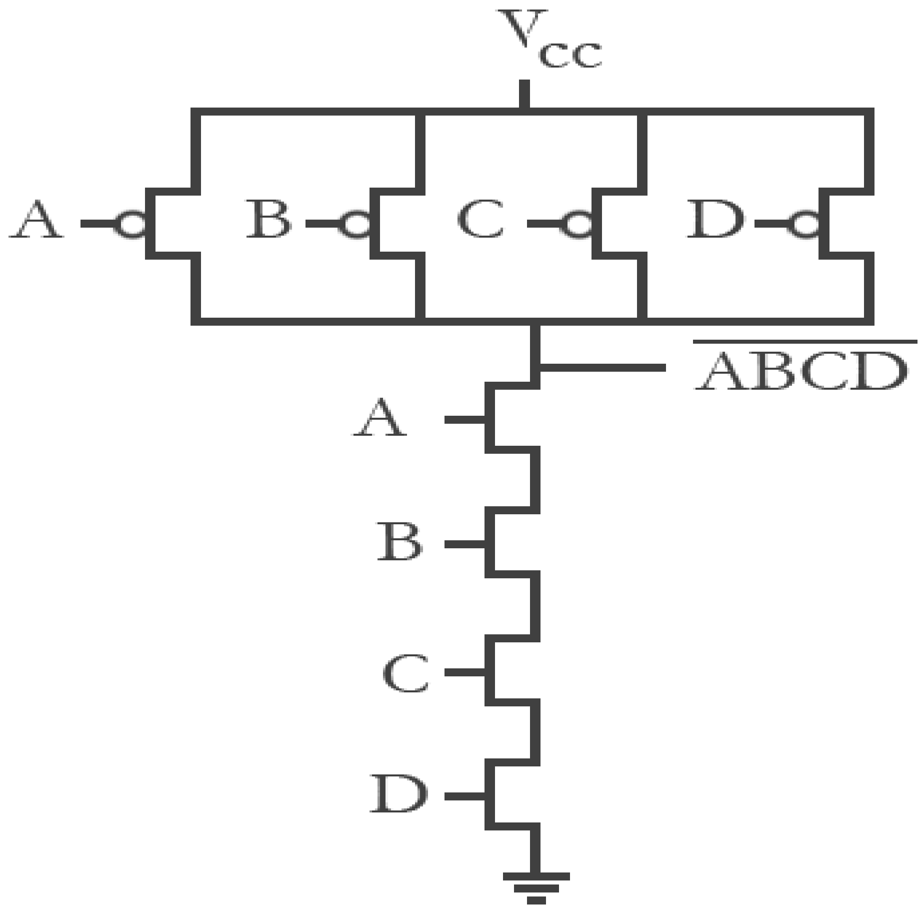
COMPLEX LOGIC GATES USING C-MOS

- 3-input NAND gate
- Y pulls low if ALL inputs are 1
- Y pulls high if ANY input is 0



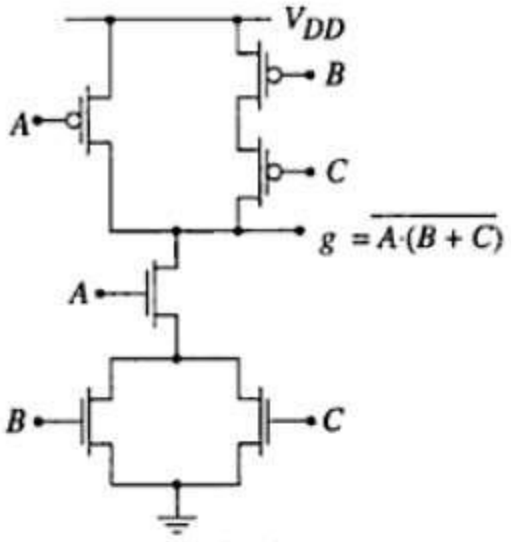
COMPLEX LOGIC GATES USING C-MOS

- 4-input NAND gate
- O/P pulls low if ALL inputs are 1
- O/P pulls high if ANY input is 0



LOGIC CASCADES

- A series and/or parallel network combinations of logic gates such as and,or,nor,nand,not gates etc to form a specific logic, is known to be cascade structure.
- Example→



(a) Logic circuit

A	B	C	g	
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	1	
1	0	1	0	← $A\cdot C=1$
1	1	0	0	← $A\cdot B=1$
1	1	1	0	← $A\cdot(B+C)=1$

(b) Function table

Figure 5.31 A complex logic CMOS gate circuit

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

Digital electronics circuits

CONCEPT OF DIGITAL COMPONENTS

DIGITAL COMPONENTS

- Digital components are mainly the devices in which inputs are digital data and the outputs are also in digital format
- It is mainly categorized into 2 types i.e.

    1.combinational components(circuits)

    2.sequential components(circuits)

COMBINATIONAL COMPONENTS

- A combinational circuit consists of logic gates whose outputs at any time is determined from only the present combination of inputs. It can be specified logically by a set of Boolean functions.

SEQUENTIAL COMPONENTS

- A sequential circuit is a combination of combinational logic circuit with a memory component. Here the output at one stage is the function of present state input and previous outputs.

BINARY ADDERS

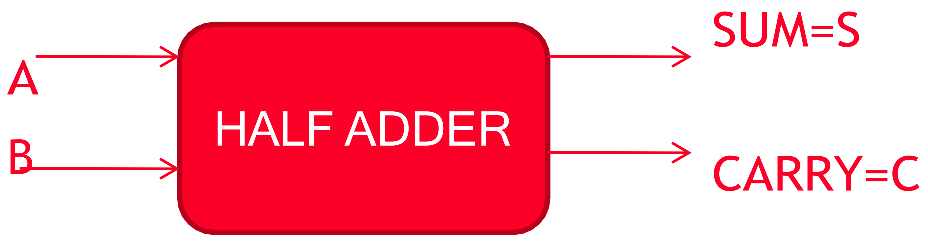
- Binary adders are the digital devices which add binary numbers
- Its of two types i.e.

  1.Half adder

  2.Full adder

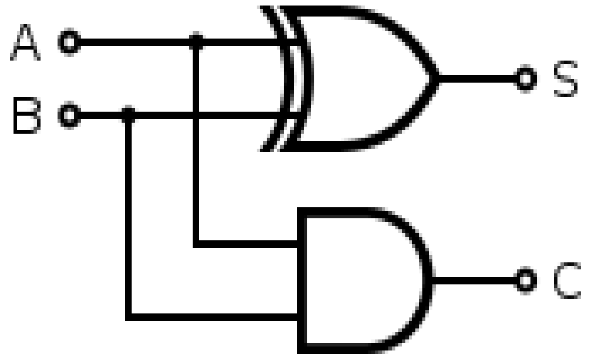
HALF ADDER

- A half adder can add two bits. It has two inputs, generally labeled *A* and *B*, and two outputs, the sum *S* and carry *C*



- Sum=S=A (XOR) B
- Carry=C=A (AND) B

HALF ADDER CIRCUIT DIAGRAM



$$S = A \oplus B$$
$$C = A \cdot B$$

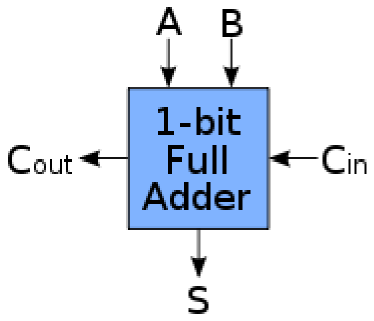
Following is the logic table for a half adder:

A	*B*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

FULL ADDER

- A full adder is capable of adding three bits: two bits and one carry bit. It has three inputs - *A*, *B*, and carry *C*, such that multiple full adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled C_i or C_{in} while the carry out is labelled C_o or C_{out}.

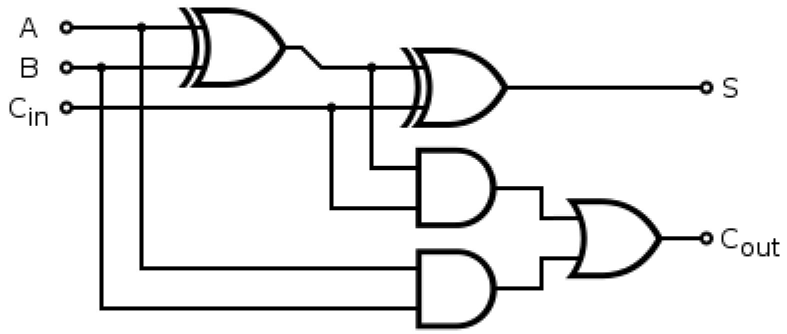
Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}



$$S = (A \oplus B) \oplus C_{in}$$

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

FULL ADDER CIRCUIT DIAGRAM

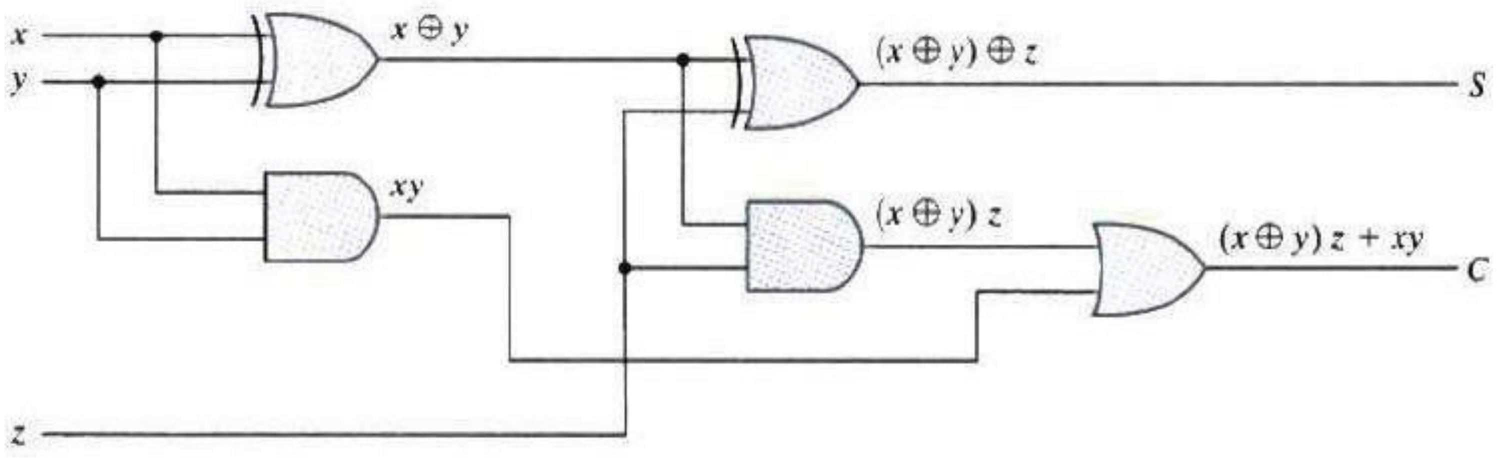


Input			Output	
A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

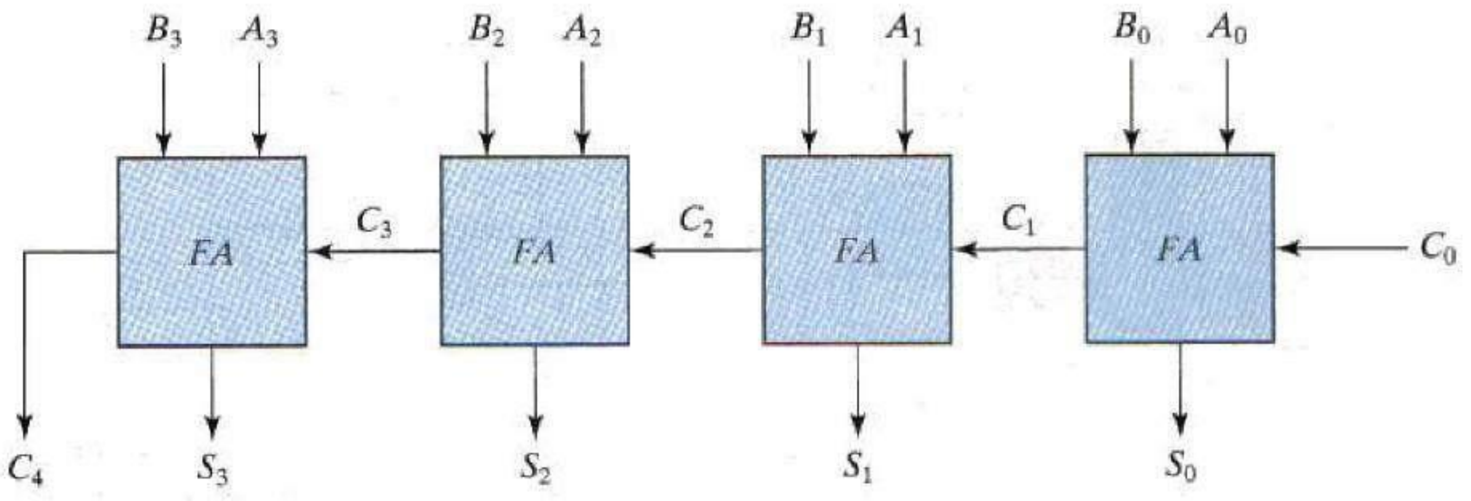
The expression for the sum and the output carry can be obtained by drawing the K map.

FULL ADDER USING HALF ADDER

- A full adder can be constructed from two half adders by connecting X and Y to the input of one half adder, connecting the sum from that to an input to the second adder, connecting Z (C_i) to the other input and OR the two carry outputs. Equivalently, *S* could be made the three-bit XOR of *X*, *Y*, and Z (C_i,) and C_o could be made the three-bit majority function of *X*, *Y*, and Z (C_i.)



MULTIPLE BIT ADDER
(4bit adder using fulladder)



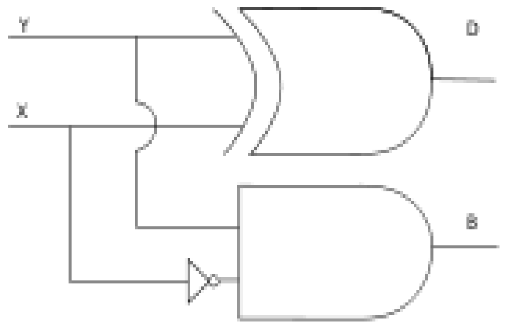
Here two binary numbers

A_3 A_2 A_1 A_0 and B_3 B_2 B_1 B_0 are added to get S_3 S_2 S_1 S_0 with a final carry $C_{4..}$

BINARY SUBTRACTION

HALF SUBTRACTOR

- The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow).

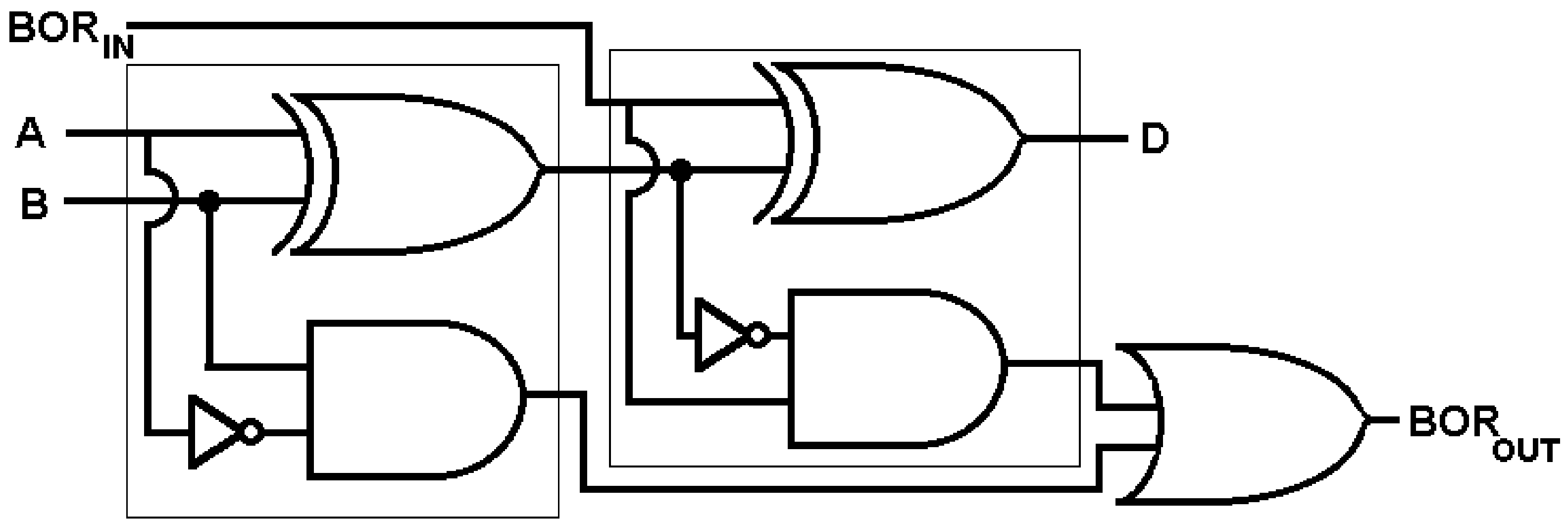


The truth table for the half subtractor is given below.

X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

From the above table one can draw the Karnaugh map or "difference" and "borrow".

FULL SUBTRACTOR



BINARY MULTIPLIER

- A binary multiplier is a electronic circuit used in digital electronics to multiply two binary numbers. It is built using binary adders.

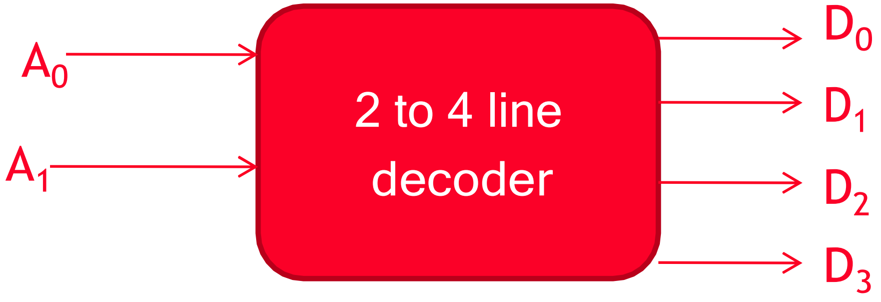
  The partial products can be trivially computed from the fact that $a_i \times b_j = a_i$ AND b_j. The complexity of the multiplier is in adding the partial products.

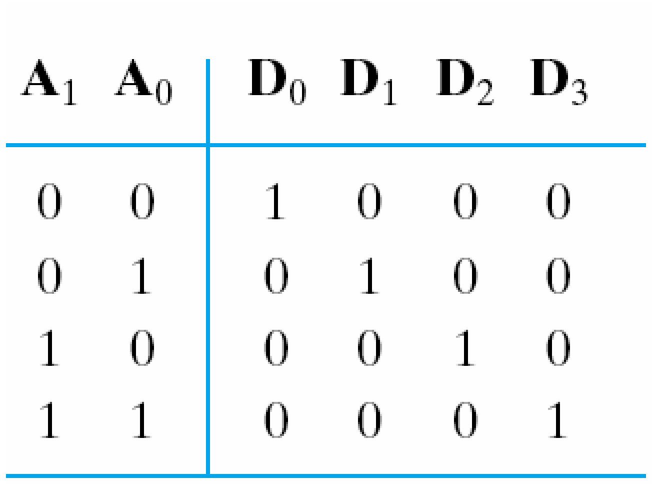
				A_3	a_2	a_1	a_0
			$\times$	B_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	A_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	A_1b_3	a_0b_3			
p_7	p_6	p_5	P_4	p_3	p_2	p_1	p_0

DECODERS

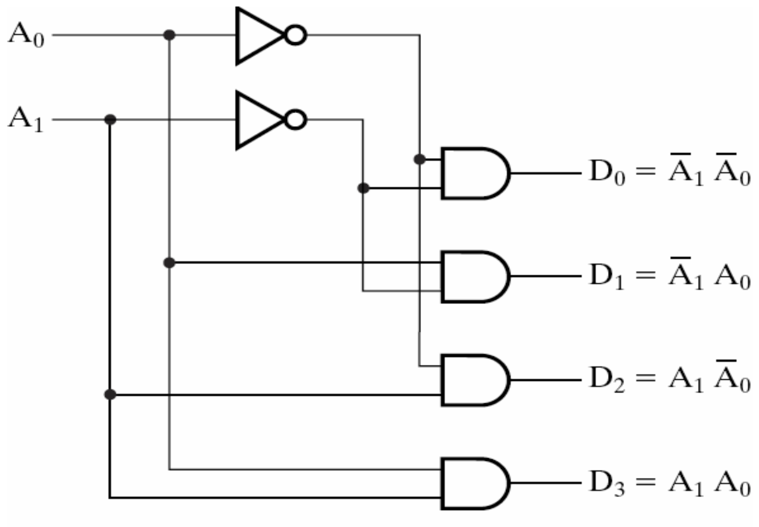
- Decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.
- If the n-bit coded information has unused combinations then decoder may have fewer than 2^n outputs.
- The decoders present here are called n-to-m line decoders, where m is less than equal to 2^n .their purpose is to to generate 2^n (or fewer) minterms of n input variables.

2 to 4 LINE DECODER



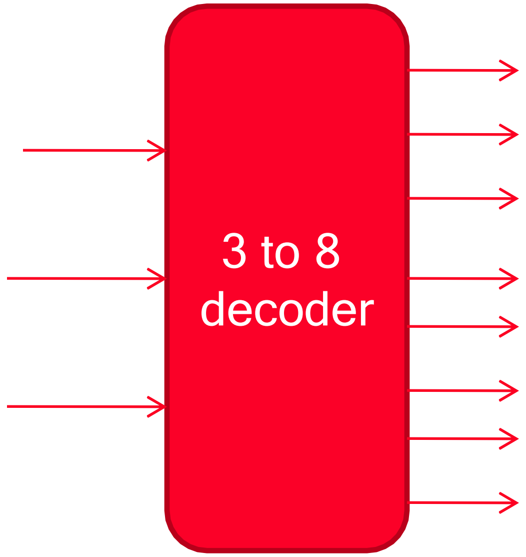


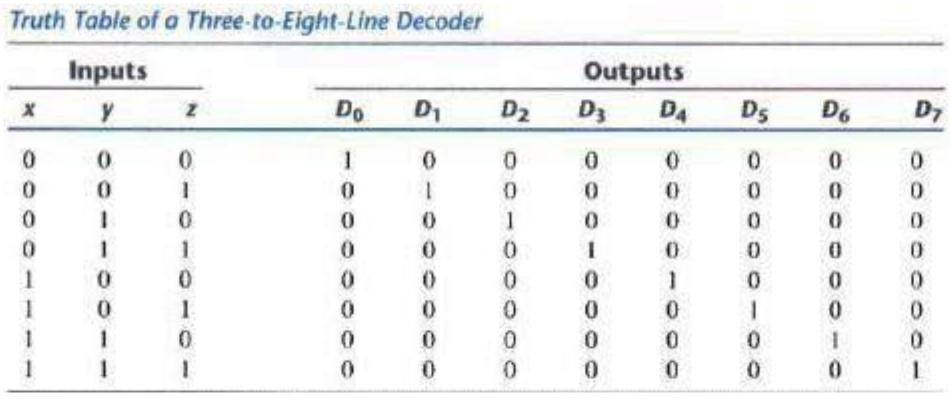
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



3 to 4 LINE DECODER

- It has three inputs and eight outputs.



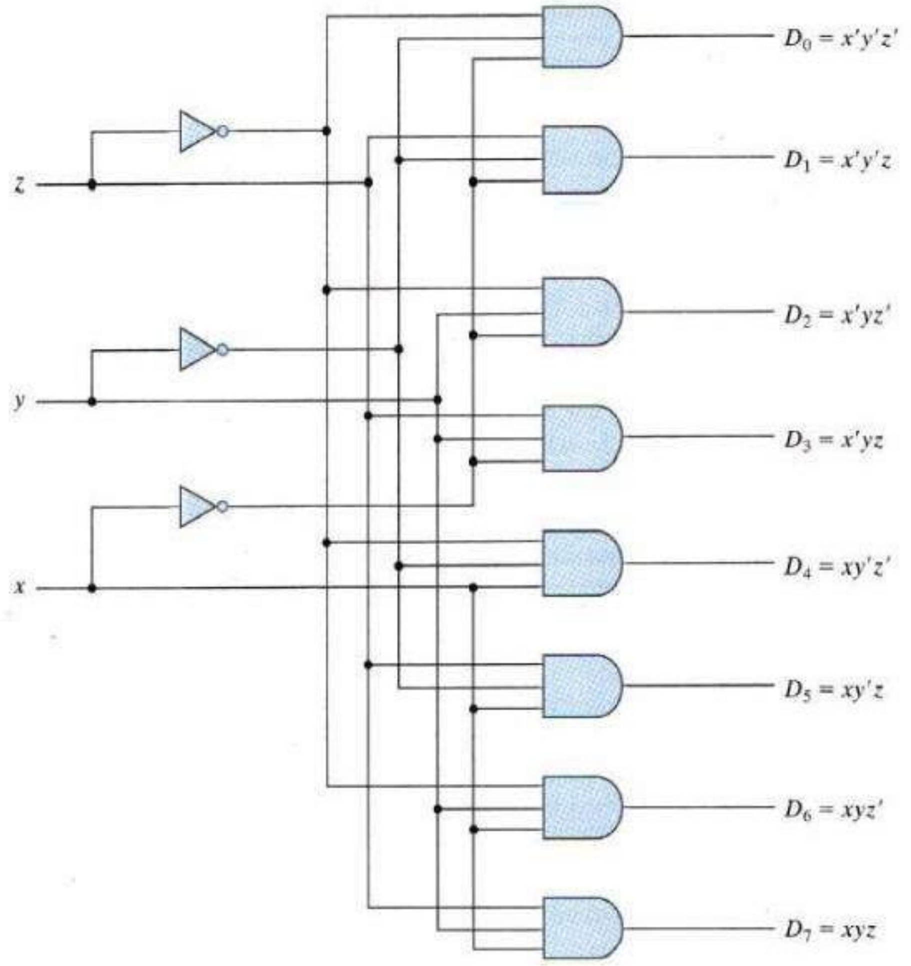


Truth Table of a Three-to-Eight-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Contd…..

(implementation of 3-8 decoder using logic gates)



ENCODER

•An encoder is a digital circuit that performs the inverse operation of a decoder.

•An encoder has 2^n (or fewer) input lines and n output lines.

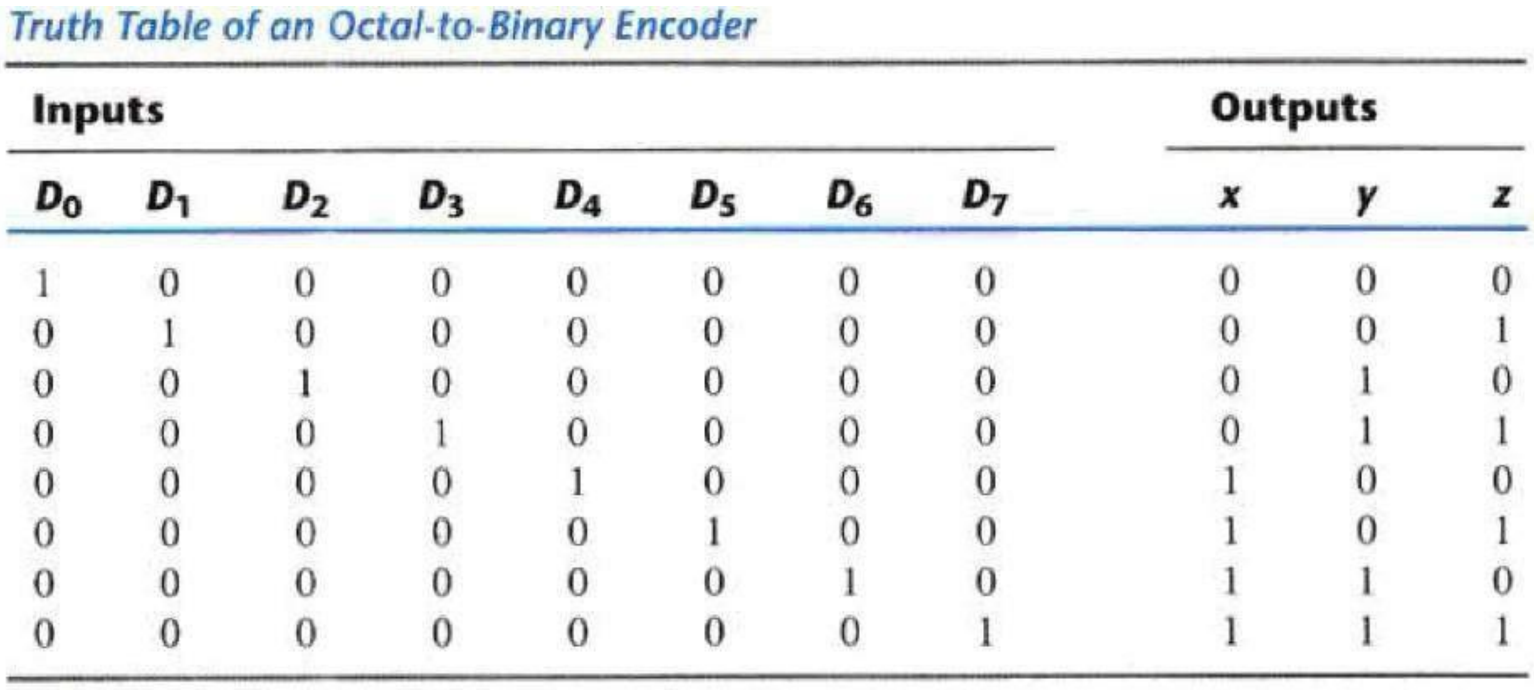
•Example:

1. octal to binary encoder (8 to 3 binary encoder)
2. priority encoder

8 TO 3 BINARY ENCODER
(octal to binary encoder)

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

Here there are 8 inputs (D_o to D_7) and 3 outputs (X , Y, Z).



Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

AMBIGUITY IN ENCODER

There are two ambiguities associated with the design of a simple encoder:

- 1. Only one input can be active at any given time. If two inputs are active simultaneously, the output

  produces an undefined combination

  for example,

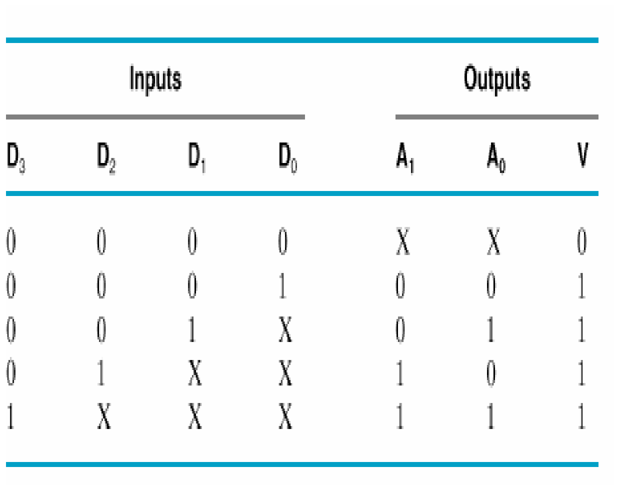
  If D3 and D6 are 1 simultaneously, the output of the encoder will be 111.To avoid this we go for priority encoder

- 2. An output with all 0's can be generated when all the

  inputs are 0's,or when D0 is equal to 1.

PRIORITY ENCODER

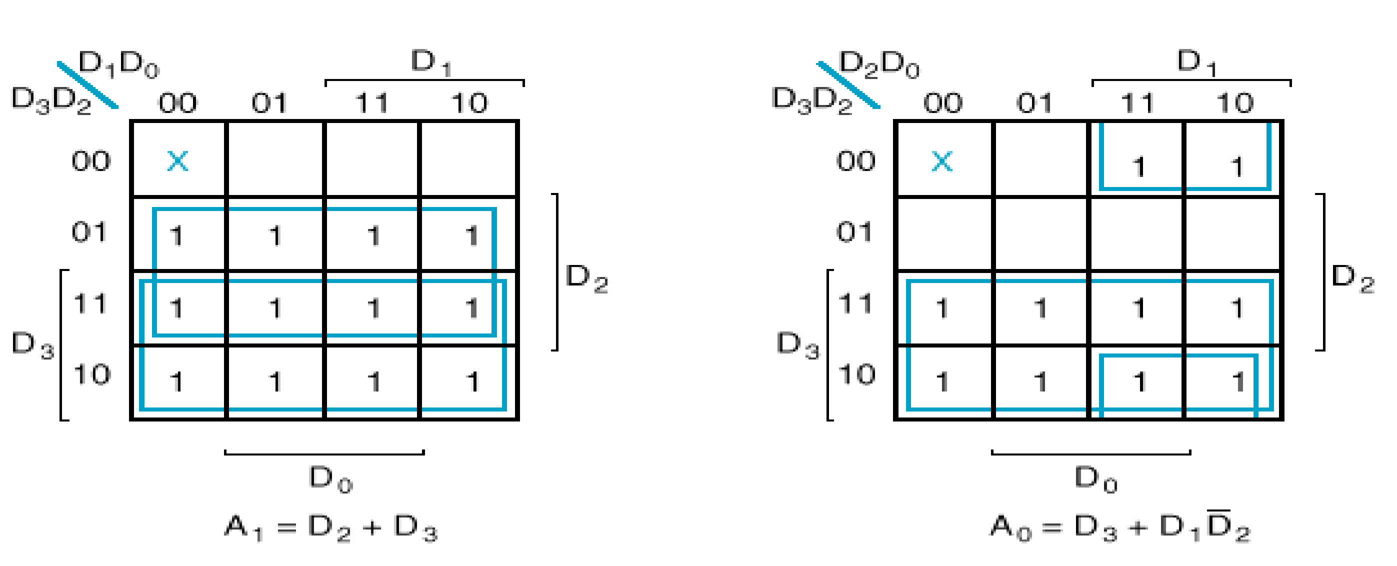
- Apriority encoder is an enccder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time,then the input having the highest priority will take precedence.



Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

In addition to the two outputs x and y , the circuit has a third output designated by V ; this is a valid bit indicator that is set to one when one or more inputs are equal to one. If all th e inputs are 0 there is no valid input hence V is equal to 0.

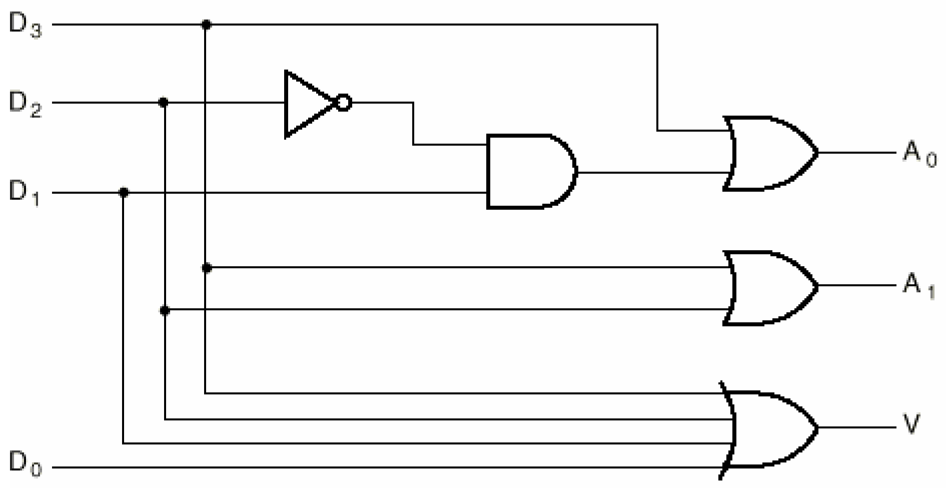
K-map of 4 to 2 PRIORITY ENCODER



$A_1 = D_2 + D_3$

$A_0 = D_3 + D_1\overline{D}_2$

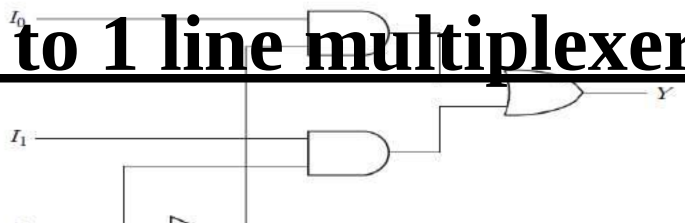
LOGIC DIAGRAM



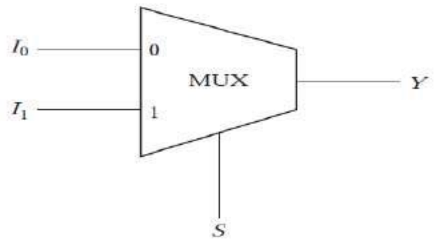
MULTIPLEXER

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines.
- Normally there are 2^n input lines and n selection lines whose bit combinations determine which input is to be selected

2 to 1 line multiplexer:

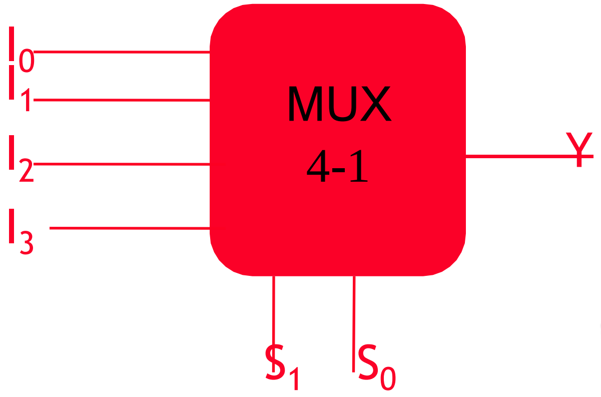


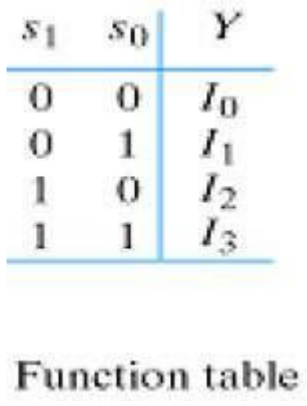
(a) Logic diagram



(b) Block diagram

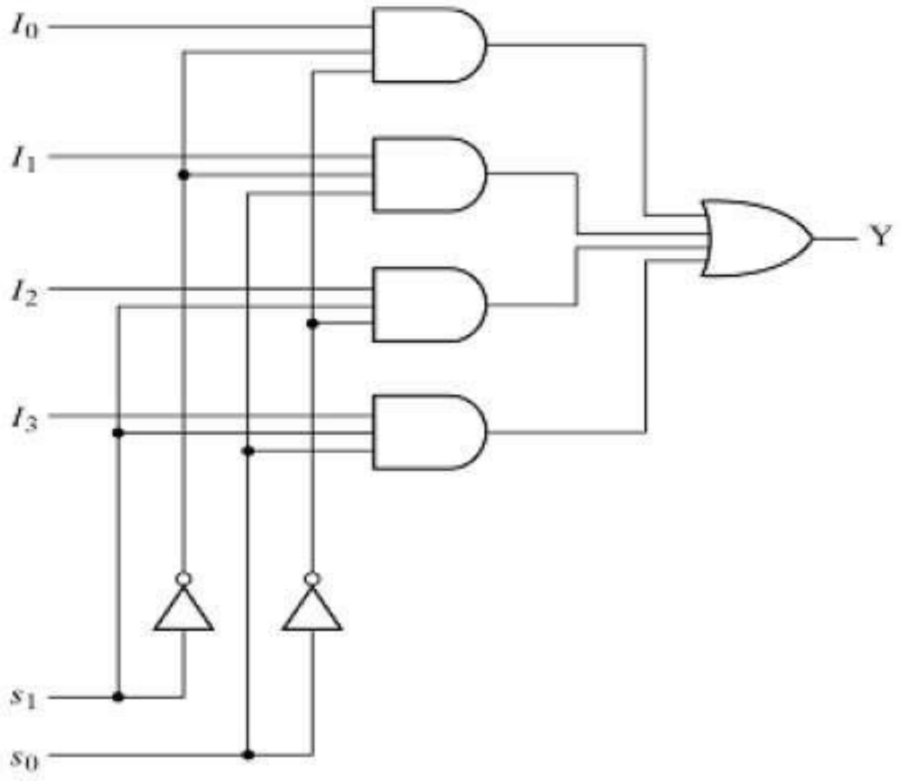
4 TO 1 MULTIPLEXER





s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

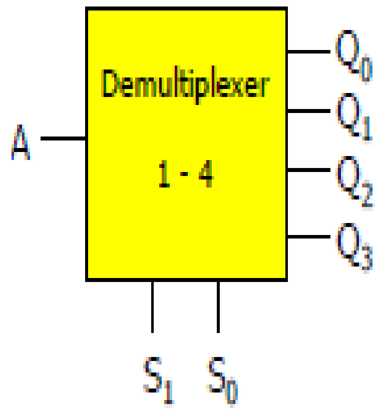
Function table

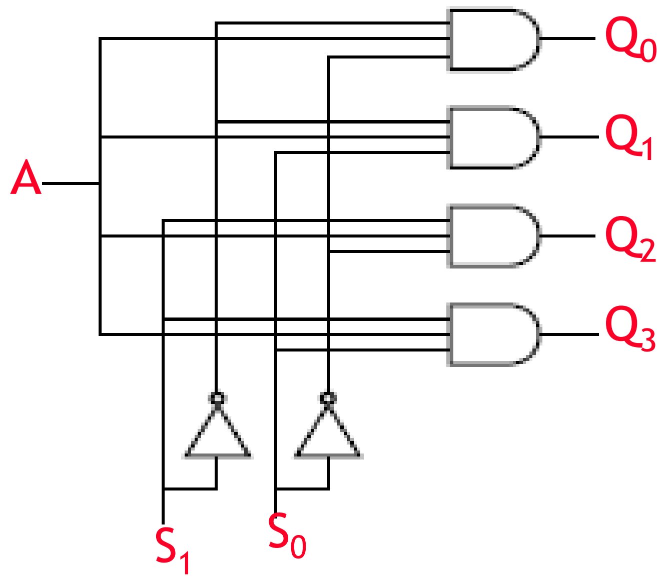


DEMULTIPLEXER

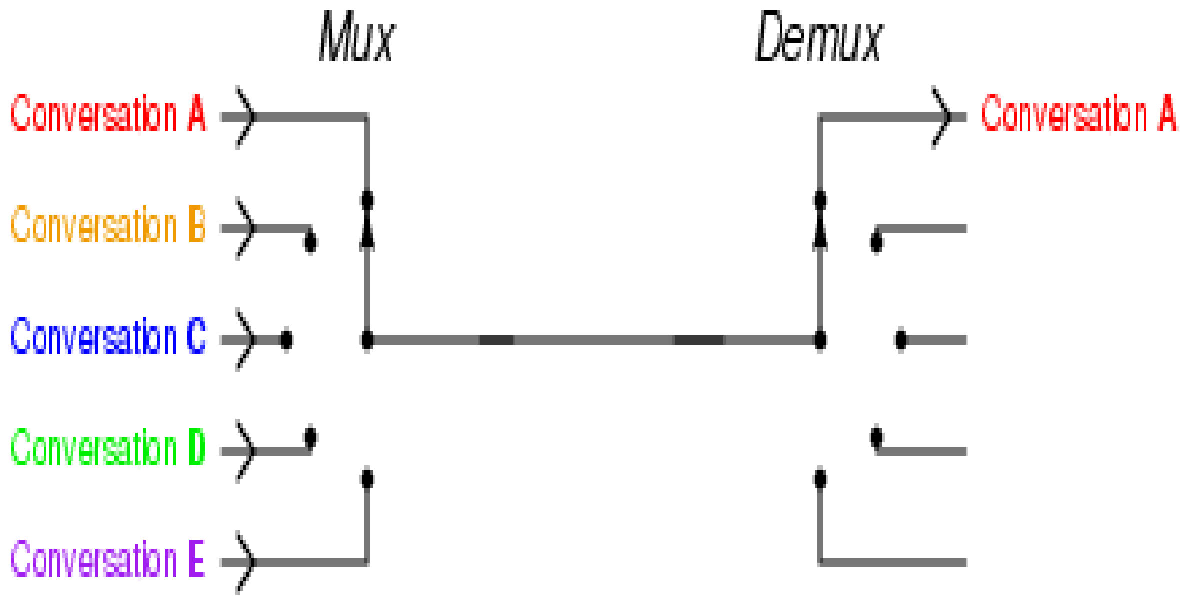
- Decoder with an enable line is called demultiplexer.
- It has a single input and the output is obtained as per the selection lines.
- There are 2^n no of outputs having n selection lines.

1 : 4 demultiplexer





Overall operation of MULTIPLEXER AND DEMULTIPLEXER



MAGNITUDE COMPARATOR

- the comparison of two numbers is an operation that determines whether one nubmer is greater than, less than or equal to the other number i.e. if A and B are two numbers then this circuit determines whether A>B or A=B or A<B.

Suppose

$A = A_3\, A_2\, A_1\, A_0$

$B = B_3\, B_2\, B_1\, B_0$

$x_i = A_iB_i + A_iB_i \qquad for \quad i = 0,1,2,3$

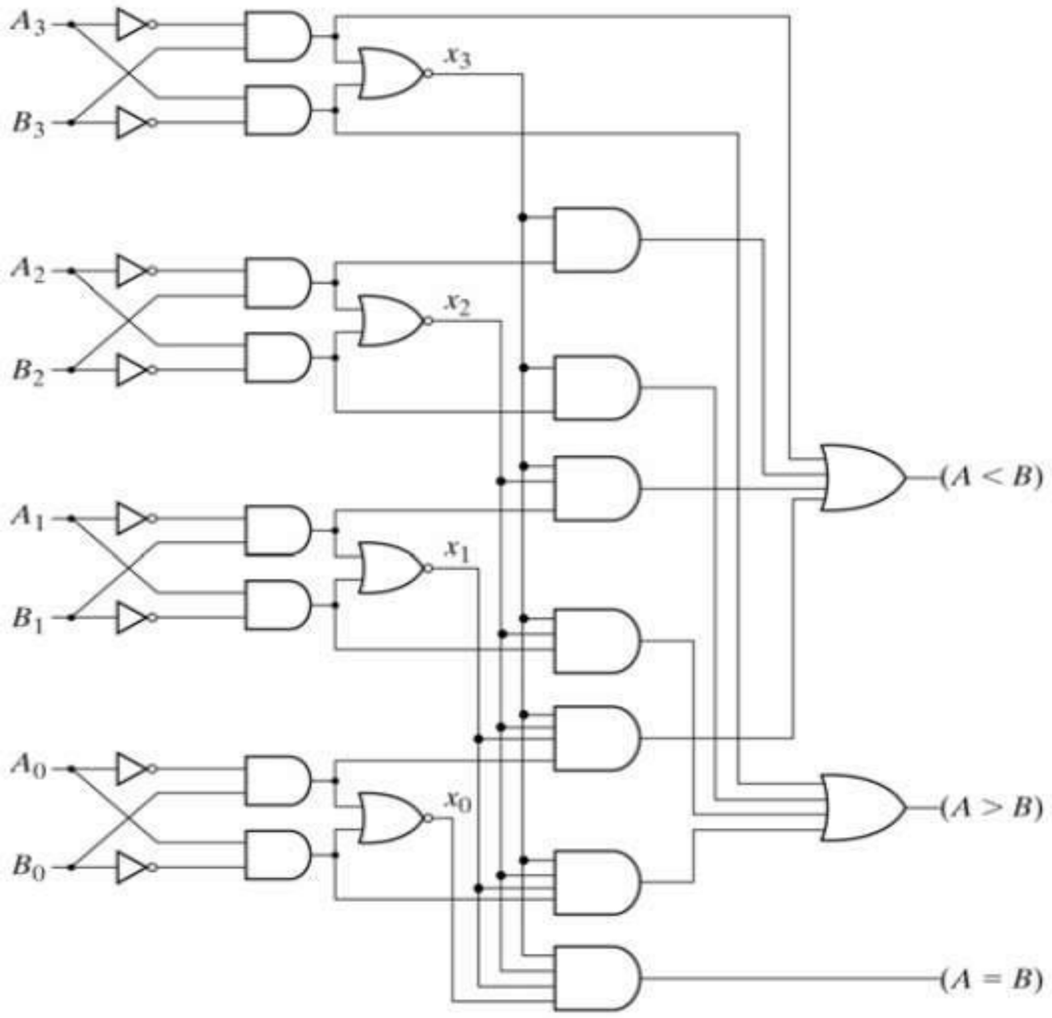
1 only if the pair of bits in i are equal

$$(A = B) = x_3x_2x_1x_0$$

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

LOGIC DIAGRAM OF MAGNITUDE COMPARATOR



MEMORY ELEMENTS

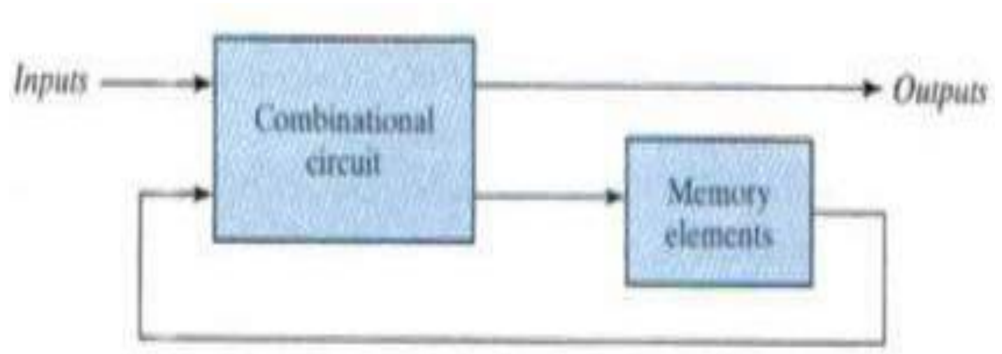
AND

SEQUENTIAL NETWORKS

SEQUENTIAL NETWORK

- Sequential circuit is a combinational circuit along with a memory element which is capable of storing a memory element.

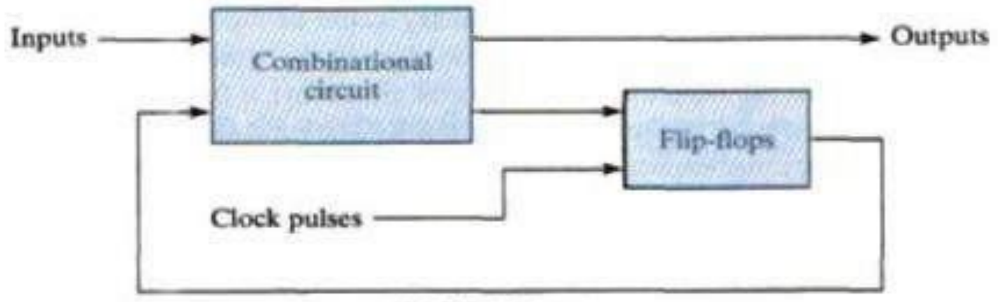
BLOCK DIAGRAM



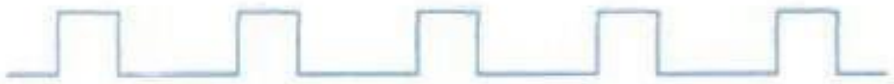
SYNCHRONOUS SEQUENTIAL LOGIC

Synchronous sequential circuits:

This circuit employs signals that affect the storage element only at discrete instant of time.here synchronization is achieved by a timing device called a clock generator,which provides a clock signal having the form of a periodic train of clock pulses.



(a) Block diagram



(b) Timing diagram of clock pulses

STORAGE ELEMENTS:

- A storage element in a digital circuit can store a binary state indefinitely, until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches.
- Storage elements that operate with a clock transition are referred to as flip-flops.

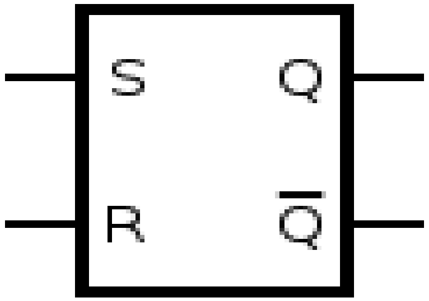
LATCHES

- Latches are bistable device and level sensitive. These are the basic building blocks of flip-flops.

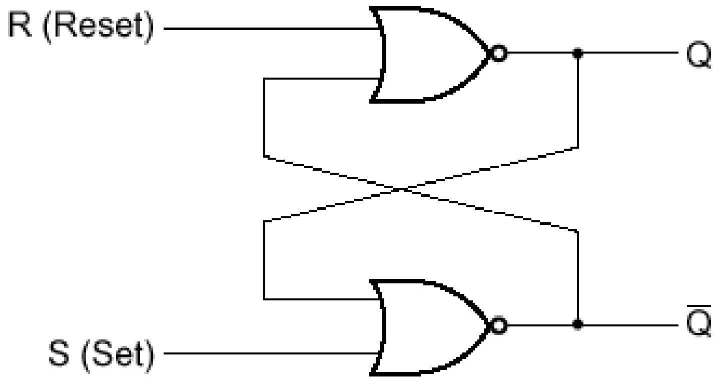
SR Latch

When using static gates as building blocks, the most fundamental latch is the simple *SR latch*, where S and R stand for *set* and *reset*. It can be constructed from a pair of cross-coupled NOR or NAND logic gates. The stored bit is present on the output marked Q

Block diagram of SR Latch



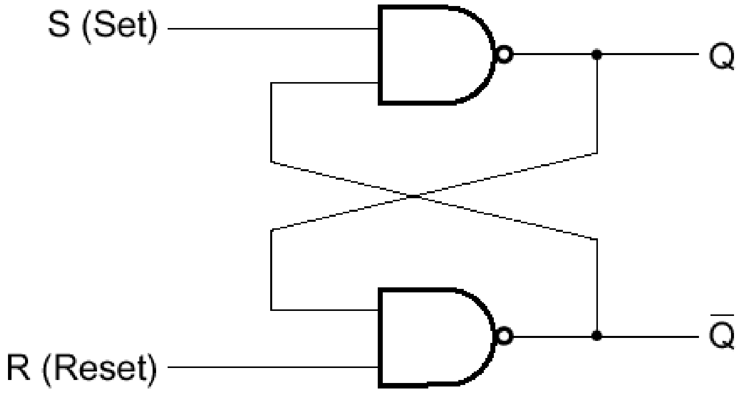
SR LATCH USING NOR AND NAND GATES



(a) Logic diagram

S	R	Q	$\overline{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

(b) Function table

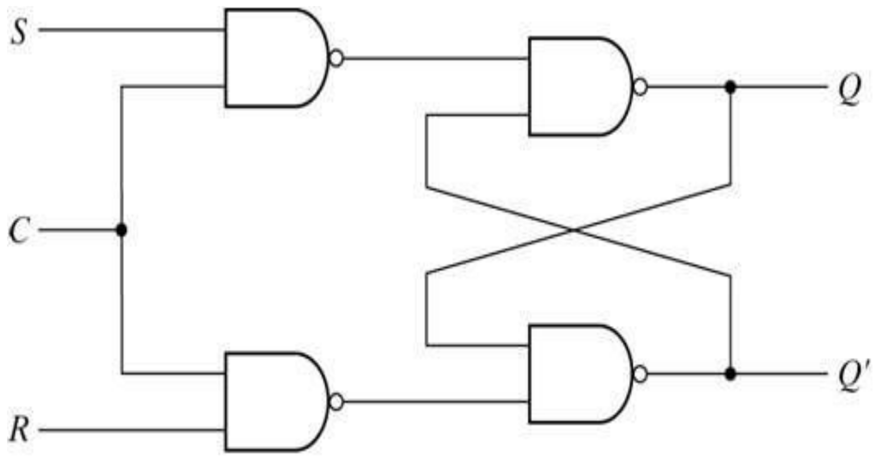


(a) Logic diagram

S	R	Q	$\overline{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

(b) Function table

SR LATCH WITH CONTROL INPUT



(a) Logic diagram

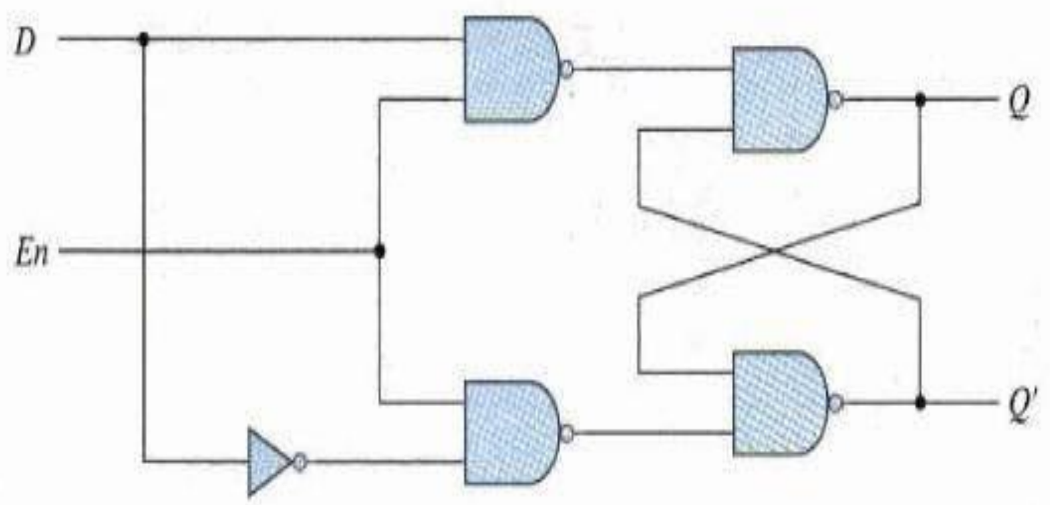
C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; set state
1	1	1	Indeterminate

(b) Function table

- Occasionally, desirable to avoid latch changes
- C= 0 disables all latch state changes
- Control signal enables data change when C = 1
- Right side of circuit same as ordinary S-R latch.

D LATCH

- D latch is also called transparent latch.

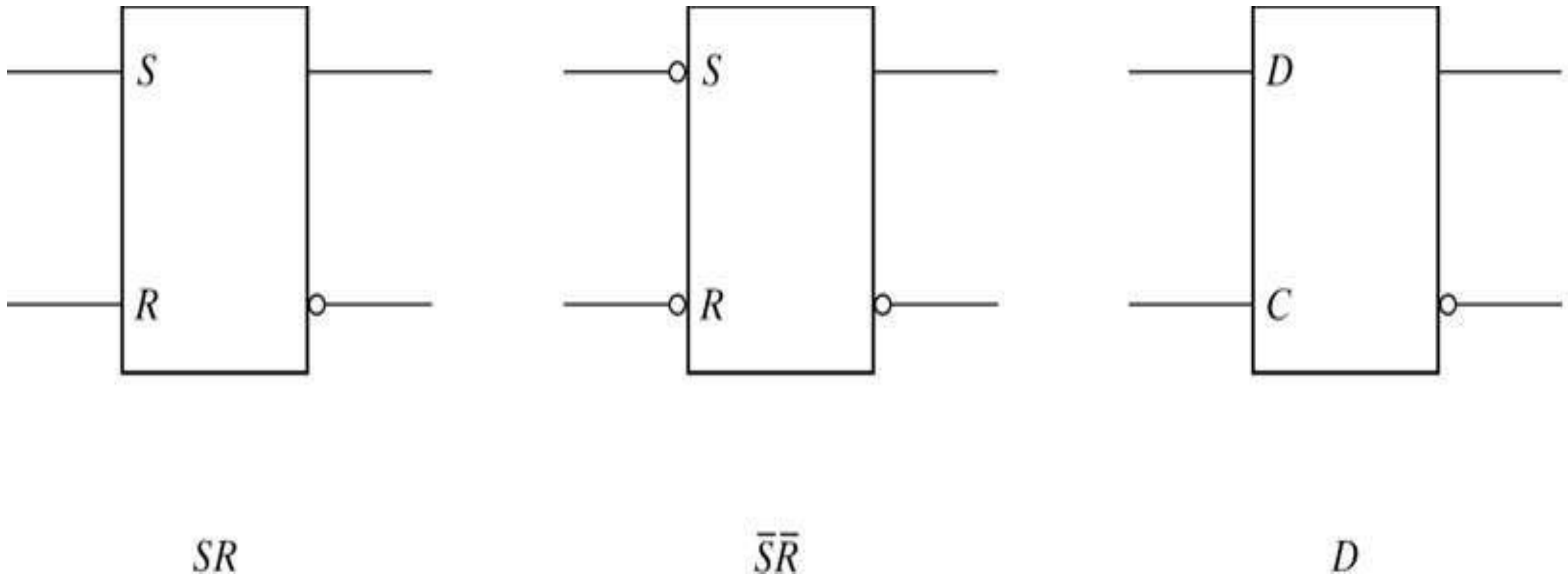


(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

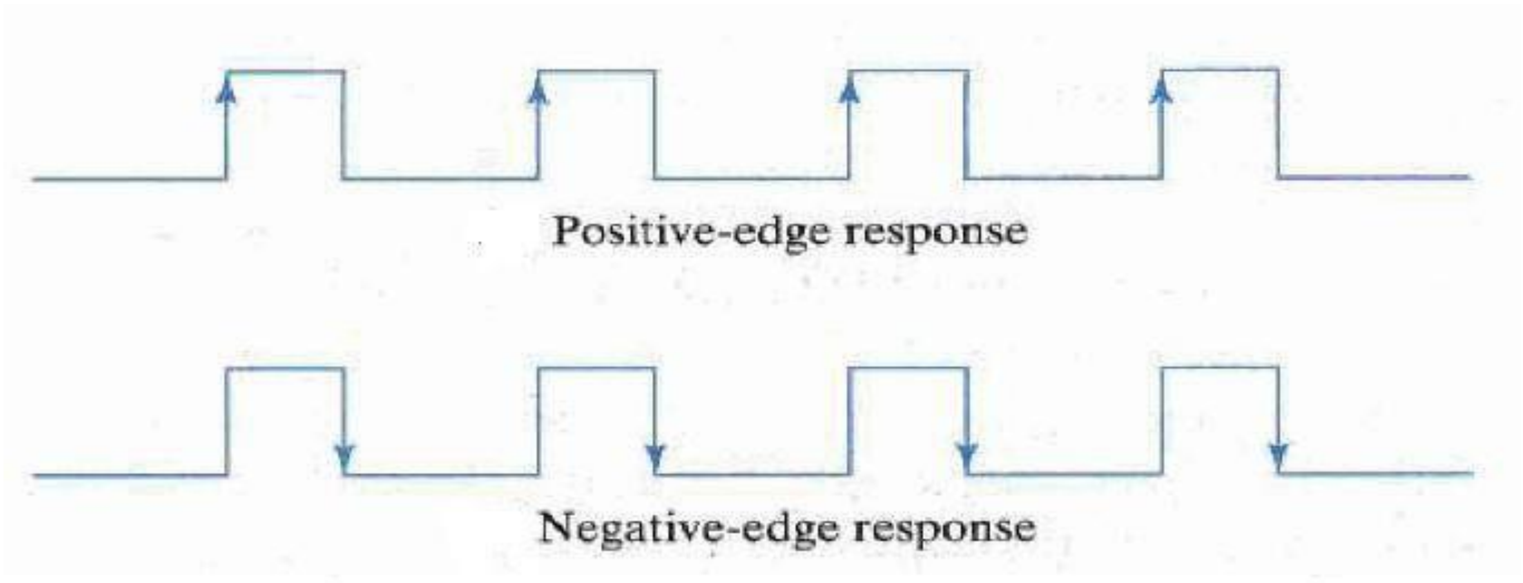
(b) Function table

GRAPHICAL SYMBOL OF LATCHES

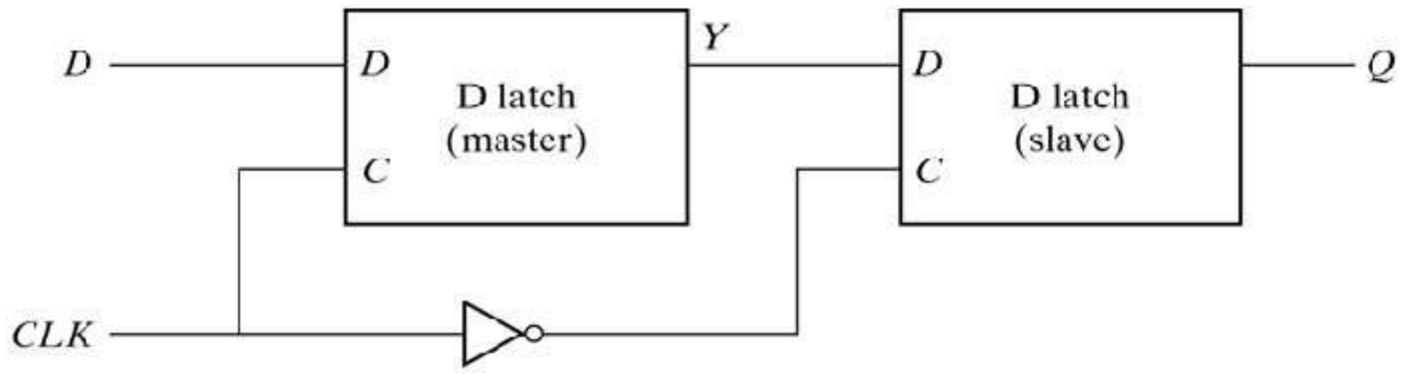


FLIP-FLOPS

- A flip-flops is a device which changes its state at the times when a change is taking place in the clock signal.
- The flip-flops are generally edge triggered i.e. they are either +ve edge edge triggered or –ve edge triggered.



MASTER SLAVE D FLIP-FLOP

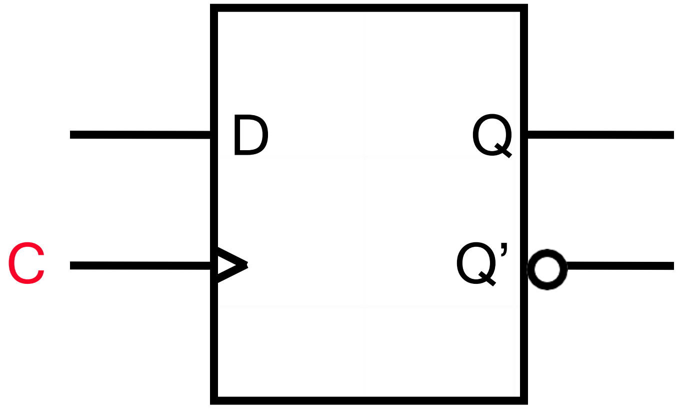


- Consider two latches combined together
- Only one *Clk* value is active at a time
- Output changes of falling edge of the clock

D FLIP FLOP

- Stores a value on the positive edge of *C*
- Input changes at other times have no effect on output

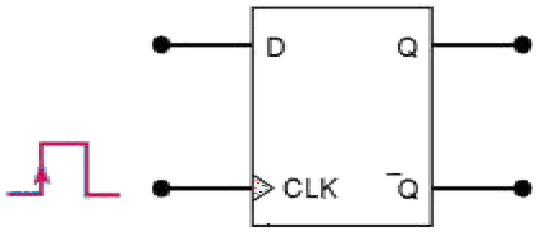
Positive edge triggered

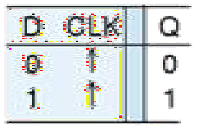


D	C	Q	Q'
0	$\uparrow$	0	1
1	$\uparrow$	1	0
X	0	Q_0	Q_0'

CLOCKED D FLIP-FLOP

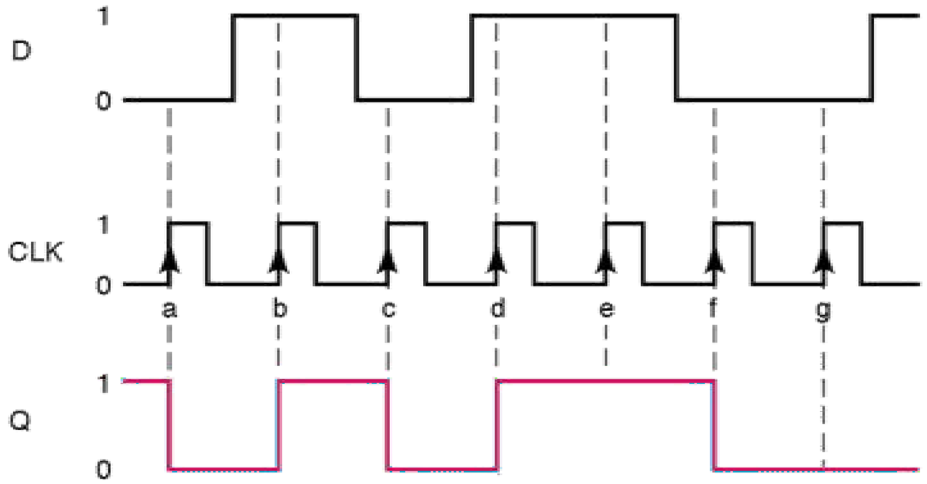
- Stores a value on the positive edge of *C*
- Input changes at other times have no effect on output





D	CLK	Q
0	↑	0
1	↑	1

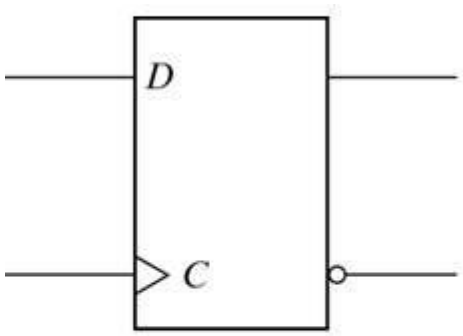
(a)



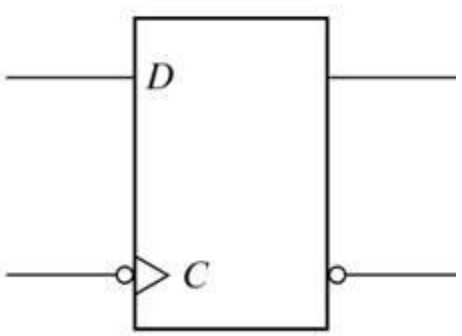
(b)

+VE AND –VE EDGE TRIGGERED D FLIP-FLOP

- D flops can be triggered on positive or negative edge
- Bubble before Clock (C) input indicates negative edge trigger

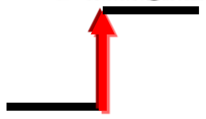


(a) Positive-edge

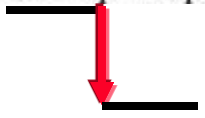


(a) Negative-edge

Graphic Symbol for Edge-Triggered *D* Flip-Flop

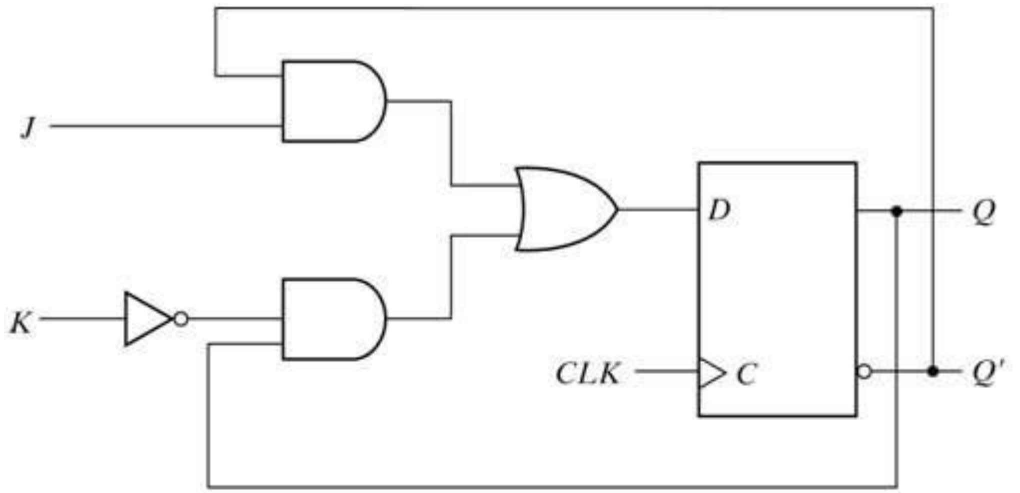


Lo-Hi edge

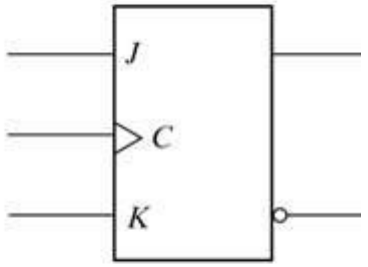


Hi-Lo edge

POSITIVE EDGE-TRIGGERED J-K FLIP-FLOP



(a) Circuit diagram



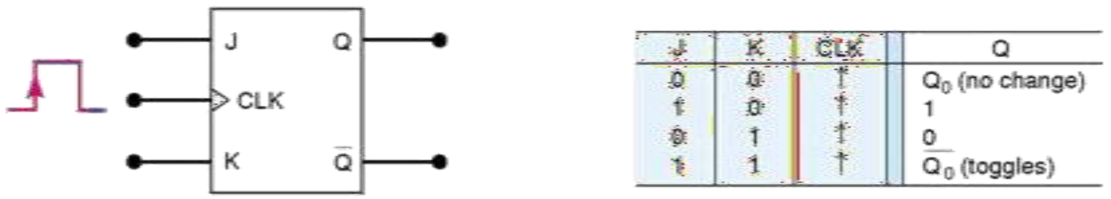
(b) Graphic symbol

JK Flip-Flop

JK Flip-Flop

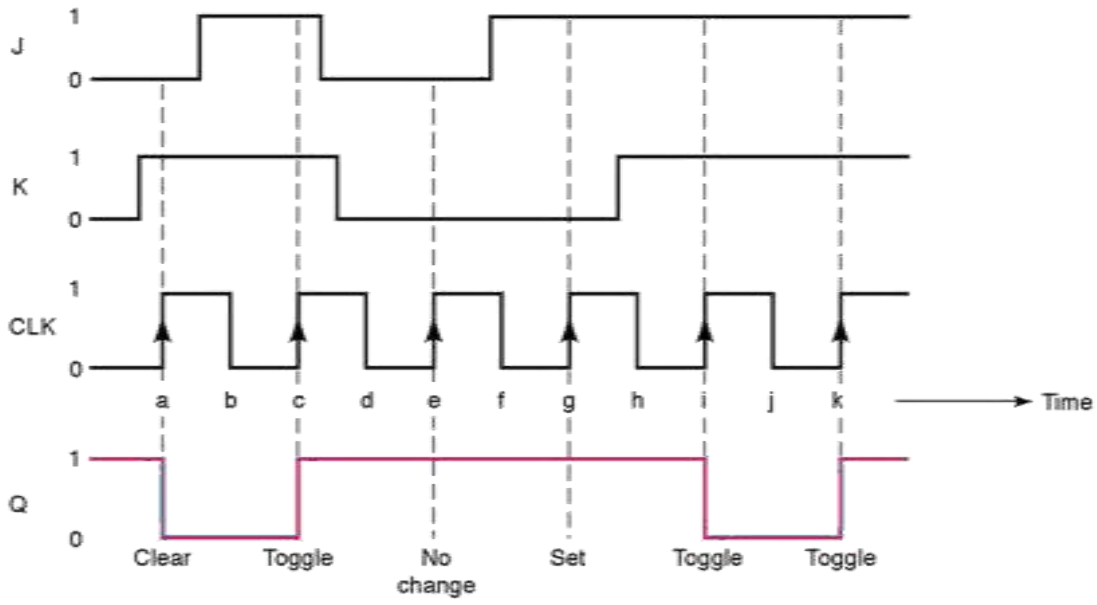
J	***K***	**Q(*t* + 1)**	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

CLOCKED J-K FLIP FLOP



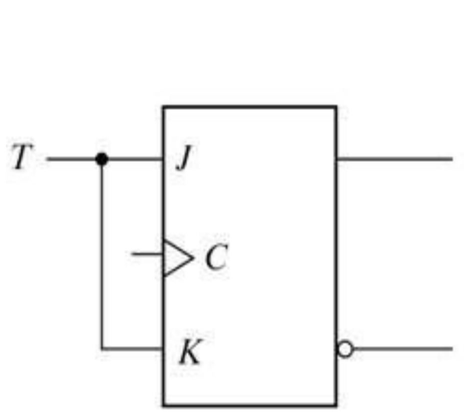
J	K	CLK	Q
0	0	↑	Q_0 (no change)
1	0	↑	1
0	1	↑	0
1	1	↑	$\overline{Q}_0$ (toggles)

(a)

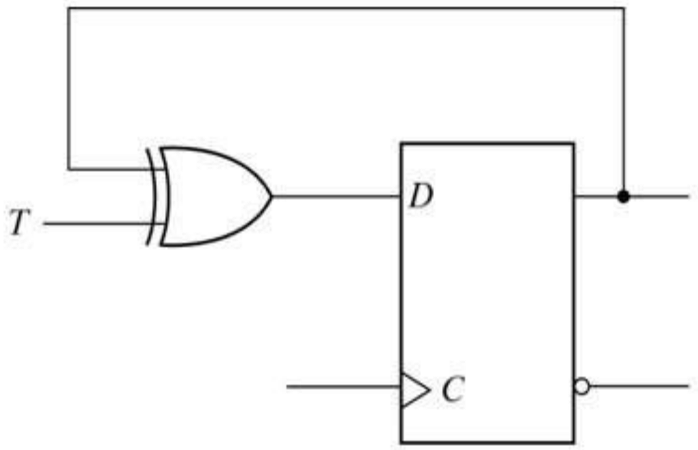


(b)

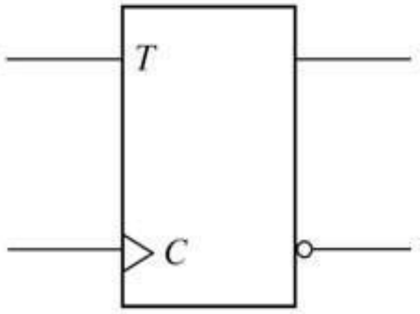
POSITIVE EDGE-TRIGGERED T FLIP-FLOP



(a) From *JK* flip-flop

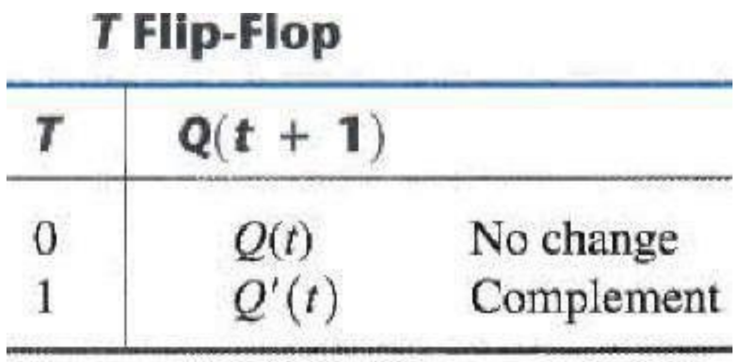


(b) From *D* flip-flop



(c) Graphic symbol

T Flip-Flop



***T* Flip-Flop**

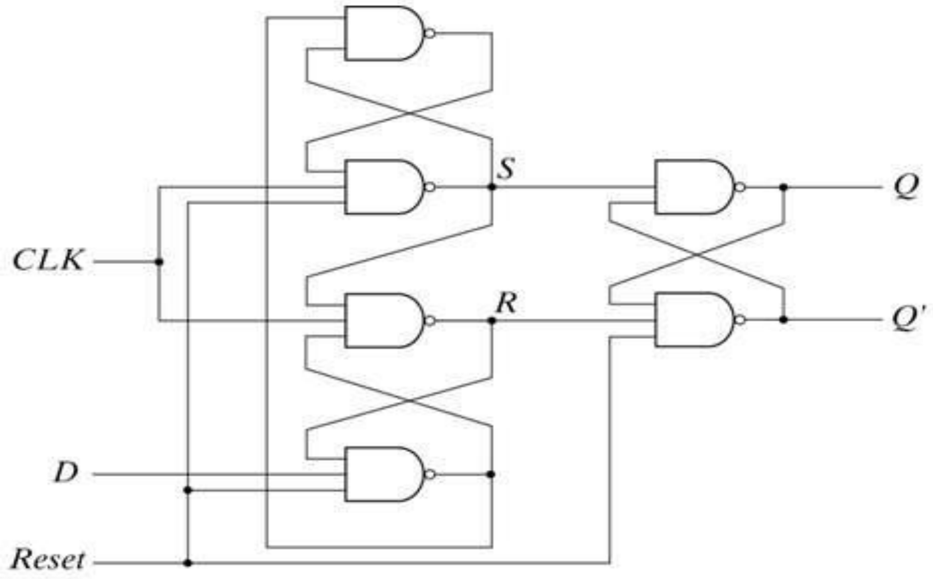
T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

ASYNCHRONOUS INPUTS

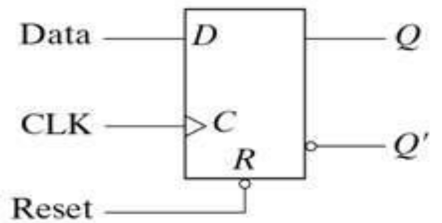
- Some of the flip flops have asynchronous inputs that are used to force the flip flop to a particular state independent of the clock.
- The input that sets the flip flop to 1 is called PRESET or DIRECT SET.
- When a power is turned on in a digital system, the state of the flip flop is unknown. the direct inputs are useful for bringing the flip flop to a known starting state prior to the clocked operation.
- Similarly there is a RESET that sets the flip flop to 0.

EXAMPLE :

Positive edge triggered d flip-flop with active low asynchronous reset is shown in the figure below.



Circuit diagram



Graphic symbol

R	C	D	Q	Q'
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

Function table

REGISTERS AND COUNTERS

REGISTERS

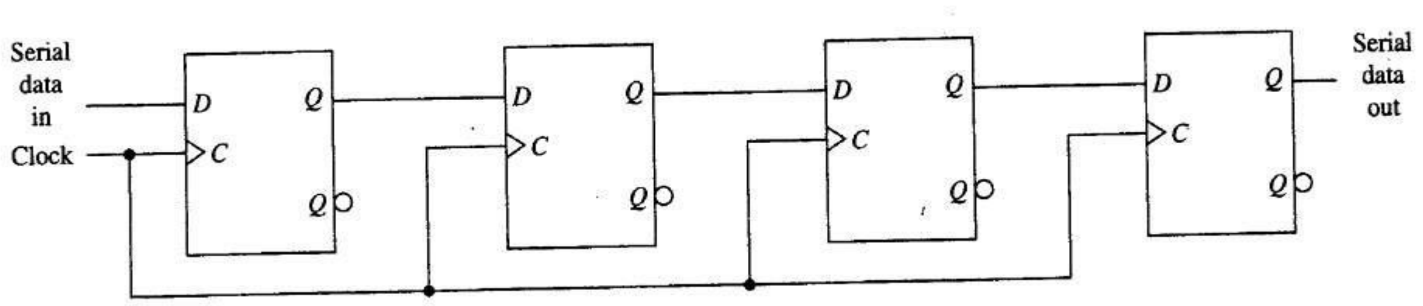
- Register is a group of flip-flops which stores several bits of binary data.
- A n-bit register has n flip-flops
- It can hold n bits of binary data
- Register may also contain combinational logic

Registers are classified as

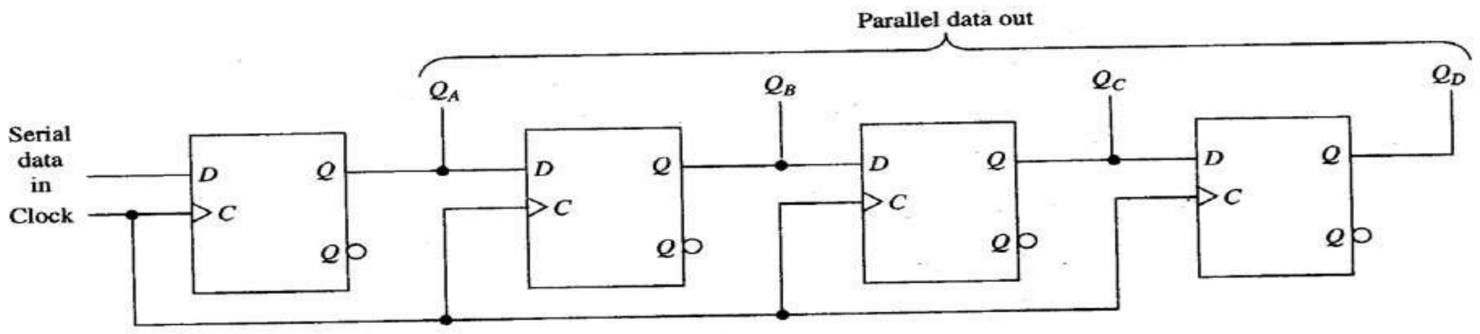
1. Serial in serial out 2.parallel in serial out

3. Serial in parallel out 4.parallel in parallel out

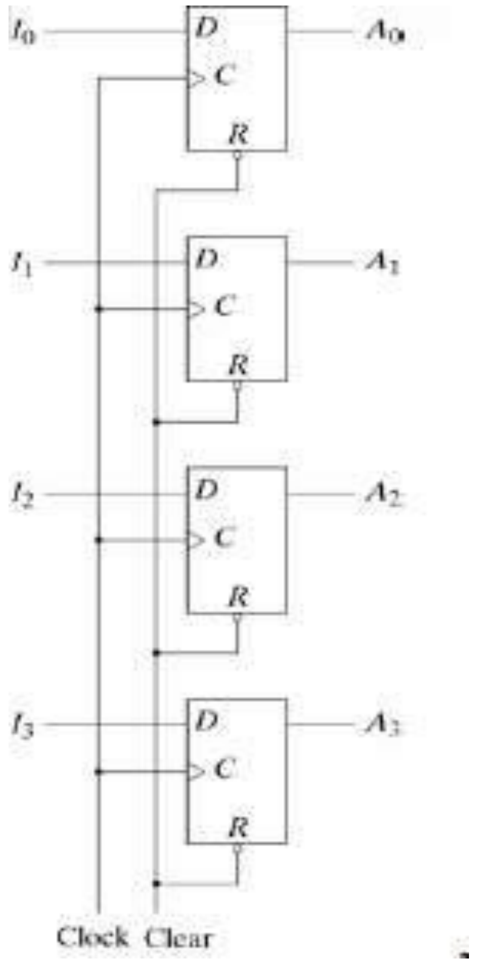
Serial-In, Serial-Out Unidirectional Shift Register



Serial-In, Parallel-Out Unidirectional Shift Register

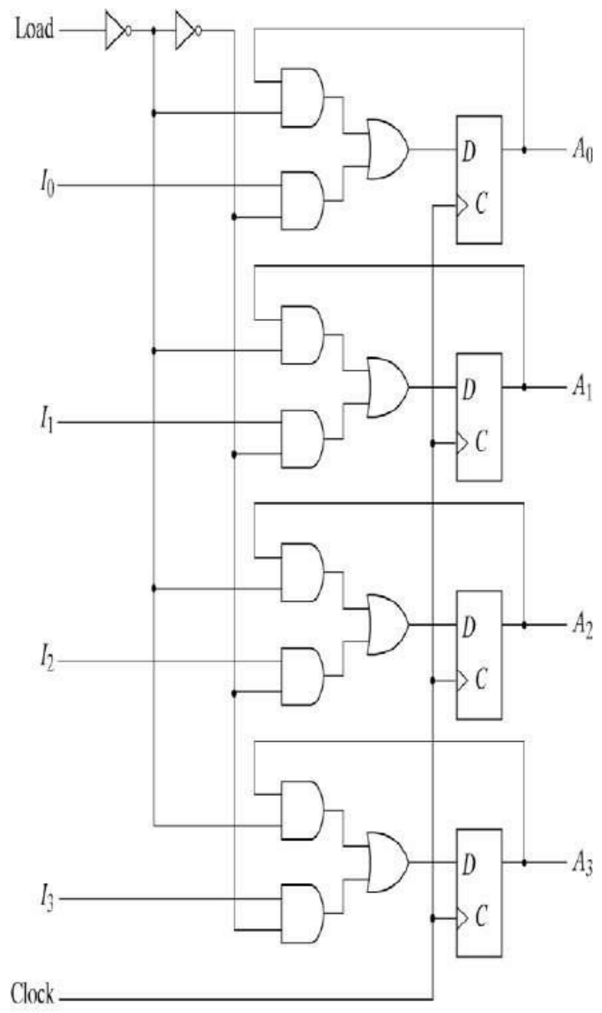


PARALLEL IN PARALLEL OUT SHIFT REGISTER



- The common clock input triggers all flip-flops and the binary data available at the four inputs are transferred into the register.
- The clear input is useful for clearing the register to all 0’s output.

REGISTER WITH PARALLEL LOAD

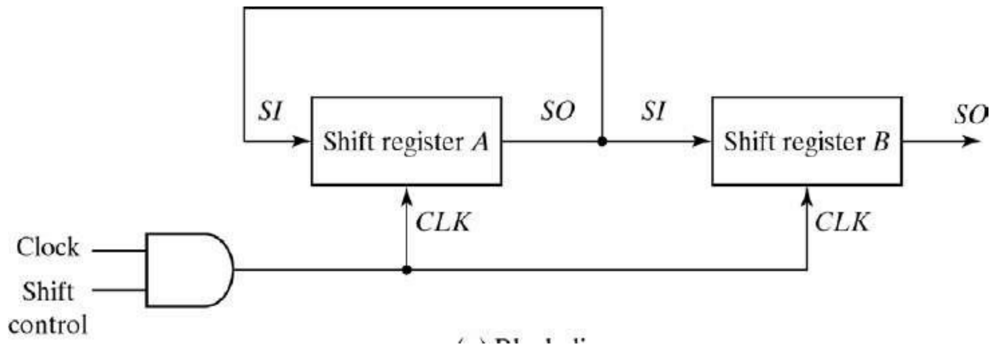


- If all the bits in a register are loaded at the same time, the loading is done in parallel.
- A 4-bit register with a load control input is shown here.
- The Load input determines the action to be taken with each clock pulse.
- The feedback connection

  from output to input is necessary because the D flip-flop does not have a "no change" condition.

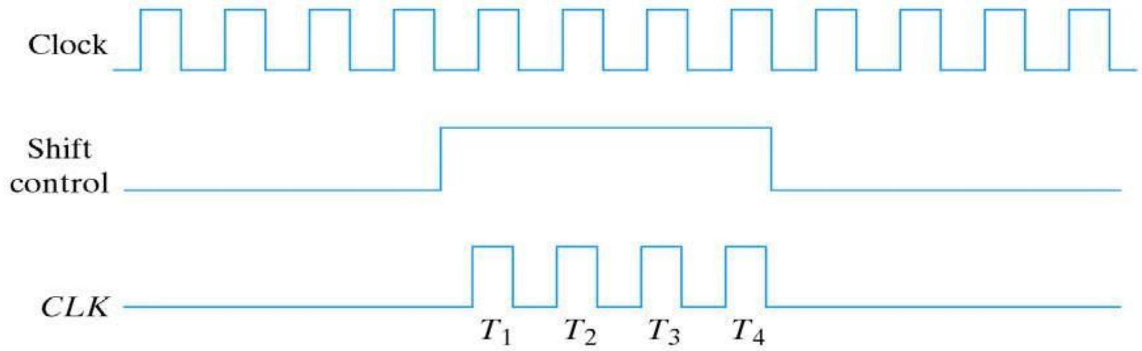
SERIAL TRANSFER

- Serial transfer of information from register A to register B is done with shift registers:

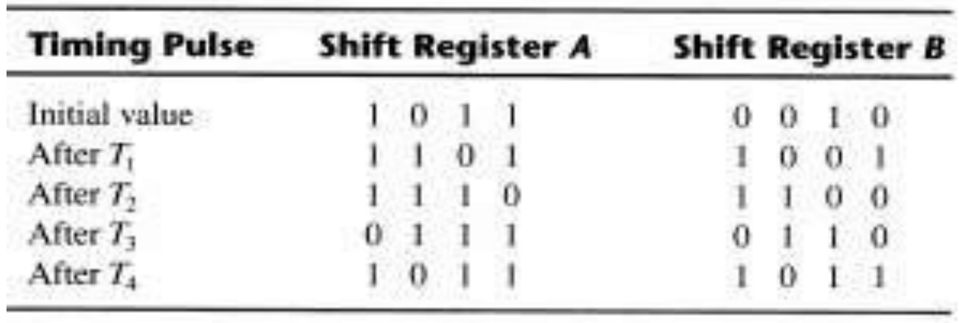


Suppose the shift registers have four bits each. The control unit that supervises the transfer must be designed such that it enables the shift registers, via the shift control signal, for a fixed time of four clock pulses:

Contd..



Assume that the binary content of A before the shift is 1011 and that of B is 0010. The serial transfer occurs in four steps as shown in the table below:



Timing Pulse	Shift Register *A*	Shift Register *B*
Initial value	1 0 1 1	0 0 1 0
After T_1	1 1 0 1	1 0 0 1
After T_2	1 1 1 0	1 1 0 0
After T_3	0 1 1 1	0 1 1 0
After T_4	1 0 1 1	1 0 1 1

COUNTERS

- Counter is a register which counts the sequence in binary form.
- The state of counter changes with application of clock pulse.
- The total no. of states in counter is called as modulus.
- If counter is modulus-n, then it has n different states.\
- Counters are available in two categories:

  ripple counters and synchronous counters.
- State diagram of counter is a pictorial representation of counter states directed by arrows in graph.

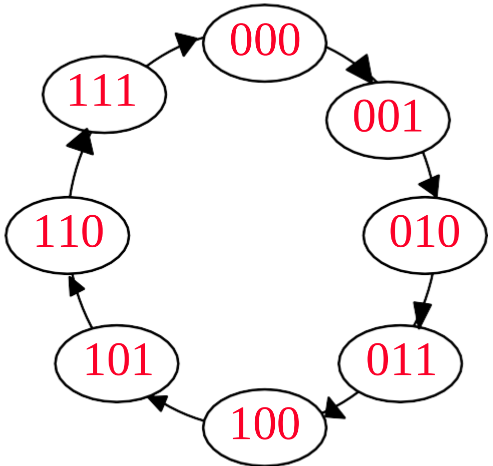
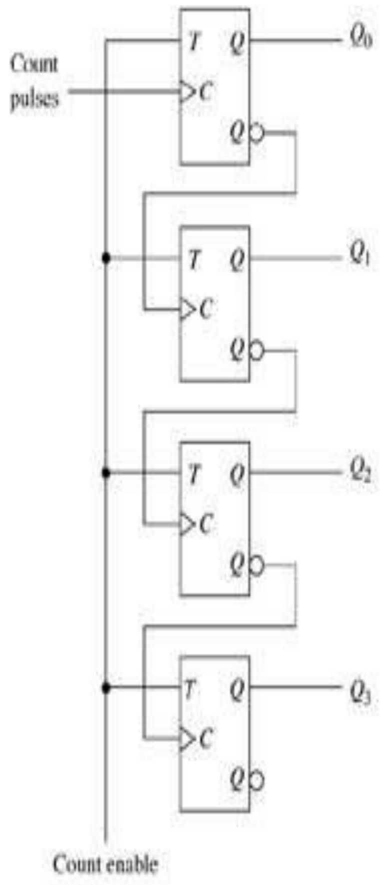
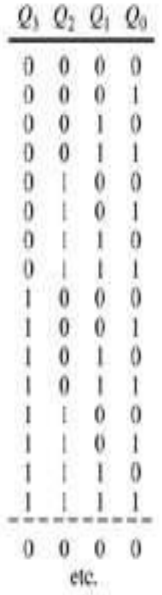


Fig. State diagram of mod-8 counter

RIPPLE (ASYNCHRONOUS COUNTER)

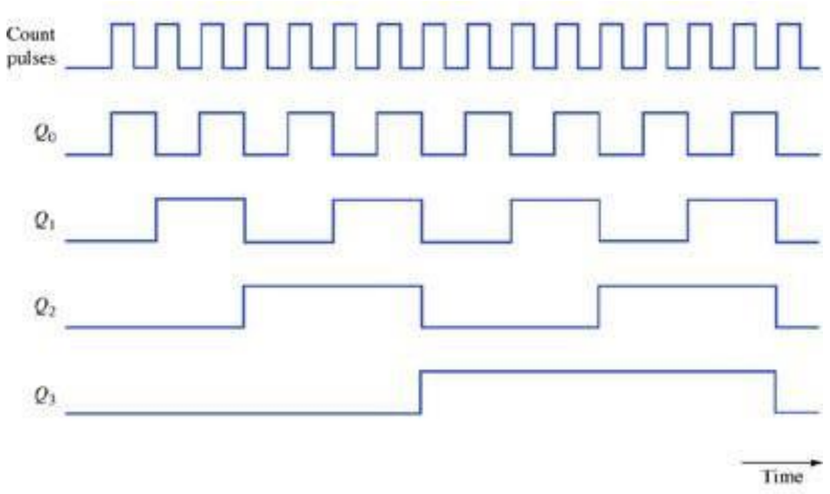




Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1
0	0	0	0

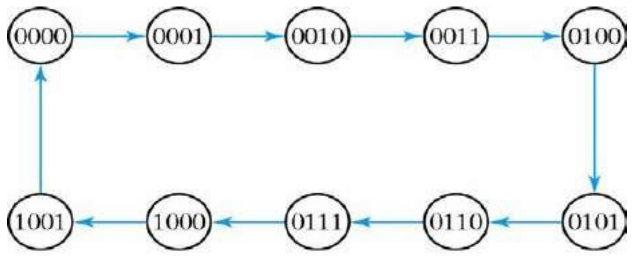
etc.

- In a ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops.
- A 4-bit binary ripple counter (mod-16) is given here.

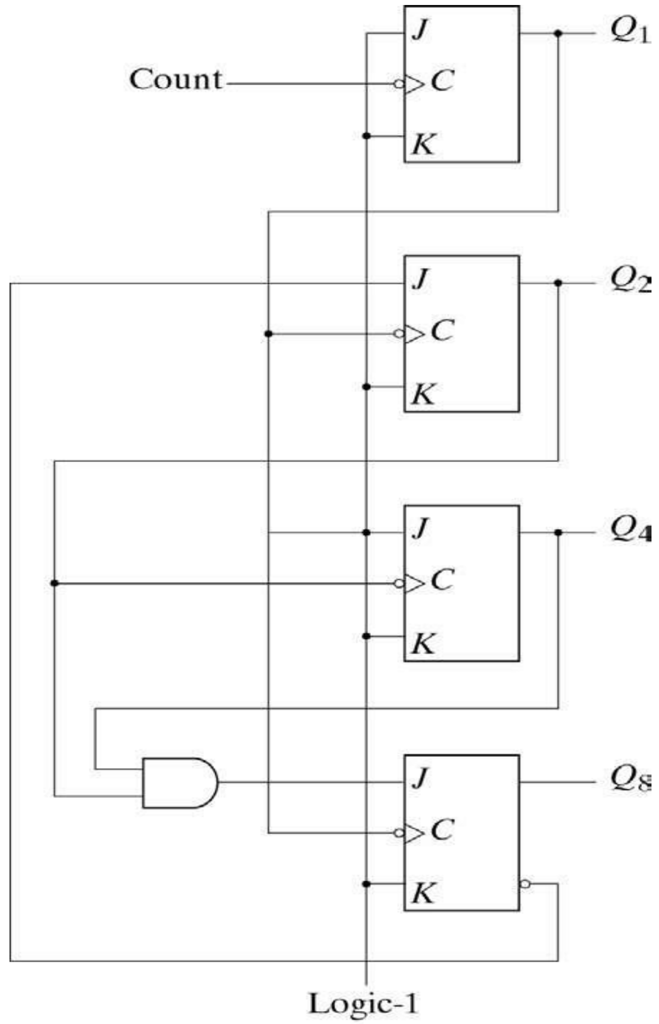


BCD RIPPLE COUNTER (MOD-10)

- A decimal counter follows a pattern of 10 states:

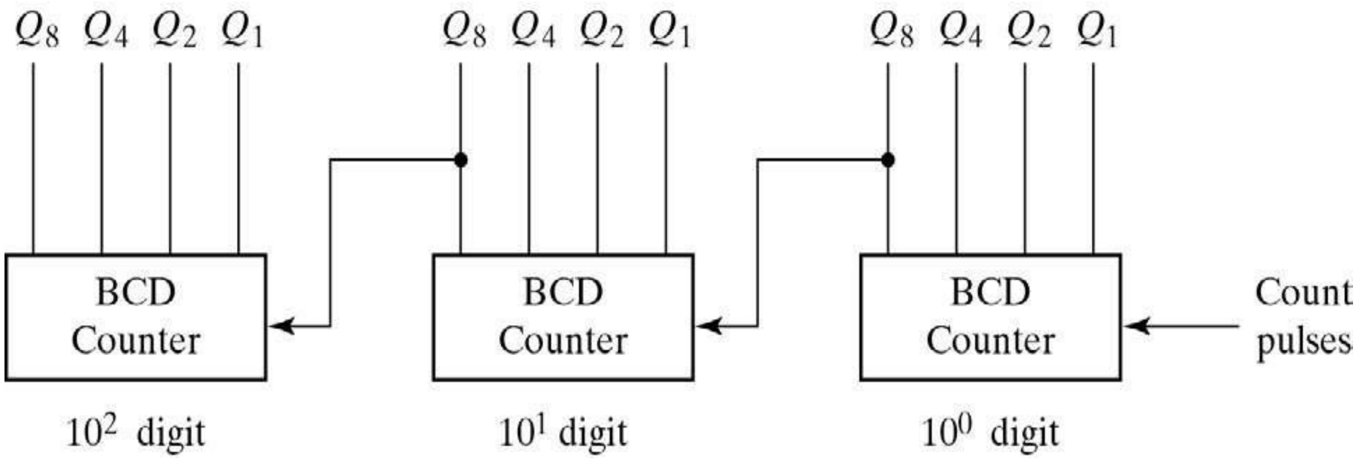


The logic diagram of a BCD counter using JK flip-flops is shown here



Contd…

- A multiple decade counter can be constructed by connecting BCD counters in cascade. A three decade counter is shown below:



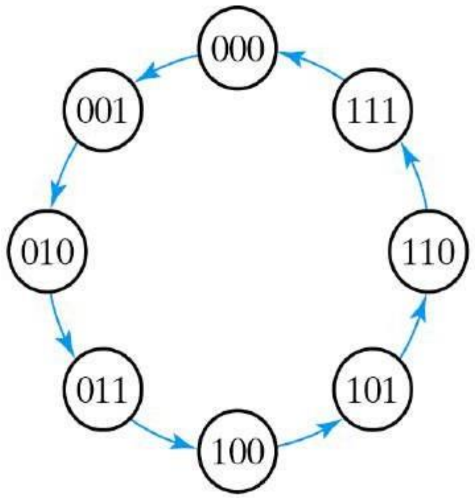
SETTLING TIME OF RIPPLE COUNTERS

- A ripple counter is also known as an asynchronous counter. The rippling behaviour affects the overall settling time.
- The worst-case delay occurs when the counter goes from its 11….1-state to its 00…0-state.
- For an n-stage binary ripple counter, the worst case
- setting time is n x Tpd, where Tpd is the propagation delay associated with each flip-flop.

SYNCHRONOUS BINARY COUNTERS

- The settling time problem associated with ripple counters is avoided in synchronous counters. In these counters, the count pulses are applied directly to the control inputs C of all flip-flops.

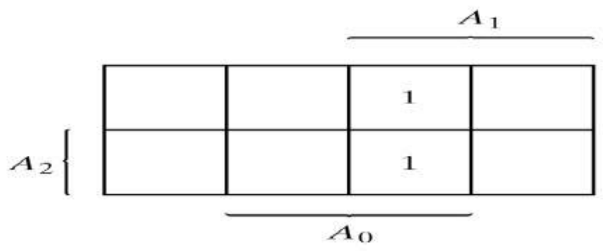
- The state diagram and state table of a 3-bit binary counter are



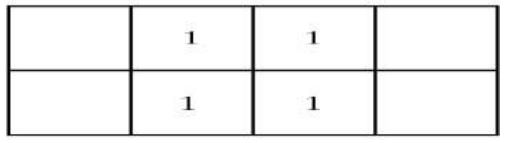
Contd…

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

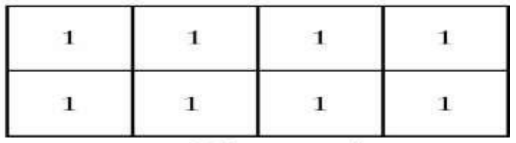
The flip-flop input equations are specified by the *Kmaps:*



$T_{A2} = A_1 A_0$



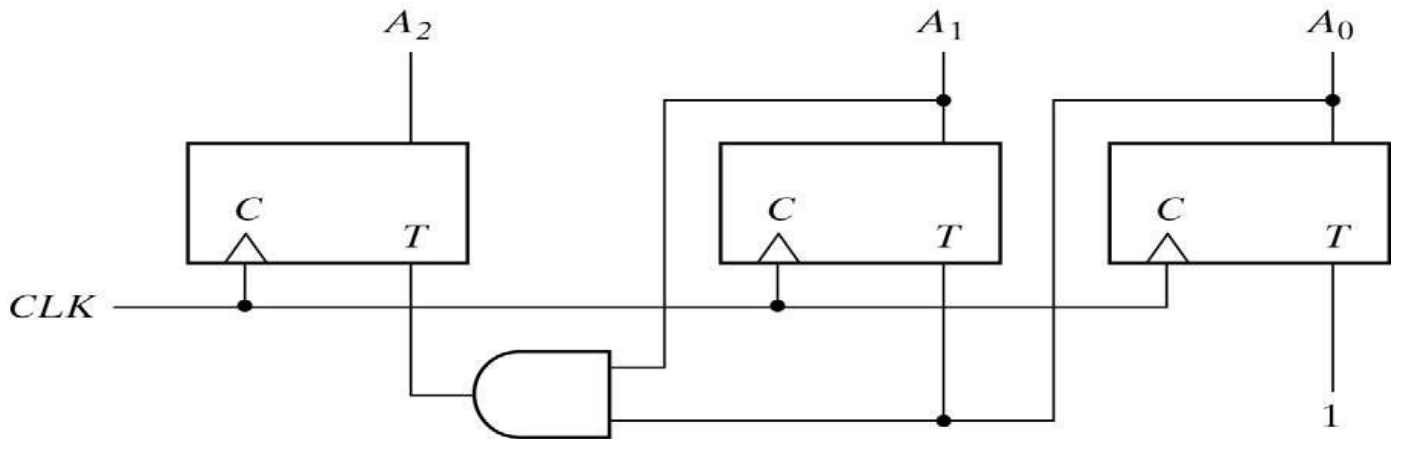
$T_{A1} = A_0$



$T_{A0} = 1$

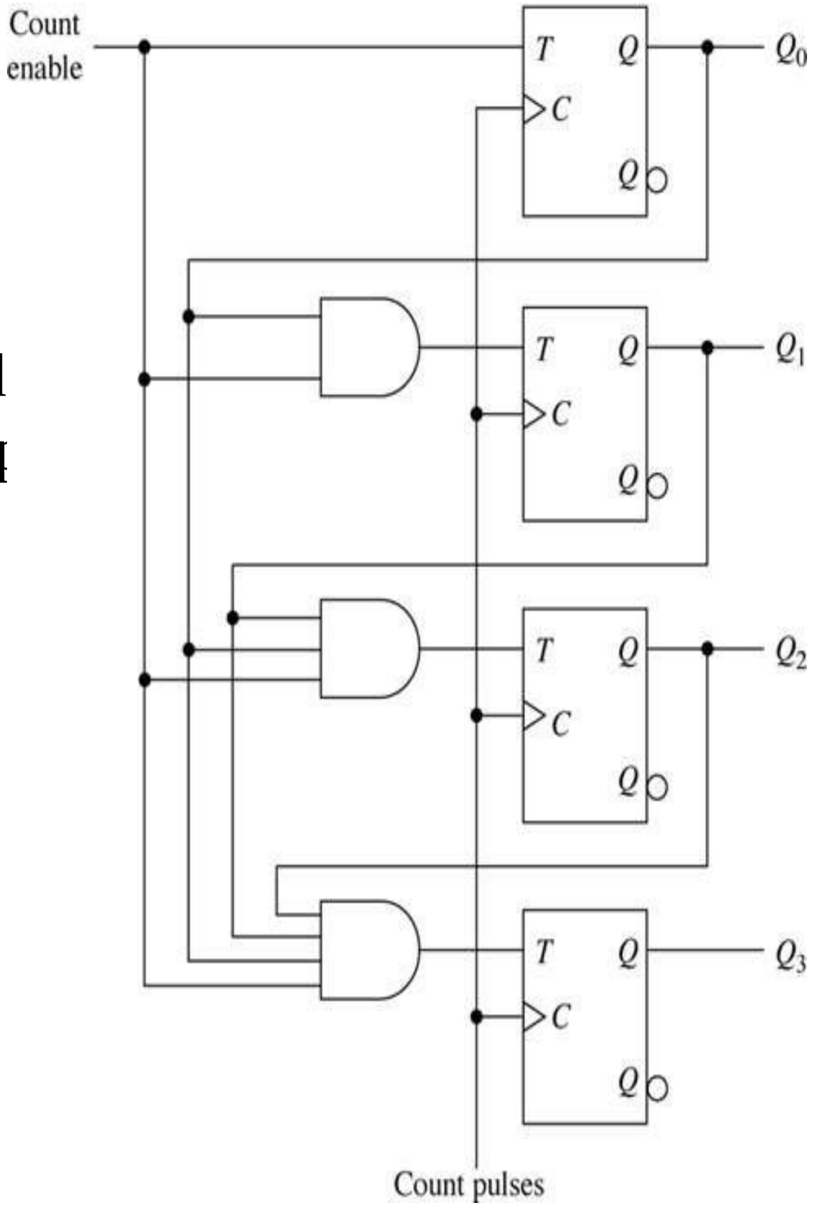
Contd…

- The input equations listed under the *K-maps* specify the combinational part of the counter. Including these functions with the three *T flip-flops*, the logic diagram of the counter is:



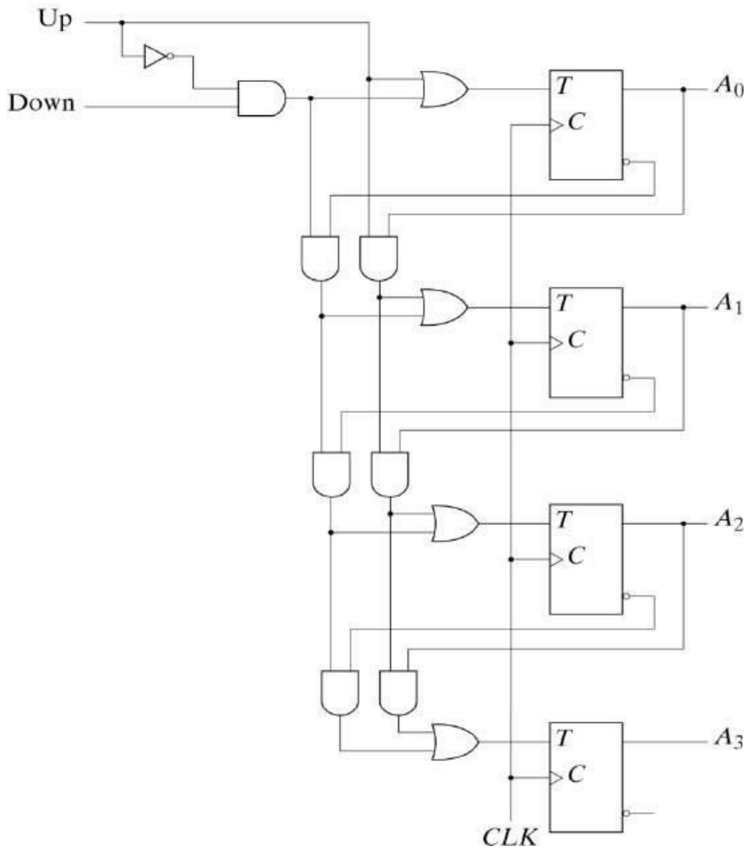
Contd…

- Synchronous counters have a regular pattern and can be constructed with complementing flip-flops and gates. The complementing flip flops can be either of the JK-type or the T-type or the D-type with X-OR gates.
- A 4-bit binary synchronous counter (count-down) with count enable function can be realized like this:



UP-DOWN BINARY COUNTER

- The circuit of a 4-bit up-down binary counter withT flip-flops is:

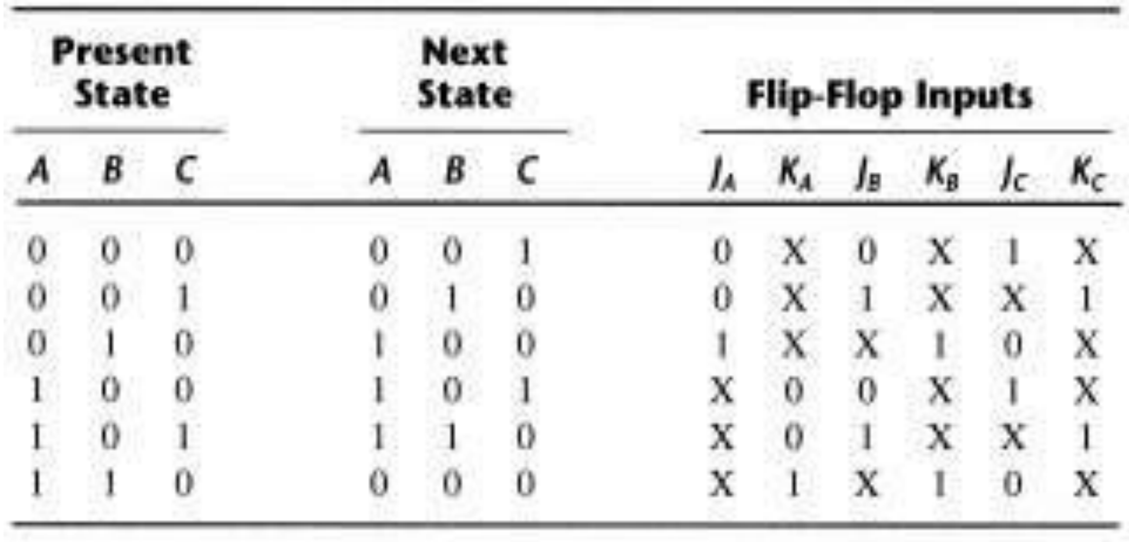


Up = 1; the circuit counts up.

- Down = 1, Up = 0; the circuit counts down.
- Up = 0, Down = 0; the circuit doesn't change state.
- Up = 1, Down = 1, the circuit counts up.

COUNTER WITH UNUSED STATES

- A circuit with n flip-flops has 2n binary states. There are occasions when a sequential circuit uses less than 2n states. The unused states maybe treated as don’t care conditions or may be assigned specific next states. Once the circuit is designed and realized, outside interference may cause it to enter one of the unused states. In that case it is important to ensure that the circuit can resume normal operation.



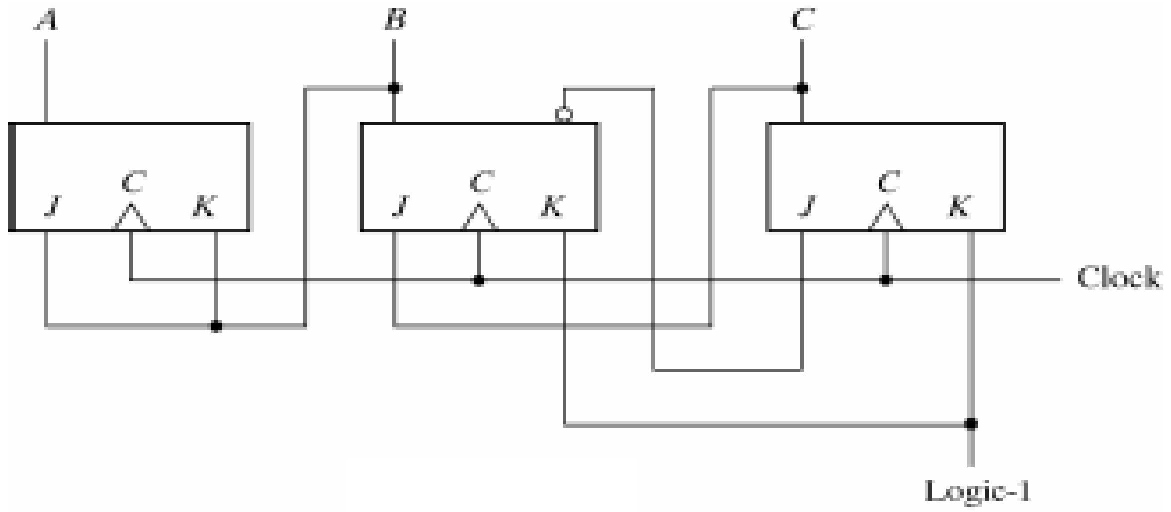
Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

Contd….

- The flip-flop input equations (after simplification) are:

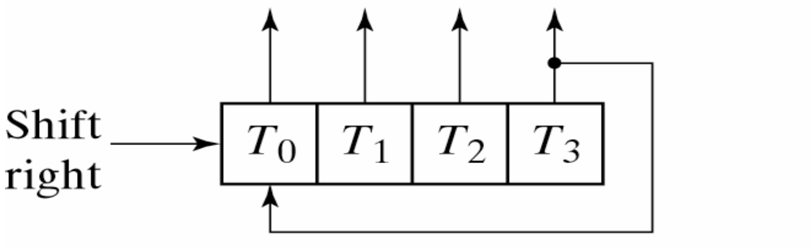
$$J_A = B \qquad K_A = B$$
$$J_B = C \qquad K_B = 1$$
$$J_C = B' \qquad K_C = 1$$

The logic diagram of the counter is:

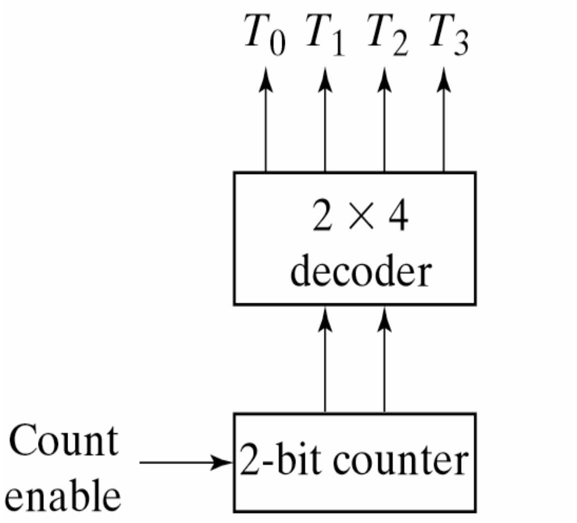


RING COUNTER

- It is a circular shift register with only one flip-flop being set at any particular time, all others are cleared. The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals. A 4-bit shift register connected as a ring counter is shown below:



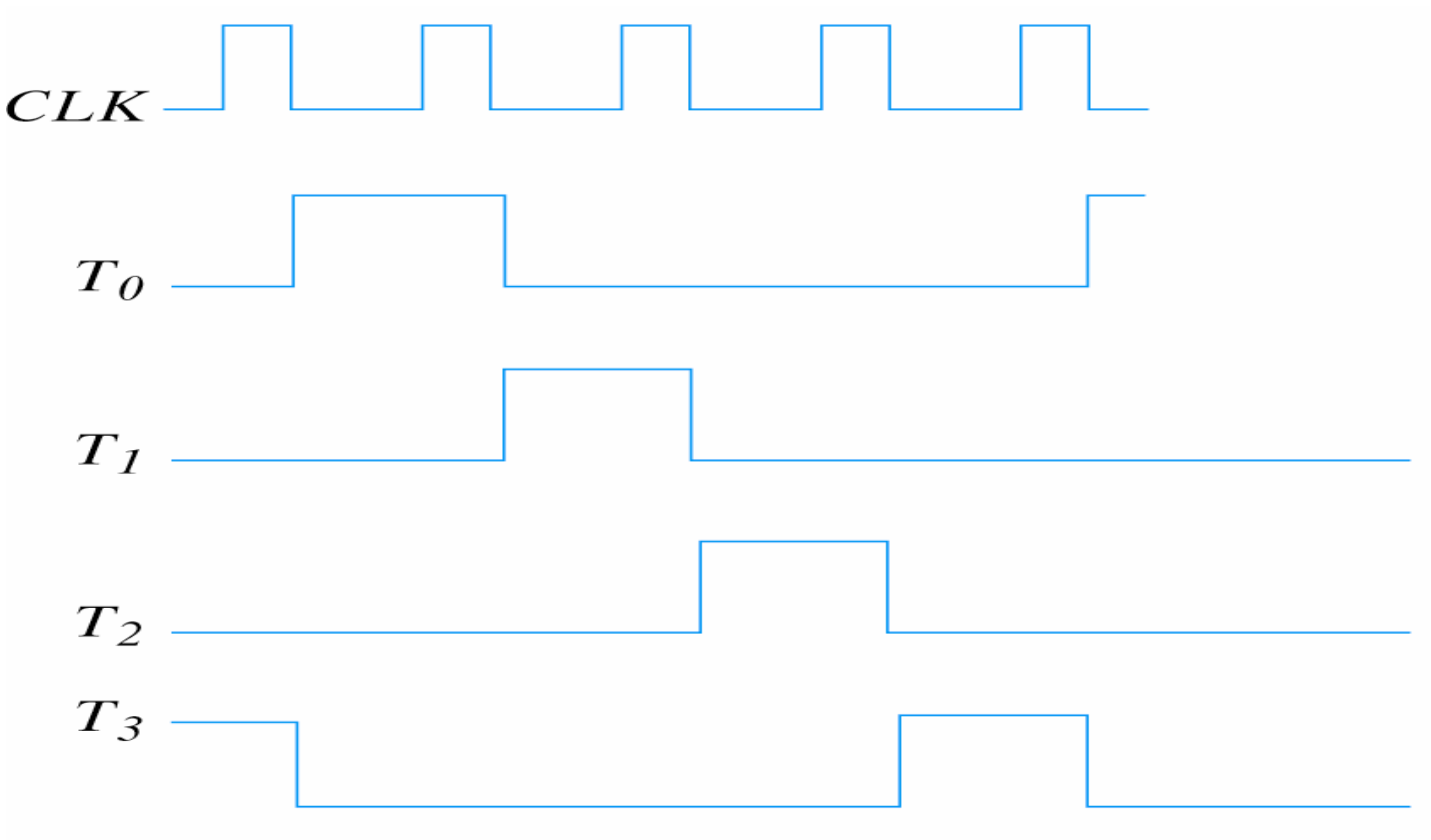
(a) Ring-counter (initial value = 1000)



(b) Counter and decoder

Contd…

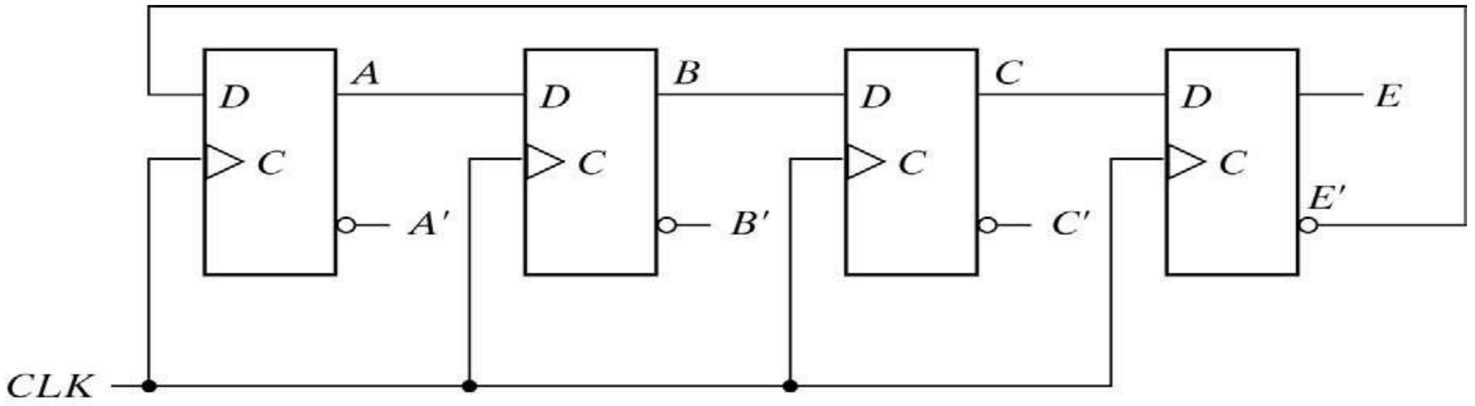
- **Timing diagram**



(c) Sequence of four timing signals

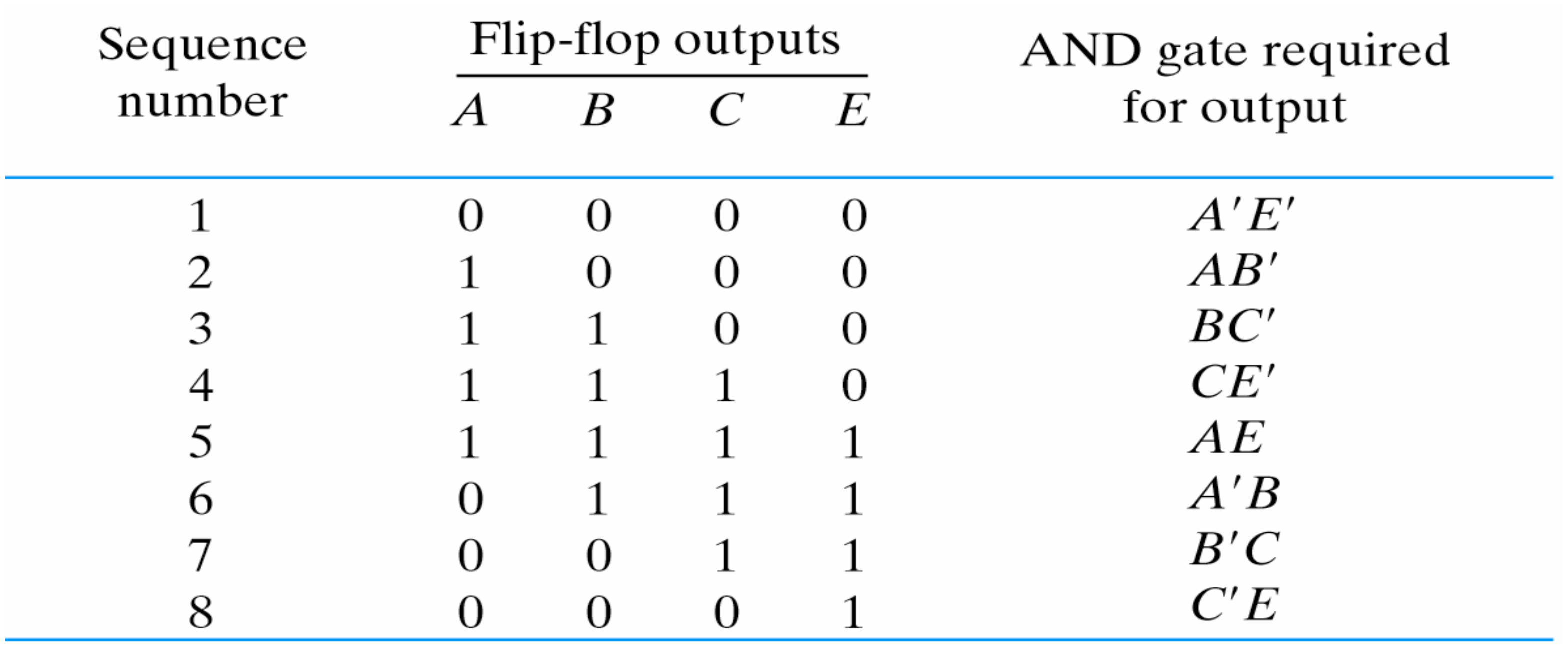
JOHNSON COUNTER

- An interesting variation of the ring counter is obtained if, instead of the Q output we take the Q′ of the last stage and feed it back to the first stage. A four-stage switch-tail counter is shown below:



Contd…..

- Starting from a cleared state, the switch-tail counter goes through a sequence of eight states as listed below:



Sequence number	Flip-flop outputs				AND gate required for output
	A	*B*	*C*	*E*	
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

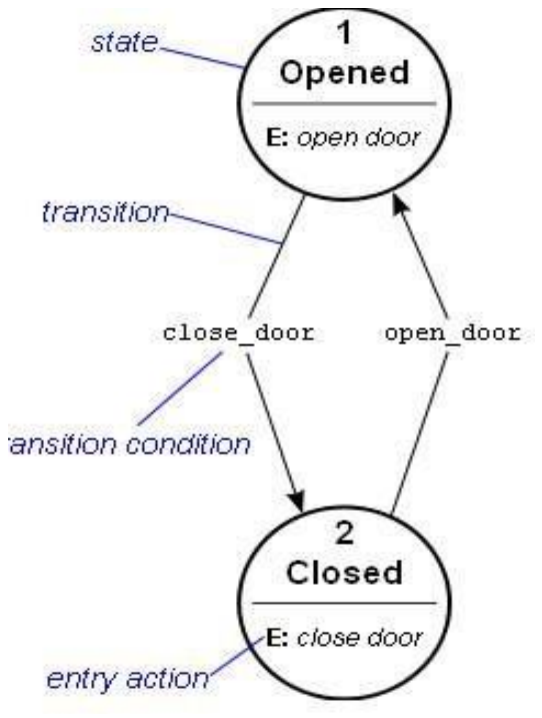
STATE MACHINES

STATE MACHINE

- A finite state machine (FSM) or finite state automaton (plural: automata), or simply a state machine, is a model of behavior composed of a finite number of states, transitions between those states, and actions. It is similar to a "flow graph" where we can inspect the way in which the logic runs when certain conditions are met. A finite state machine is an abstract model of a machine with a primitive internal memory.
- Two types of “ STATE machines”
  - Mealy machine
  - Moore machine

Contd…

- **EXAMPLE**



state machine tells
about how the states of a particular system or machine changes .Before designing a circuit we have to analyze how the states are changing in that system. That can be done using a state machine.

DATA STORING DEVICES
RANDOM ACCESS MEMORY (RAM)

RANDOM ACCESS MEMORY (RAM)

- Memory is a collection of storage cells with associated input and output circuitry
- Possible to read and write cells
- Random access memory (RAM) contains words of information
- Data accessed using a sequence of signals
  - Leads to timing waveforms
- Decoders are an important part of memories
  - Selects specific data in the RAM
- Static RAM loses values when circuit power is removed.

PRELIMINARIES

- RAMs contain a collection of data bytes
  - A collection of bytes is called a word
  - A sixteen bit word contains two bytes
  - Capacity of RAM device is usually described in bytes (e.g. 16 MB)
- Write operations write data to specific words
- Read operations read data from specific words
- Note: new notation for OR gate

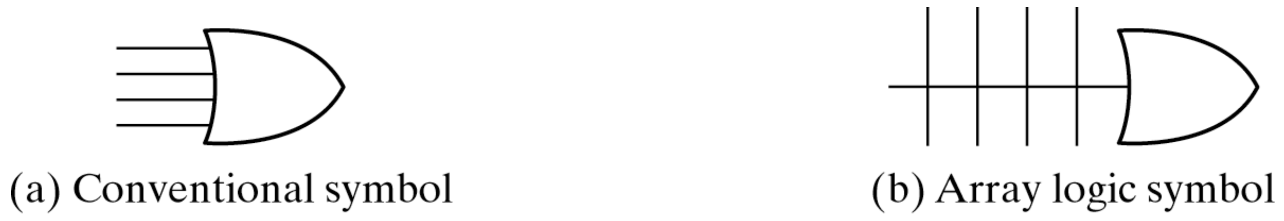
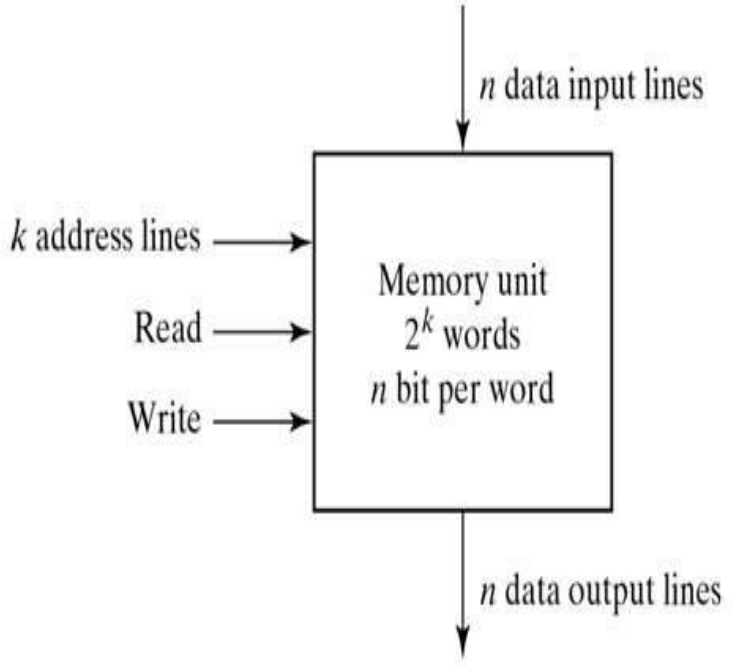


Fig. 7-1 Conventional and Array Logic Diagrams for OR Gate

RAM INTERFACE SIGNALS

- Data input and output lines carry data
- Memory contains 2^k words
- k address lines select one word out of 2^k
- Read asserted when data to be transferred to output
- Write asserted when data input to be stored



Block Diagram of a Memory Unit

TYPES OF RANDOM ACCESS MEMORIES

- **Static random access memory (SRAM)**
  - Operates like a collection of latches
  - Once value is written, it is guaranteed to remain in the memory as long as power is applied
  - Generally expensive
  - Used inside processors (like the Pentium)
- **Dynamic random access memory (DRAM)**
  - Generally, simpler internal design than SRAM
  - Requires data to be rewritten (refreshed), otherwise data is lost
  - Often hold larger amount of data than SRAM
  - Longer access times than SRAM
  - Used as main memory in computer systems

Inside the RAM Device

- Address inputs go into decoder
- Word line selects a row of bits (word)
- Data passes through OR gate
- Each binary cell (BC) stores one bit
- Input data stored if Read/Write is 0
- Output data driven if Read/Write is 1

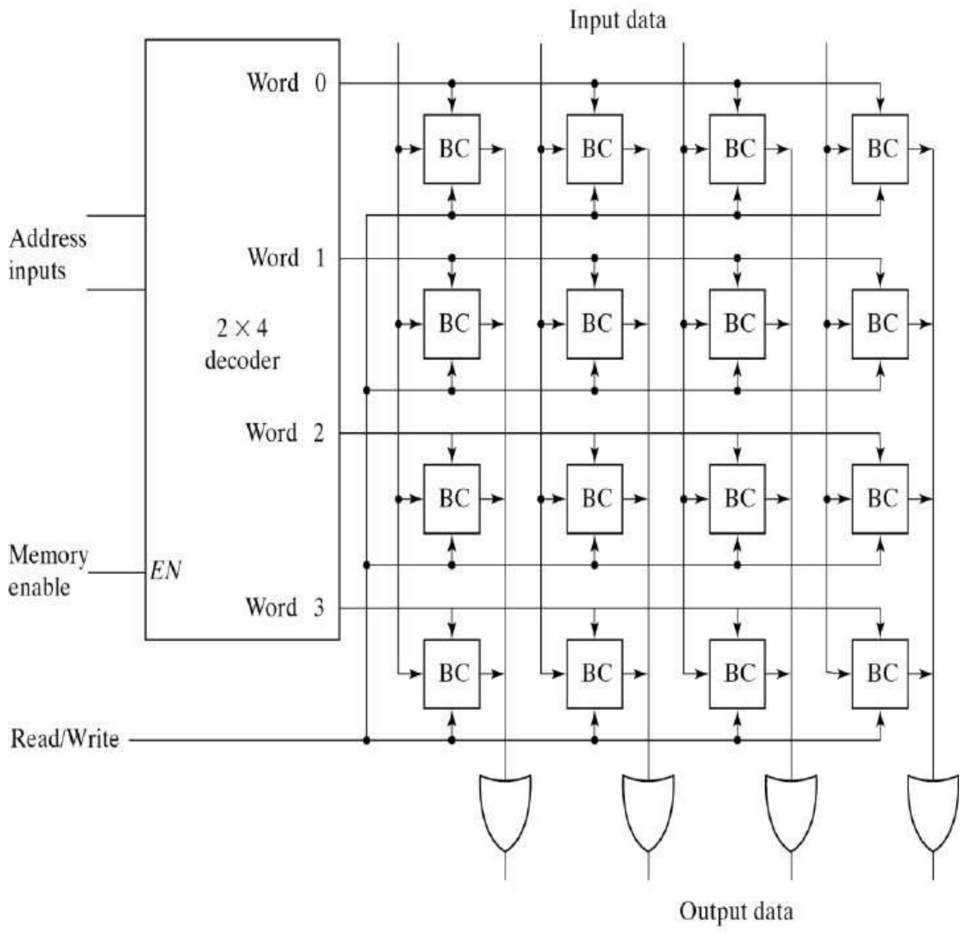


Fig. 7-6 Diagram of a 4 × 4 RAM

Read Only Memory (ROM)

- ROM holds programs and data permanently even when computer is switched off
- Data can be read by the CPU in any order so ROM is also direct access
- The contents of ROM are fixed at the time of manufacture
- Stores a program called the bootstrap loader that helps start up the computer
- Access time of between 10 and 50 nanoseconds

Types of ROM

- **1. Programmable Read Only Memory (PROM)**
- Empty of data when manufactured
- May be permanently programmed by the user

- **2. Erasable Programmable Read Only Memory(EPROM)**
- Can be programmed, erased and reprogrammed
- The EPROM chip has a small window on top allowing it to be erased by shining ultra-violet light on it
- After reprogramming the window is covered to prevent new contents being erased
- Access time is around 45 – 90 nanoseconds

3. Electrically Erasable Programmable Read Only Memory

- Reprogrammed electrically without using ultraviolet light
- Must be removed from the computer and placed in a special machine to do this
- Access times between 45 and 200 nanoseconds

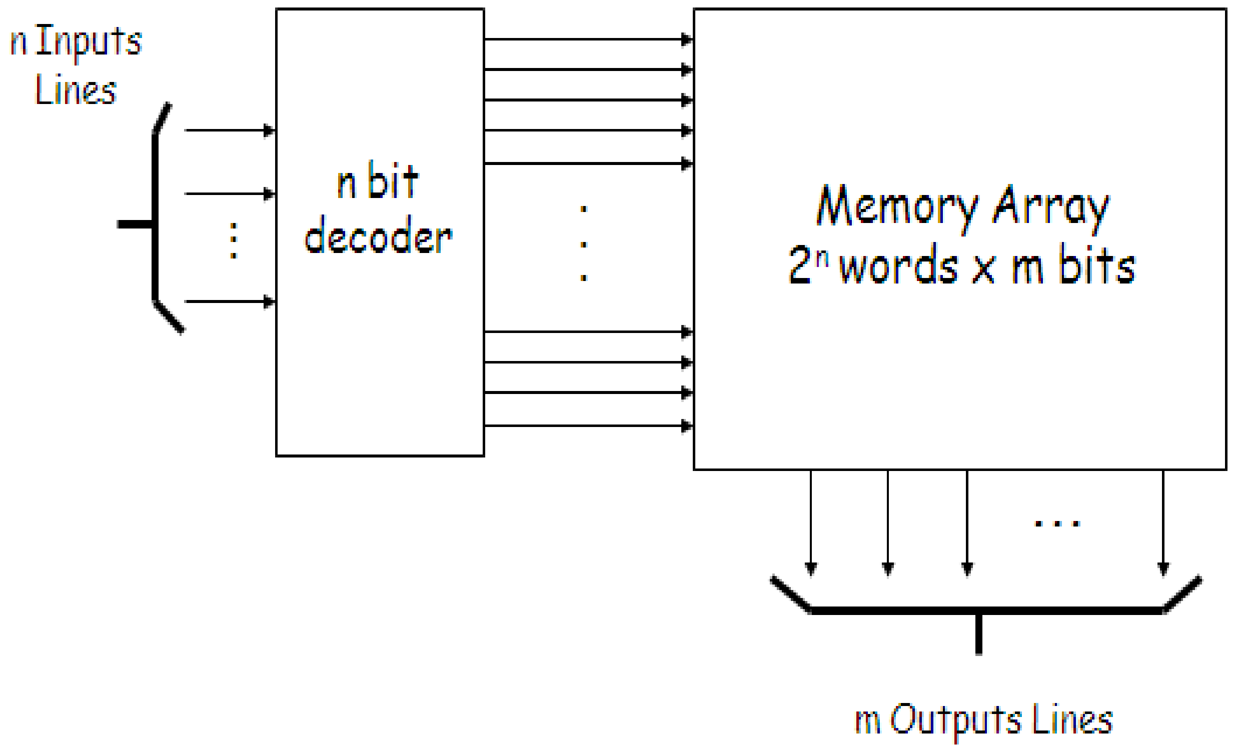
4. Flash ROM

- Similar to EEPROM
- However, can be reprogrammed while still in the computer
- Easier to upgrade programs stored in Flash ROM
- Used to store programs in devices e.g. modems
- Access time is around 45 – 90 nanoseconds

5. ROM cartridges

- Commonly used in games machines
- Prevents software from being easily copied

ROM Internal Structure



INTERNAL LOGIC OF 32X8 ROM

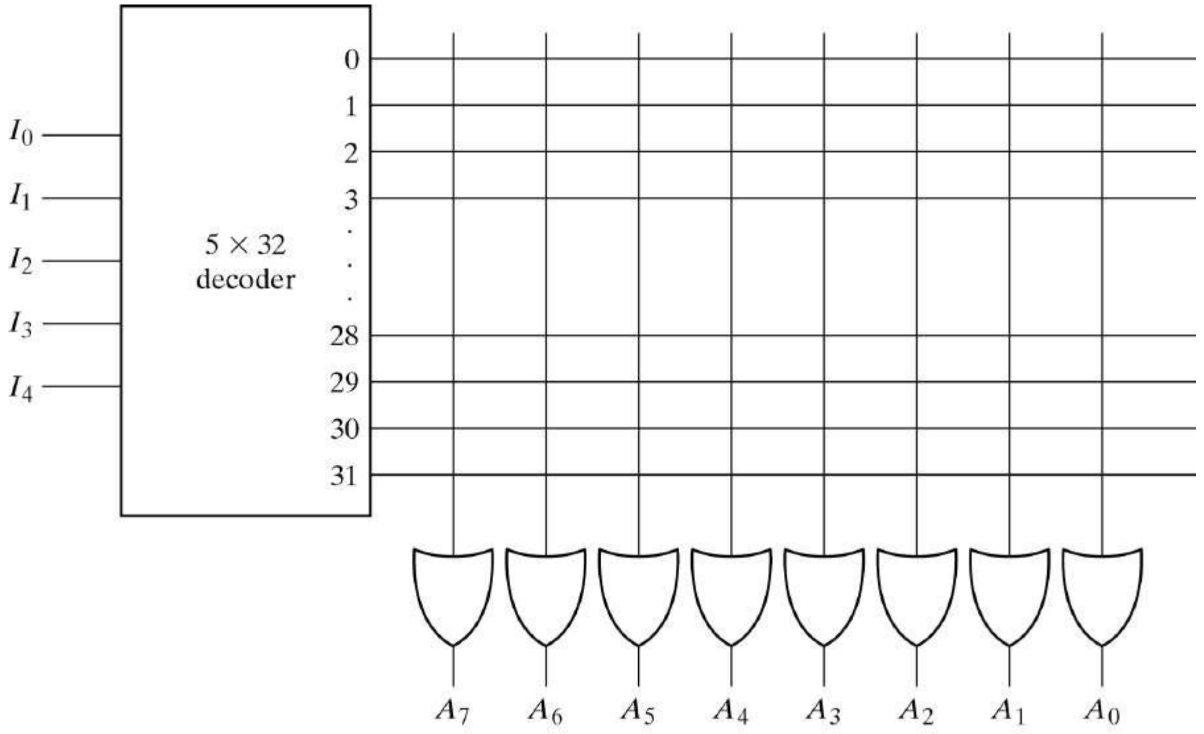


Fig. 7-10 Internal Logic of a 32 × 8 ROM

THANK YOU