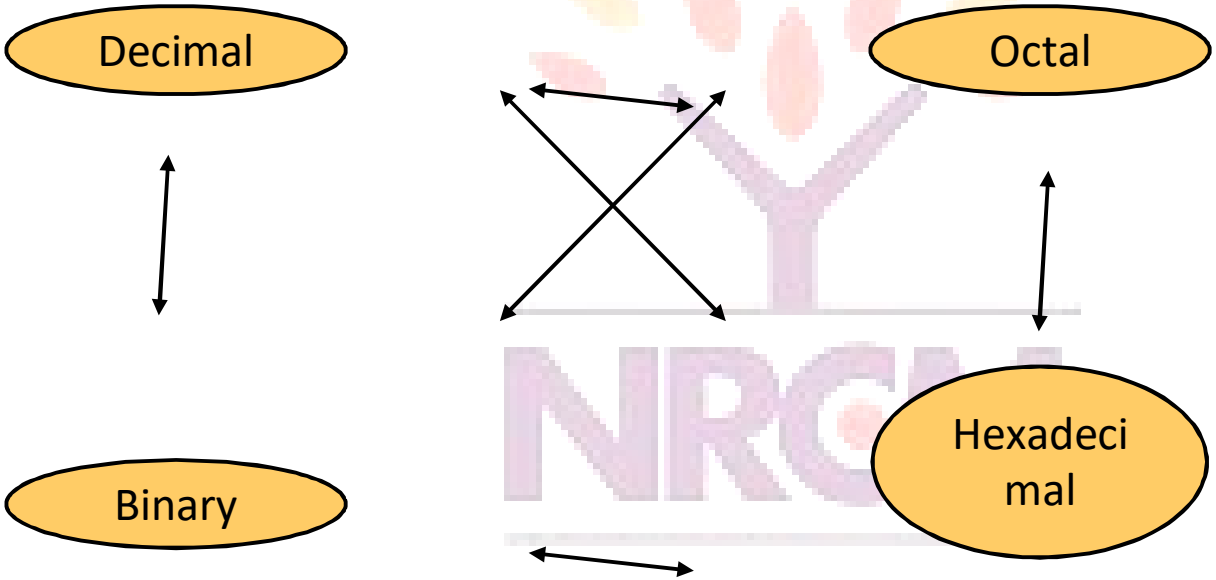
Number Systems

Common Number Systems

System	Base	Symbols	Used by humans?	Used in computers?
Decimal	10	0, 1, … 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, … 7	No	No
Hexa-decimal	16	0, 1, … 9, A, B, … F	No	No

Conversion Among Bases

- The possibilities:



Quick Example

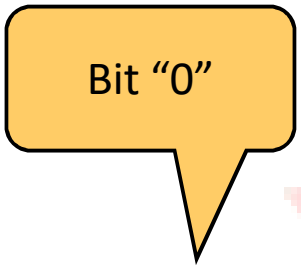
$25_{10} = 11001_2 = 31_8 = 19_{16}$

Base

Binary to Decimal

- Technique
  - Multiply each bit by 2^n, where *n* is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

Example



101011_2 => $1 \times 2^0 = 1$

$1 \times 2^1 = 2$

$0 \times 2^2 = 0$

$1 \times 2^3 = 8$

$0 \times 2^4 = 0$

$1 \times 2^5 = 32$

43_{10}

Octal to Decimal

- Technique
  - Multiply each bit by 8^n, where *n* is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

724_8 =>

$4 \times 8^0 = 4$

$2 \times 8^1 = 16$

$7 \times 8^2 = 448$

468_{10}

- Technique
  - Multiply each bit by 16^n, where n is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

ABC_{16} =>

$C \times 16^0 = 12 \times 1 = 12$

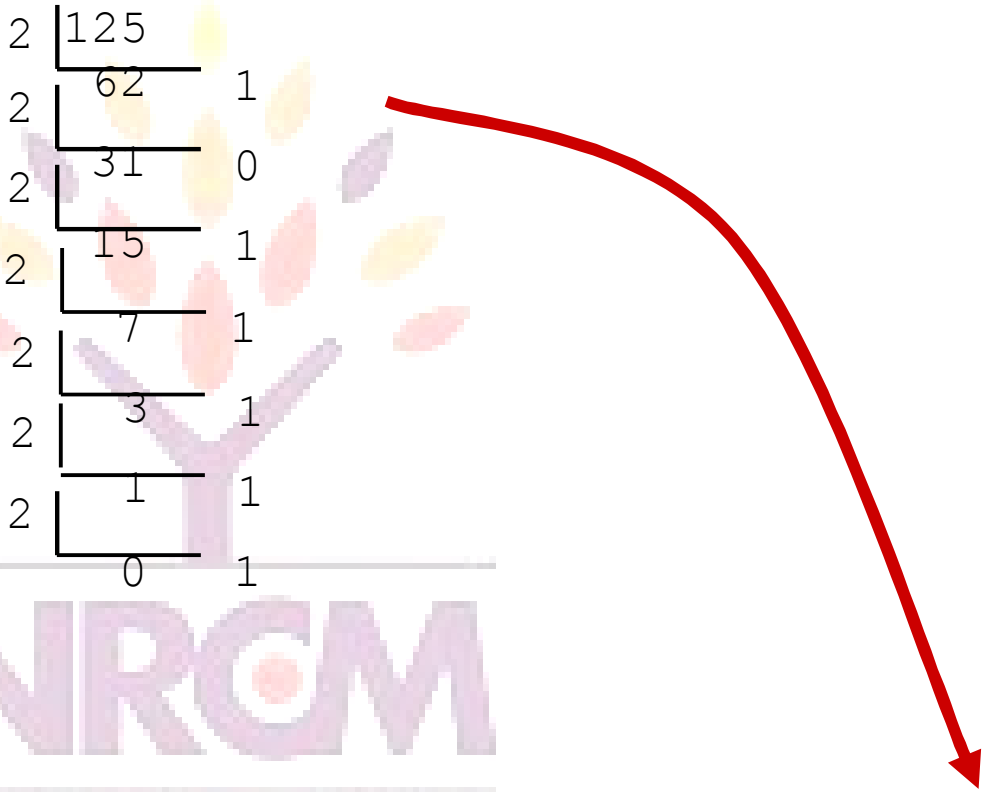
$B \times 16^1 = 11 \times 16 = 176$

$A \times 16^2 = 10 \times 256 = 2560$

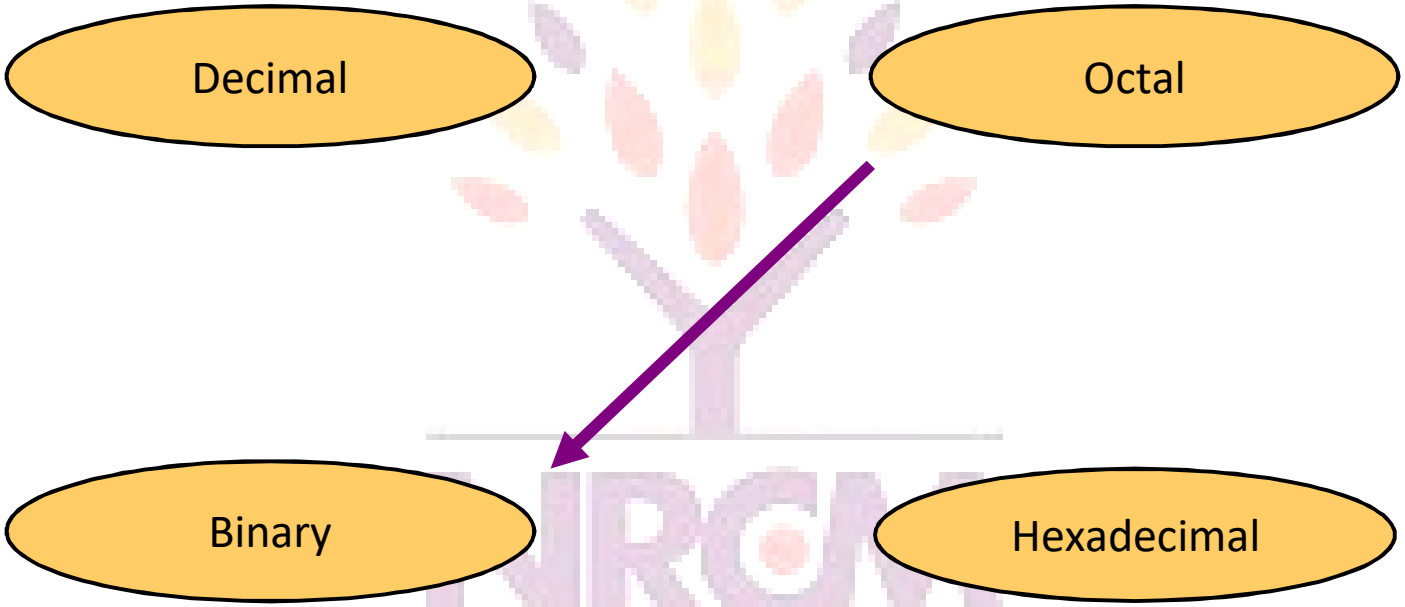
2748_{10}

- Technique
  - Divide by two, keep track of the remainder
  - First remainder is bit 0 (LSB, least-significant bit)
  - Second remainder is bit 1
  - Etc.

$125_{10} = ?_2$



$125_{10} = 1111101_2$



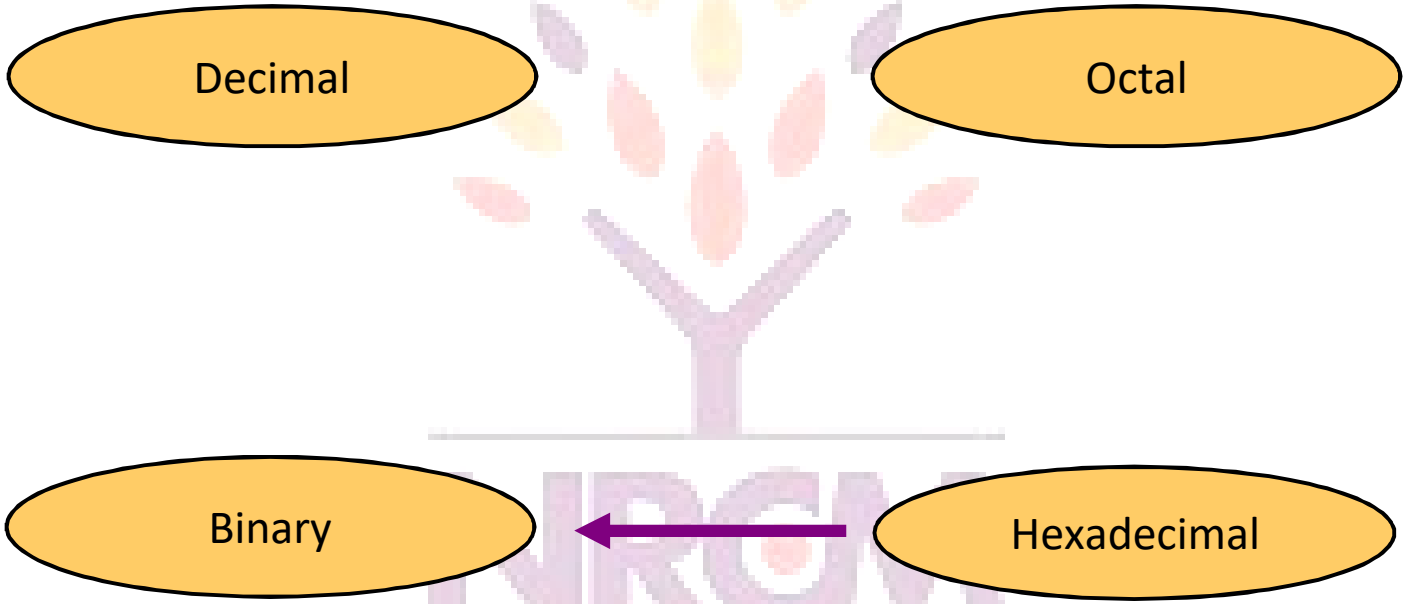
- Technique
  - Convert each octal digit to a 3-bit equivalent binary representation

$705_8 = ?_2$

7 0 5

111 000 101

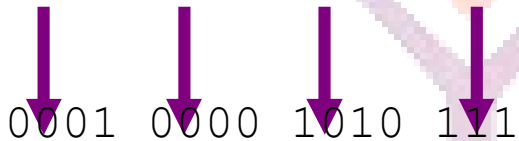
$705_8 = 111000101_2$



- Technique
  - Convert each hexadecimal digit to a 4-bit equivalent binary representation

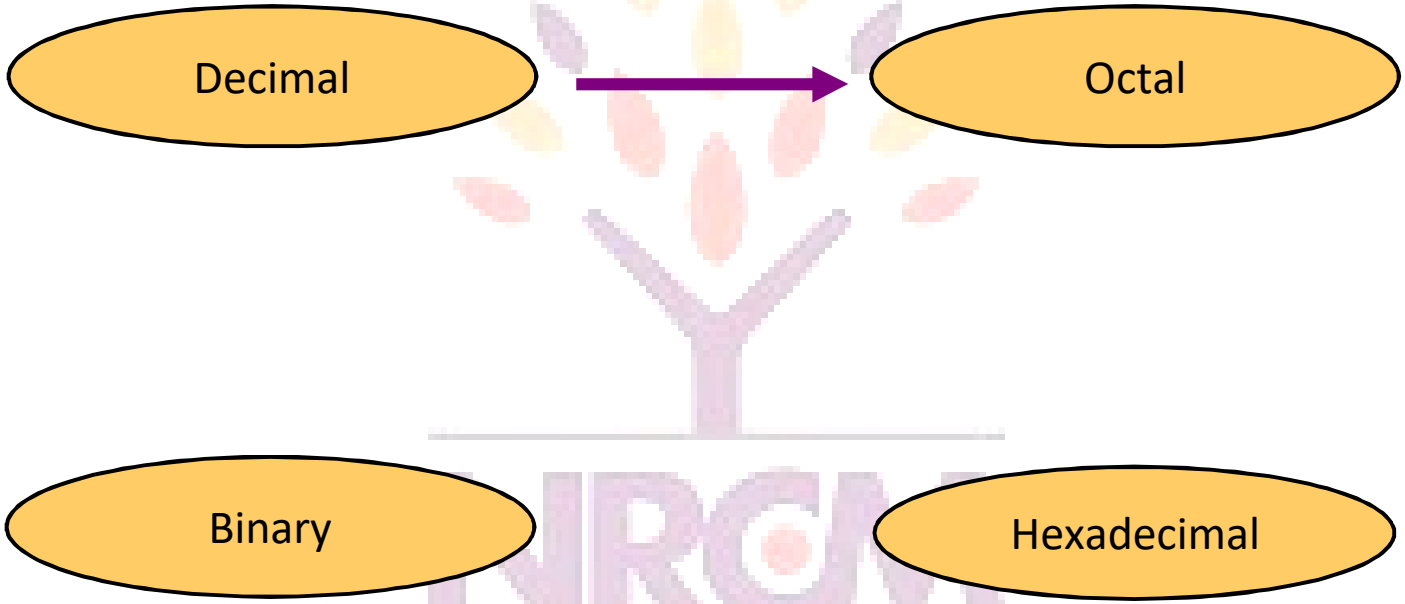
$10AF_{16} = ?_2$

1 0 A F



0001 0000 1010 1111

$10AF_{16} = 0001000010101111_2$

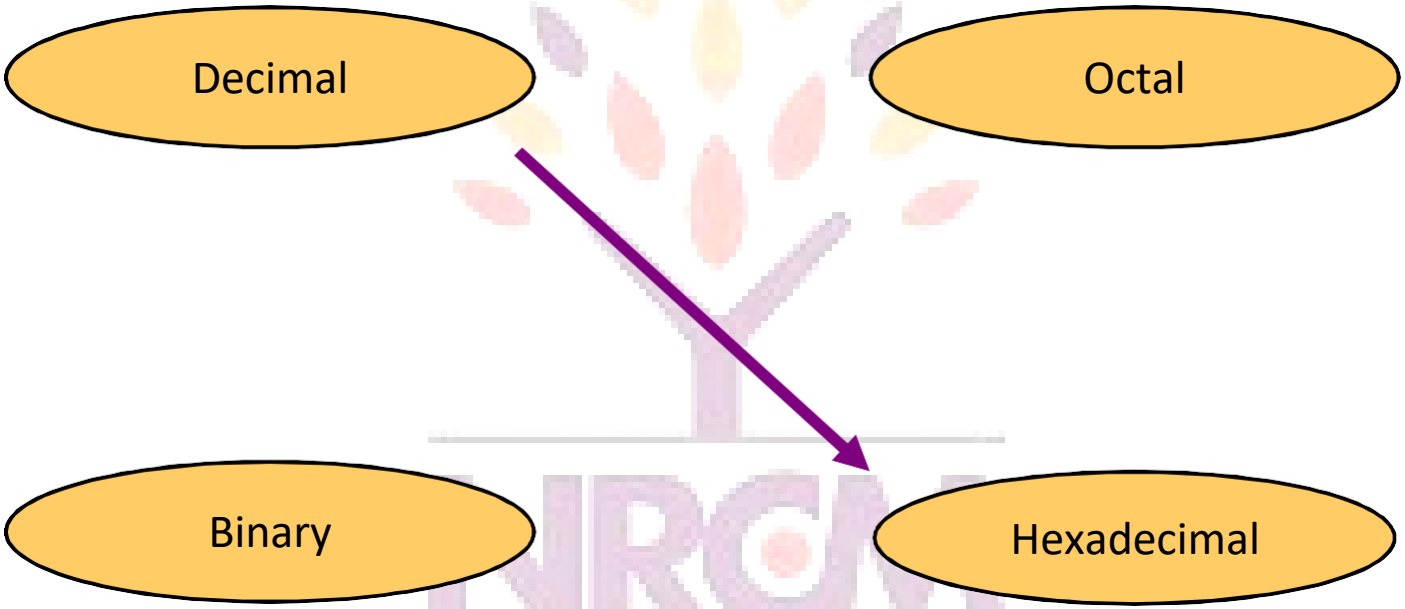


- Technique
  - Divide by 8
  - Keep track of the remainder

$1234_{10} = ?_8$

8	1234	
	154	2
8	19	2
8	2	3
8	0	2

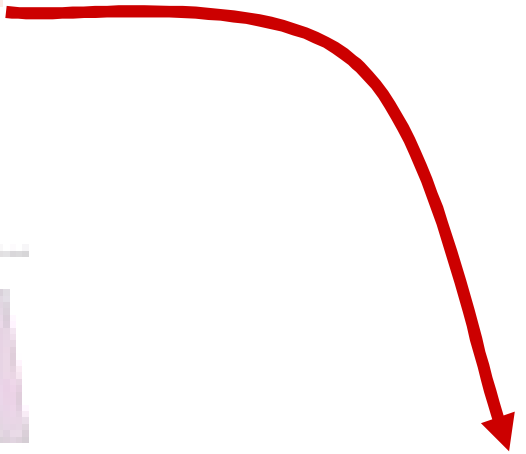
$1234_{10} = 2322_8$



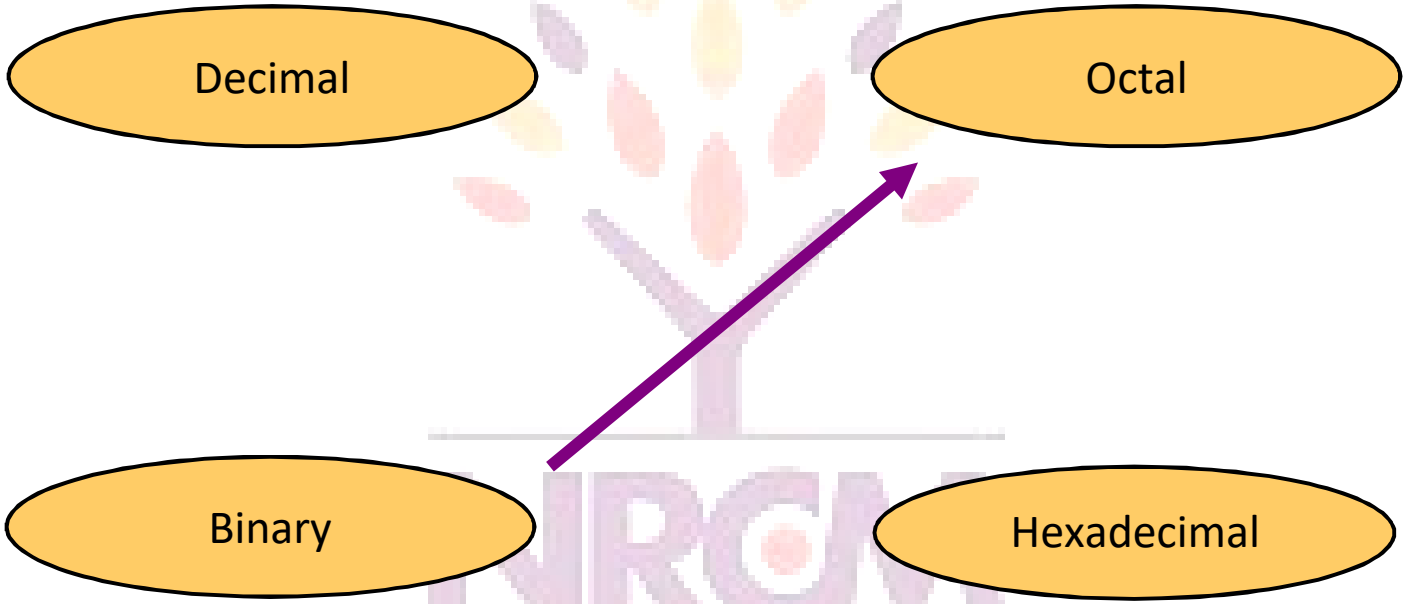
- Technique
  - Divide by 16
  - Keep track of the remainder

$1234_{10} = ?_{16}$

16	1234	
16	77	2
16	4	13 = D
	0	4

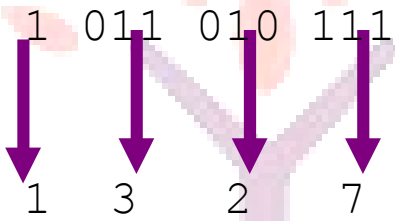


$1234_{10} = 4D2_{16}$

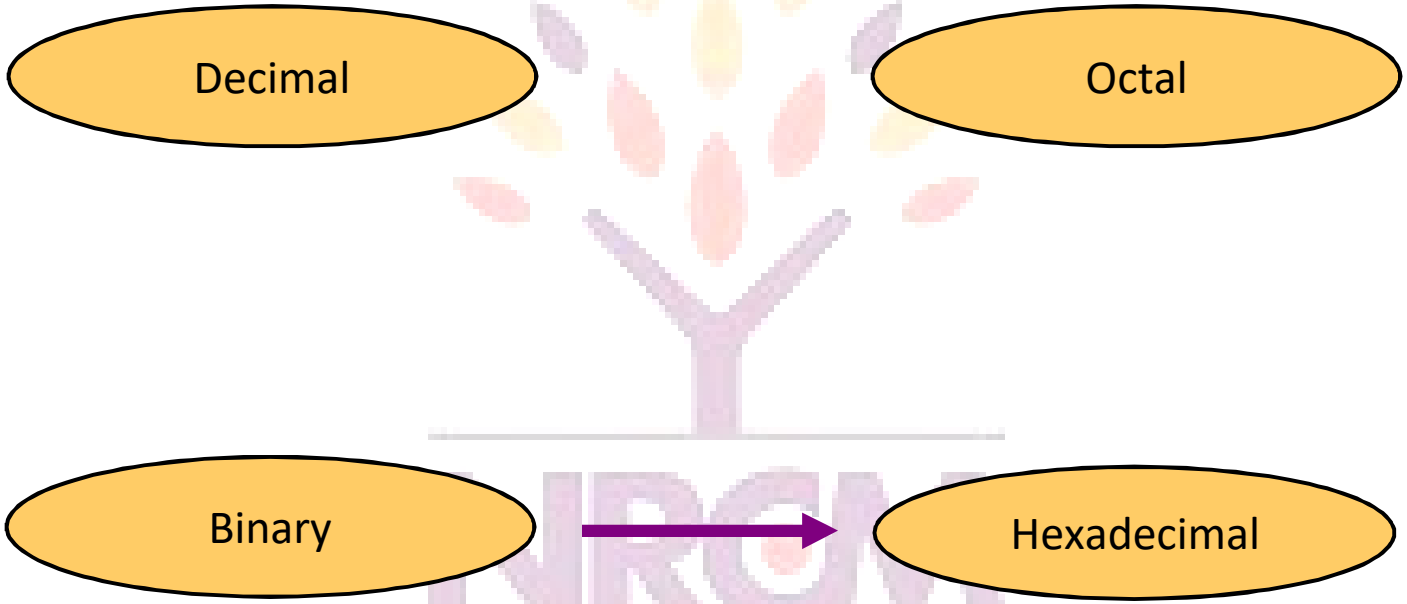


- Technique
  - Group bits in threes, starting on right
  - Convert to octal digits

$1011010111_2 = ?_8$

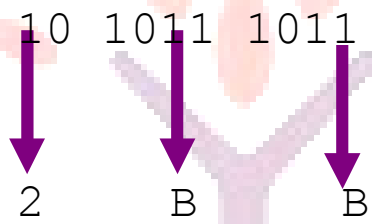


$1011010111_2 = 1327_8$

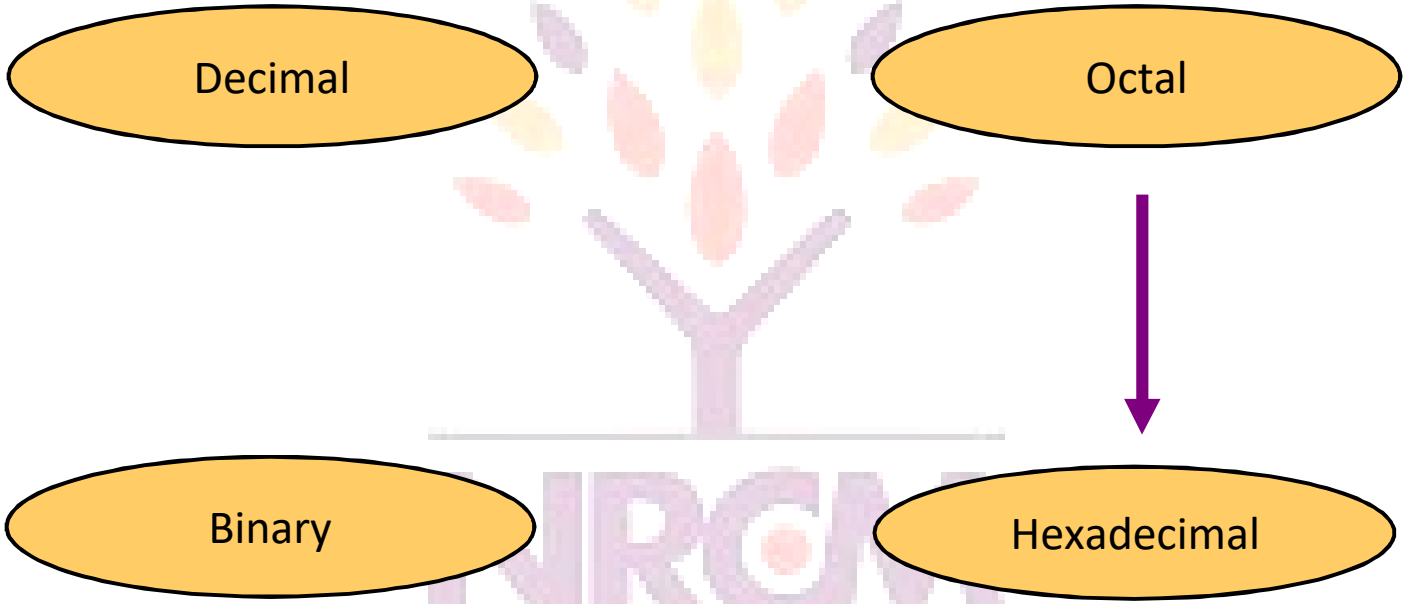


- Technique
  - Group bits in fours, starting on right
  - Convert to hexadecimal digits

$1010111011_2 = ?_{16}$

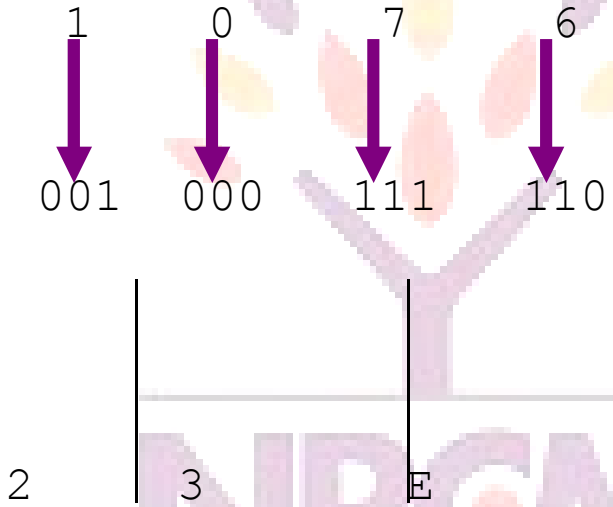


$1010111011_2 = 2BB_{16}$

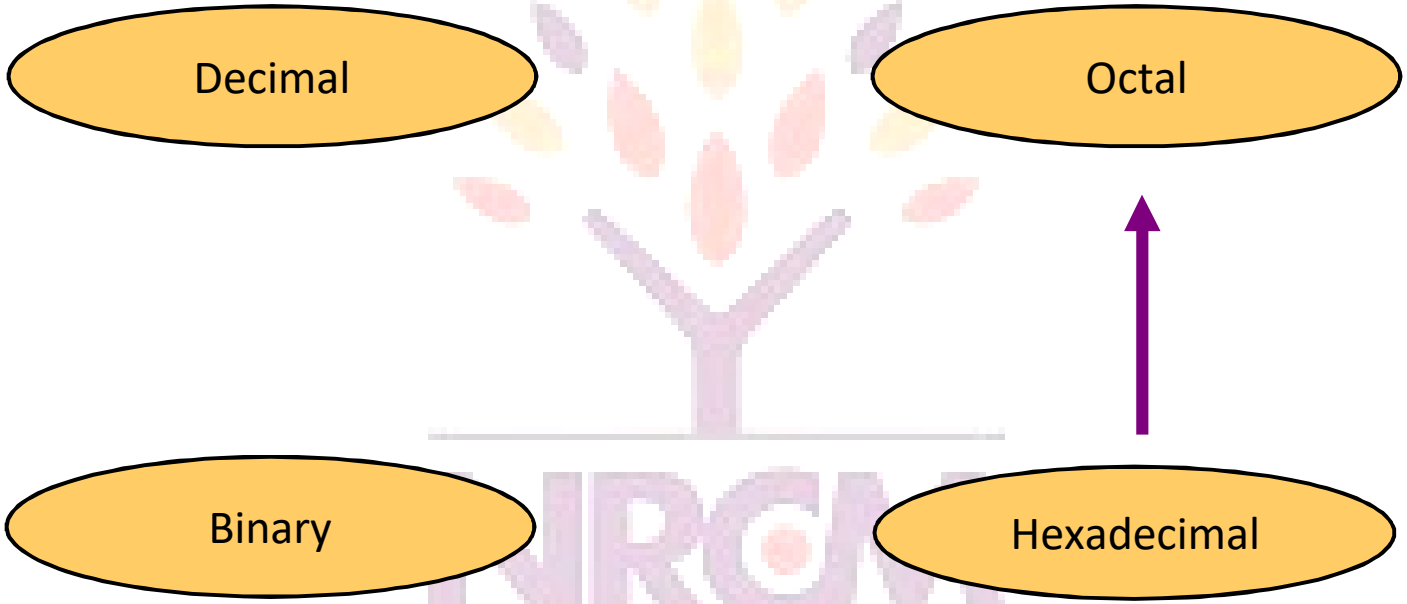


- Technique
  - Use binary as an intermediary

$1076_8 = ?_{16}$

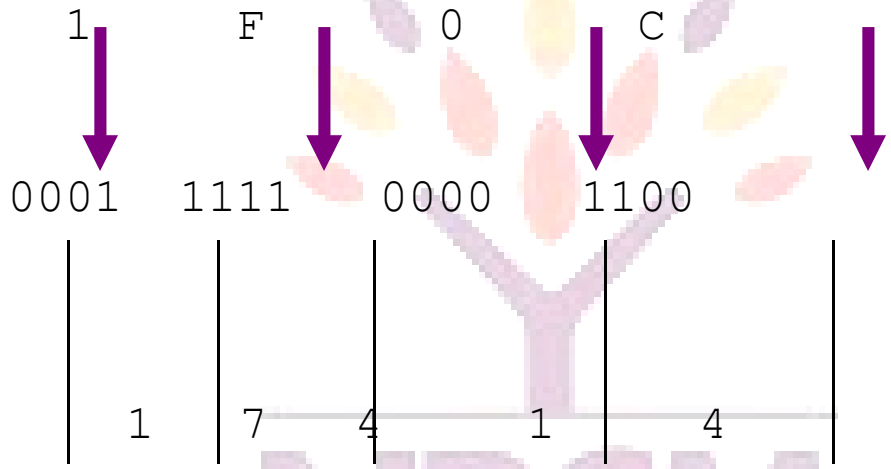


$1076_8 = 23E_{16}$



- Technique
  - Use binary as an intermediary

$1F0C_{16} = ?_8$



$1F0C_{16} = 17414_8$

- Base 10

Power	Preface	Symbol	Value
10^{-12}	pico	p	.000000000001
10^{-9}	nano	n	.000000001
10^{-6}	micro	μ	.000001
10^{-3}	milli	m	.001
10^{3}	kilo	k	1000
10^{6}	mega	M	1000000
10^{9}	giga	G	1000000000
10^{12}	tera	T	1000000000000

- Base 2

Power	Preface	Symbol	Value
2^{10}	kilo	k	1024
2^{20}	mega	M	1048576
2^{30}	Giga	G	1073741824

- What is the value of “k”, “M”, and “G”?
- In computing, particularly w.r.t. memory, the base-2 interpretation generally applies

- Two 1-bit values

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	10

"two"

Binary Addition (2 of 2)

- Two *n*-bit values
  - Add individual bits
  - Propagate carries
  - E.g.,

```
  1     1
  10101      21
+ 11001    + 25
-------    ----
101110       46
```

Multiplication (1 of 2)

- Decimal (just for fun)

```
   35
x 105
  175
 000
  35
3675
```


pp. 39

Multiplication (2 of 2)

- Binary, two *n*-bit values
  - As with decimal values
  - E.g.,

```
    1110
  x 1011
  ------
    1110
   1110
  0000
 1110
 --------
 10011010
```

- Decimal to decimal (just for fun)

$3.14 \Rightarrow$

$4 \times 10^{-2} = 0.04$

$1 \times 10^{-1} = 0.1$

$3 \times 10^{0} = 3$

3.14

- Binary to decimal

$10.1011 \Rightarrow \quad 1 \times 2^{-4} = 0.0625$

$1 \times 2^{-3} = 0.125$

$0 \times 2^{-2} = 0.0$

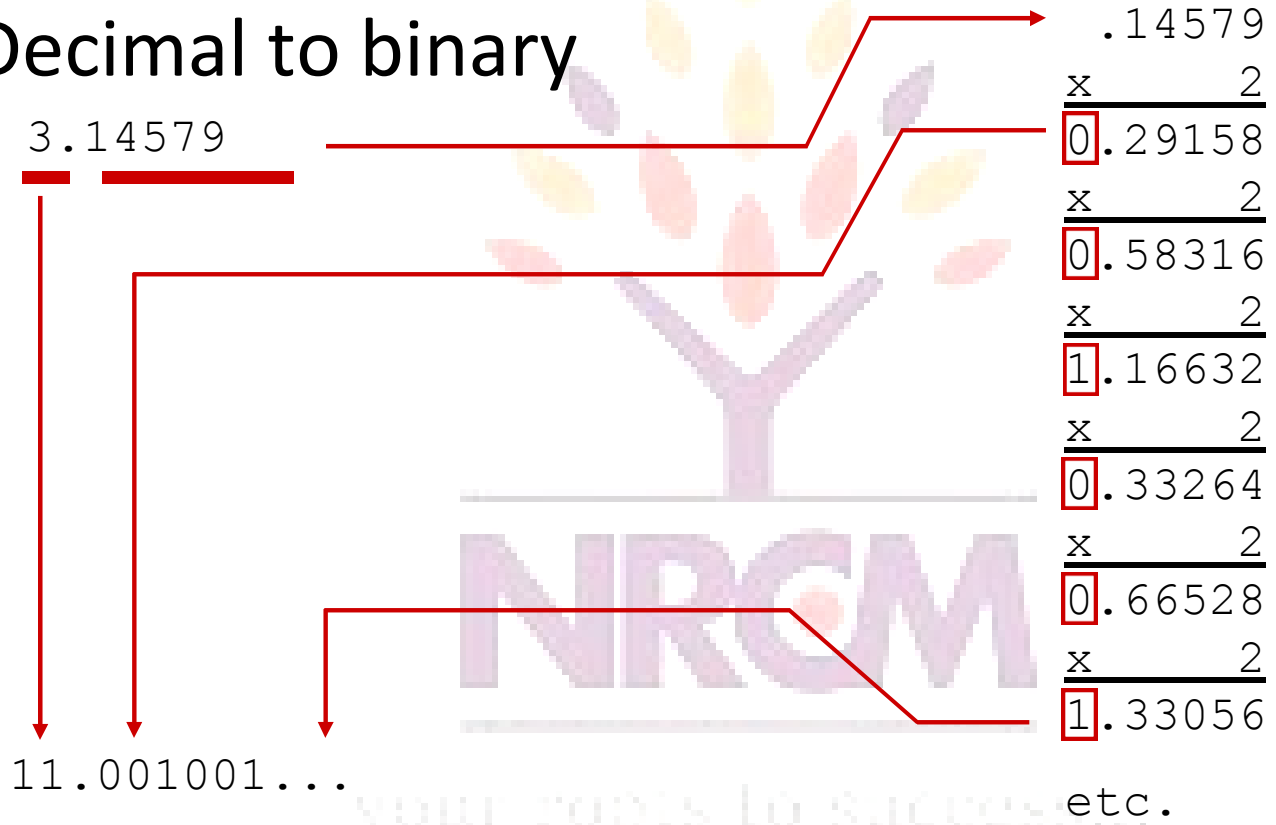
$1 \times 2^{-1} = 0.5$

$0 \times 2^{0} = 0.0$

$1 \times 2^{1} = 2.0$

2.6875

- Decimal to binary



Boolean Algebra

Introduction

- 1854: Logical algebra was published by **George Boole** → known today as "Boolean Algebra"
  - It's a convenient way and systematic way of expressing and analyzing the operation of logic circuits.
- 1938: **Claude Shannon** was the first to apply Boole's work to the analysis and design of logic circuits.

Boolean Operations & Expressions

- *Variable* – a symbol used to represent a logical quantity.
- *Complement* – the inverse of a variable and is indicated by a bar over the variable.
- *Literal* – a variable or the complement of a variable.

Laws & Rules of Boolean Algebra

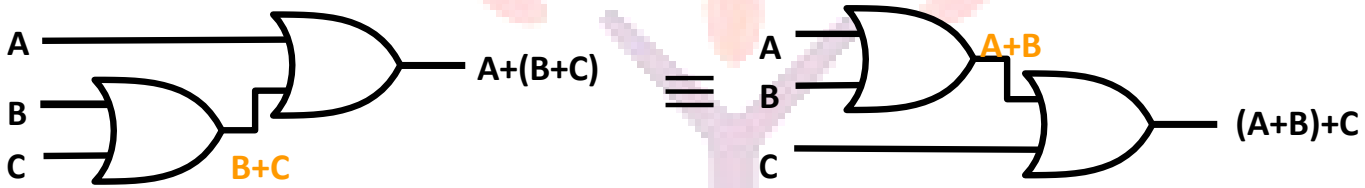
- The basic laws of Boolean algebra:
  - The **commutative** laws
  - The **associative** laws
  - The **distributive** laws

Commutative Laws

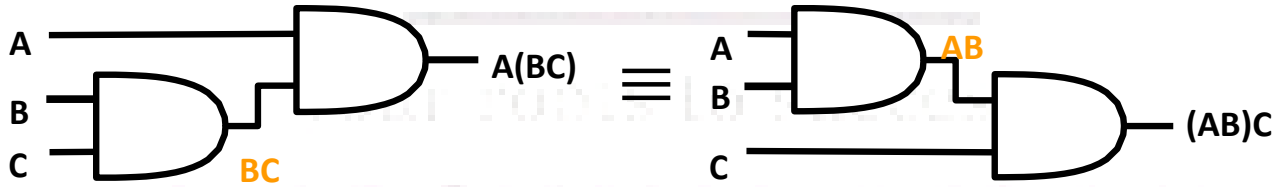
- The *commutative law of addition* for two variables is written as: *A+B = B+A*

- The *commutative law of multiplication* for two variables is written as: *AB = BA*

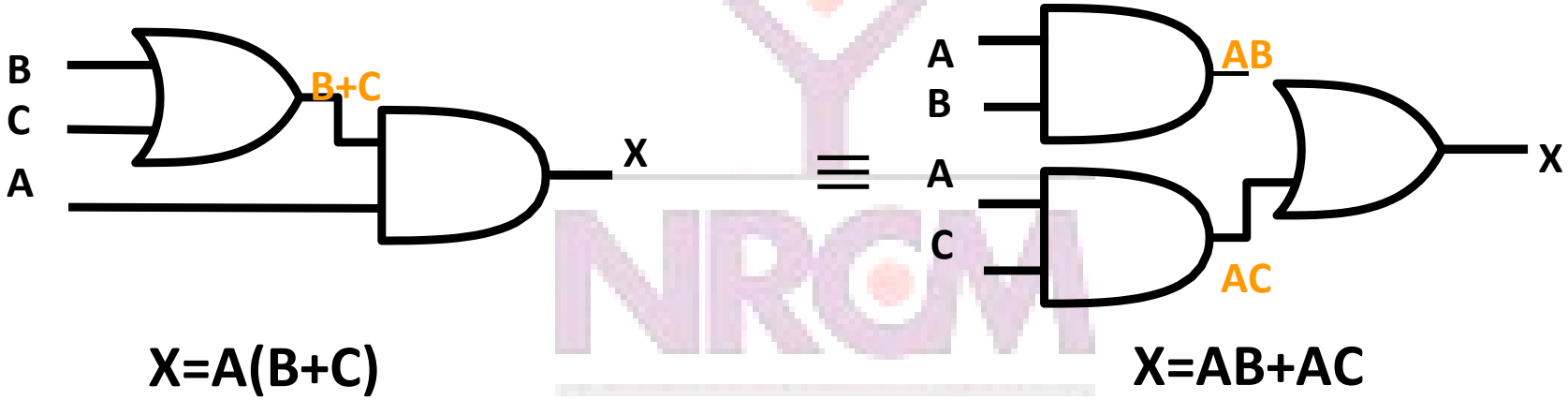
- The *associative law of addition* for 3 variables is written as: *A+(B+C) = (A+B)+C*



- The *associative law of multiplication* for 3 variables is written as: *A(BC) = (AB)C*



- The *distributive law* is written for 3 variables as follows: *A(B+C) = AB + AC*



1. $A + 0 = A$

2. $A + 1 = 1$

3. $A \bullet 0 = 0$

4. $A \bullet 1 = A$

5. $A + A = A$

6. $A + \overline{A} = 1$

7. $A \bullet A = A$

8. $A \bullet \overline{A} = 0$

9. $\overline{\overline{A}} = A$

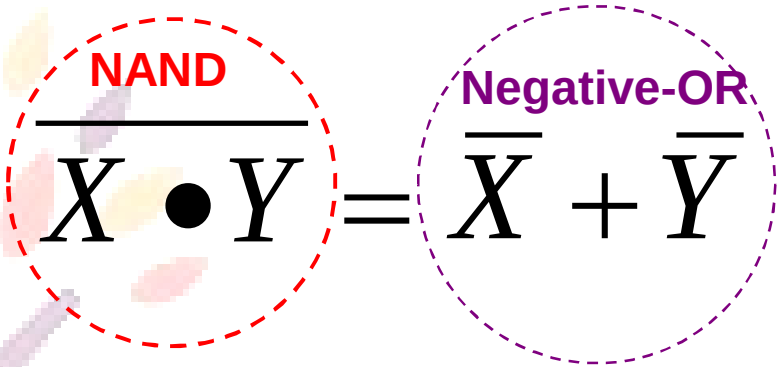
10. $A + AB = A$

11. $A + \overline{A}B = A + B$

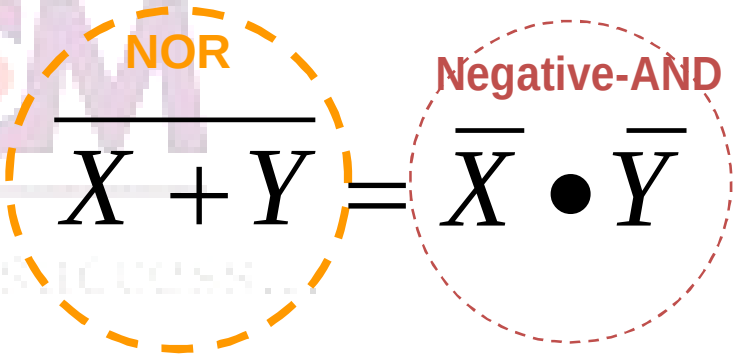
12. $(A + B)(A + C) = A + BC$

- DeMorgan's theorems provide mathematical verification of:
  - the equivalency of the NAND and negative-OR gates
  - the equivalency of the NOR and negative-AND gates.

- The complement of two or more ANDed variables is equivalent to the OR of the complements of the individual variables.



- The complement of two or more ORed variables is equivalent to the AND of the complements of the individual variables.



- Apply DeMorgan's theorems to the expressions:

$$\overline{X \bullet Y \bullet Z}$$

$$\overline{X + Y + Z}$$

$$\overline{\overline{X} + \overline{Y} + \overline{Z}}$$

$$\overline{\overline{W} \bullet \overline{X} \bullet \overline{Y} \bullet \overline{Z}}$$

- Apply DeMorgan's theorems to the expressions:

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

$$\overline{ABC+DEF}$$

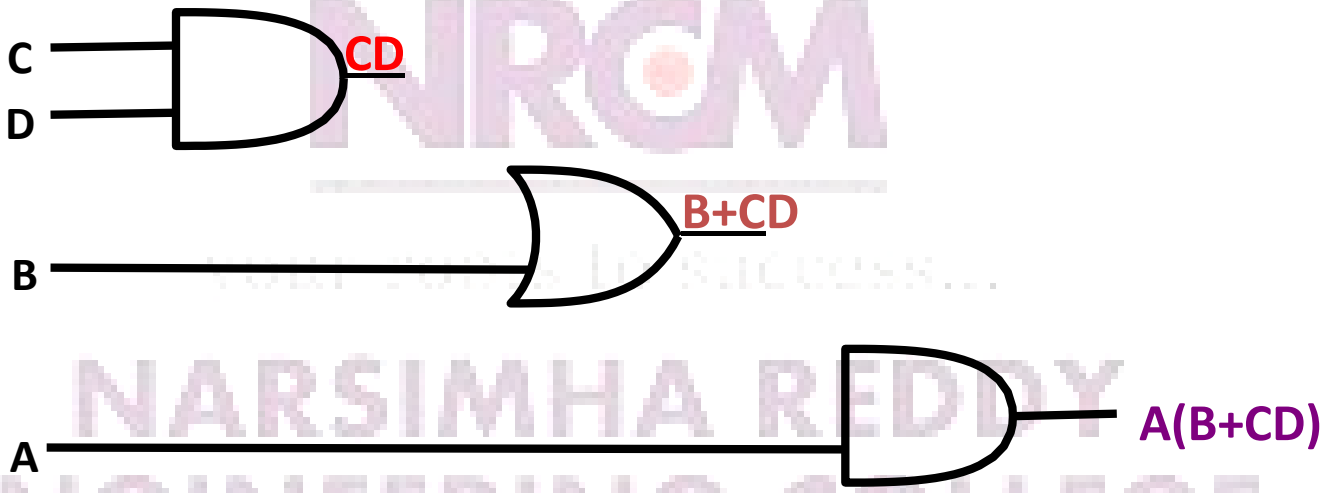
$$\overline{A\overline{B}+\overline{C}D+EF}$$

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

- Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates
  - so that the output can be determined for various combinations of input values.

Boolean Expression for a Logic Circuit

- To derive the Boolean expression for a given logic circuit, begin at the left-most inputs and work toward the final output, writing the expression for each gate.



Constructing a Truth Table for a Logic Circuit

- Once the Boolean expression for a given logic circuit has been determined, a truth table that shows the output for all possible values of the input variables can be developed.
  - Let's take the previous circuit as the example:

    A(B+CD)
  - There are four variables, hence 16 (2^4) combinations of values are possible.

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression
  - To evaluate the expression *A(B+CD)*, first find the values of the variables that make the expression equal to 1 (using the rules for Boolean add & mult).
  - In this case, the expression equals 1 only if A=1 and B+CD=1 because

$$A(B+CD) = 1 \cdot 1 = 1$$

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression (cont')
  - Now, determine when *B+CD* term equals 1.
  - The term *B+CD=1* if either *B=1* or *CD=1* or if both *B* and *CD* equal 1 because

$$B+CD = 1+0 = 1$$

$$B+CD = 0+1 = 1$$

$$B+CD = 1+1 = 1$$

- The term *CD=1* only if *C=1* and *D=1*

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression (cont')
  - Summary:
  - *A(B+CD)=1*
    - When *A=1* and *B=1* regardless of the values of *C* and *D*
    - *When A=1* and *C=1* and *D=1* regardless of the value of *B*
  - The expression *A(B+CD)=0* for all other value combinations of the variables.

Constructing a Truth Table for a Logic Circuit

- Putting the results in truth table format

A(B+CD)=1

When A=1 and B=1 regardless of the values of C and D

When A=1 and C=1 and D=1 regardless of the value of B

INPUTS				OUTPUT
A	B	C	D	A(B+CD)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	1	1

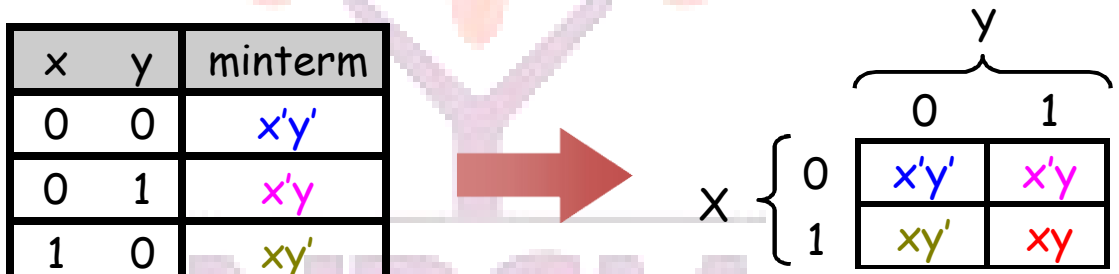
Karnaugh Maps

- Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:
  - minimal sum of products (MSP)
  - minimal product of sums (MPS)
- Goal of the simplification.
  - There are a minimal number of product/sum terms
  - Each term has a minimal number of literals
- Circuit-wise, this leads to a *minimal* two-level implementation

Re-arranging the Truth Table

- A two-variable function has four possible minterms. We can re-arrange
  these minterms into a Karnaugh map

x	y	minterm
0	0	x'y'
0	1	x'y
1	0	xy'
1	1	xy



- Now we can easily see which minterms contain common literals
  - Minterms on the left and right sides contain y' and y respectively
  - Minterms in the top and bottom rows contain x' and x respectively

Karnaugh Map Simplifications

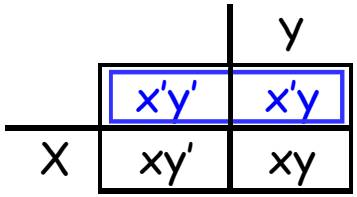
- Imagine a two-variable sum of minterms:

$$x'y' + x'y$$

- Both of these minterms appear in the top row of a Karnaugh map, which

  means that they both contain the literal x'

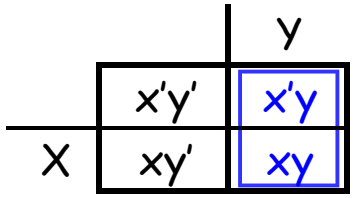
$x'y' + x'y$	$= x'(y' + y)$	[Distributive]
	$= x' \bullet 1$	[$y + y' = 1$]
	$= x'$	[$x \bullet 1 = x$]



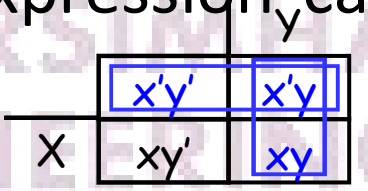
- What happens if you simplify this expression using Boolean algebra?

More Two-Variable Examples

- Another example expression is x'y + xy
  - Both minterms appear in the right side, where y is uncomplemented
  - Thus, we can reduce x'y + xy to just y



- How about x'y' + x'y + xy?
  - We have x'y' + x'y in the top row, corresponding to x'
  - There's also x'y + xy in the right side, corresponding to y
  - This whole expression can be reduced to x' + y



A Three-Variable Karnaugh Map

- For a three-variable expression with inputs x, y, z, the arrangement of minterms is more tricky:

X \ YZ	00	01	11	10
0	x′y′z′	x′y′z	x′yz	x′yz′
1	xy′z′	xy′z	xyz	xyz′

X \ YZ	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

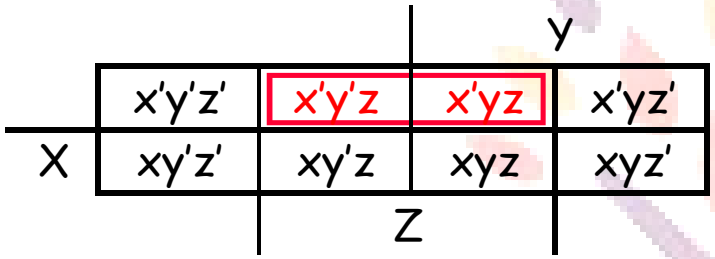
X \ Y Z	x′y′z′	x′y′z	x′yz	x′yz′
X	xy′z′	xy′z	xyz	xyz′

X \ Y Z	m_0	m_1	m_3	m_2
X	m_4	m_5	m_7	m_6

- Another way to label the K-map (use whichever you like):

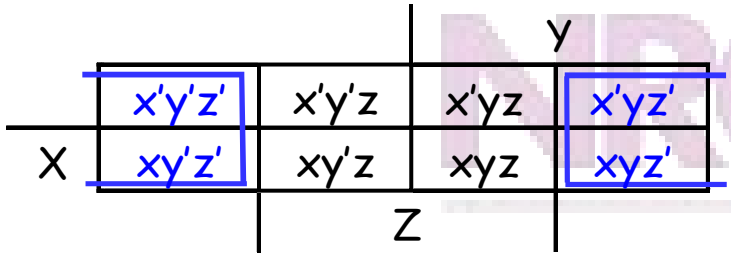
Why the funny ordering?

- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out



$$
\begin{aligned}
& x'y'z + x'yz \\
= \; & x'z(y' + y) \\
= \; & x'z \bullet 1 \\
= \; & x'z
\end{aligned}
$$

- "Adjacency" includes wrapping around the left and right sides:

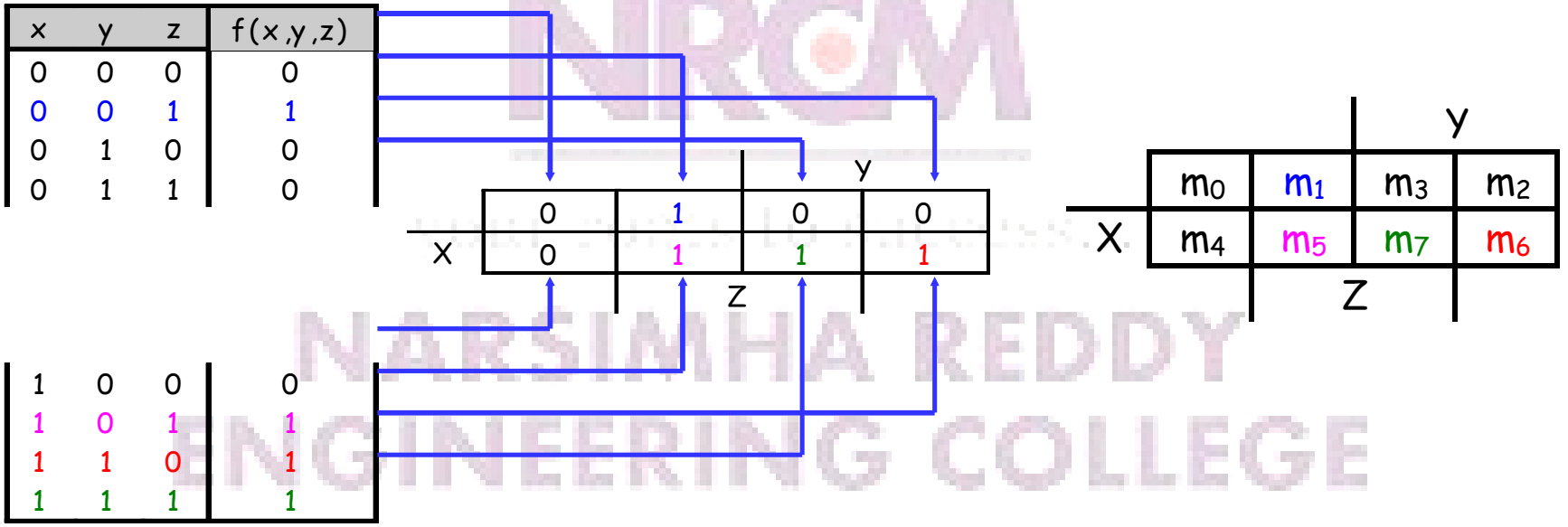


$$
\begin{aligned}
& x'y'z' + xy'z' + x'yz' + xyz' \\
= \; & z'(x'y' + xy' + x'y + xy) \\
= \; & z'(y'(x' + x) + y(x' + x)) \\
= \; & z'(y'+y) \\
= \; & z'
\end{aligned}
$$

- We'll use this property of adjacent squares to do our simplifications.

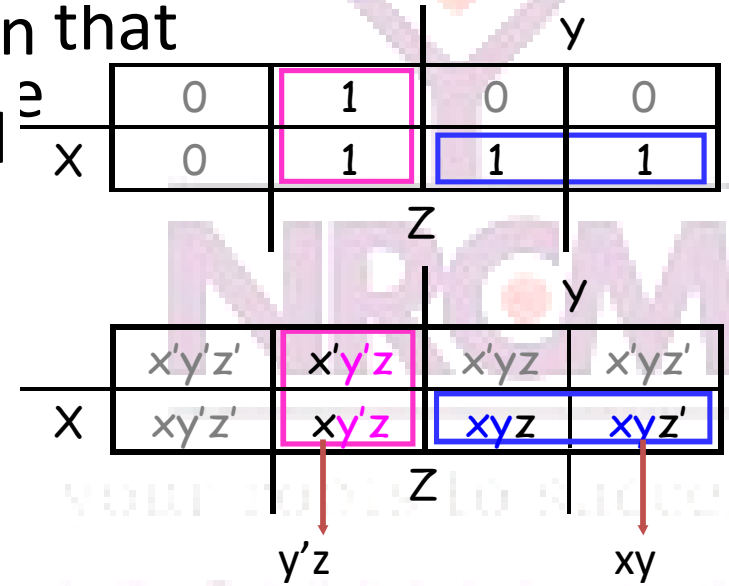
K-maps From Truth Tables

- We can fill in the K-map directly from a truth table
  - The output in row *i* of the table goes into square m_i of the K-map
  - Remember that the rightmost columns of the K-map are "switched"



Reading the MSP from the K-map

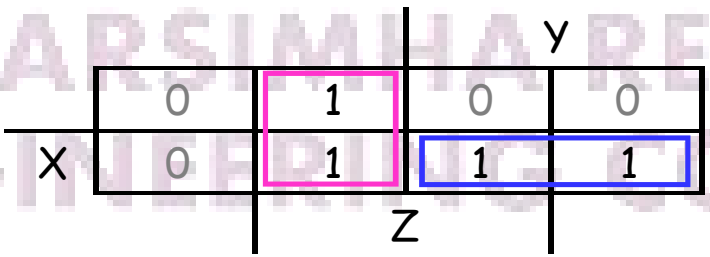
- You can find the minimal SoP expression
  - Each rectangle corresponds to one product term
  - The product is determined by finding the common literals in that rectangle



F(x,y,z)= y'z + xy

Grouping the Minterms Together

- The most difficult step is grouping together all the 1s in the K-map
  - Make rectangles around groups of one, two, four or eight 1s
  - All of the 1s in the map should be included in at least one rectangle
  - Do *not* include any of the 0s
  - Each group corresponds to one product term



For the Simplest Result

- Make as few rectangles as possible, to minimize the number of products in the final expression.
- Make each rectangle as large as possible, to minimize the number of literals in each term.
- Rectangles can be overlapped, if that makes them larger.

K-map Simplification of SoP Expressions

- Let's consider simplifying $f(x,y,z) = xy + y'z + xz$

- You should convert the expression into a sum of minterms form,
  - The easiest way to do this is to make a truth table for the function, and then read off the minterms
  - You can either write out the literals or use the minterm shorthand

- Here is the truth table and sum of minterms for our example:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$f(x,y,z) = x'y'z + xy'z + xyz' + xyz$

$= m_1 + m_5 + m_6 + m_7$

Unsimplifying Expressions

- You can also convert the expression to a sum of minterms with Boolean algebra
  - Apply the distributive law in reverse to add in missing variables.
  - Very few people actually do this, but it's occasionally useful.

$$
\begin{aligned}
xy + y'z + xz &= (xy \bullet 1) + (y'z \bullet 1) + (xz \bullet 1) \\
&= (xy \bullet (z' + z)) + (y'z \bullet (x' + x)) + (xz \bullet (y' + y)) \\
&= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\
&= xyz' + xyz + x'y'z + xy'z \\
&= m_1 + m_5 + m_6 + m_7
\end{aligned}
$$

- In both cases, we're actually "unsimplifying" our example expression
  - The resulting expression is larger than the original one!
  - But having all the individual minterms makes it easy to combine them together with the K-map

Making the Example K-map

- In our example, we can write f(x,y,z) in two equivalent ways

f(x,y,z) = x'y'z + xy'z + xyz' + xyz

		y		
	x'y'z'	x'y'z	x'yz	x'yz'
X	xy'z'	xy'z	xyz	xyz'
		Z		

f(x,y,z) = $m_1 + m_5 + m_6 + m_7$

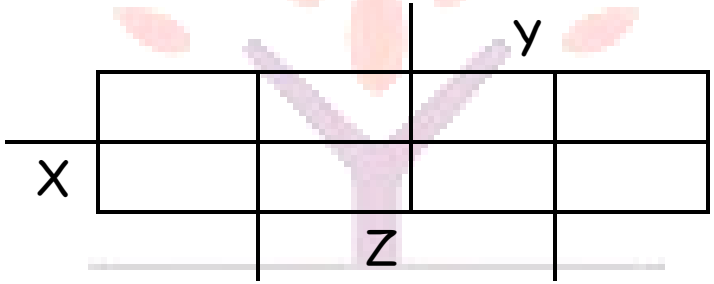
		y		
	m_0	m_1	m_3	m_2
X	m_4	m_5	m_7	m_6
		Z		

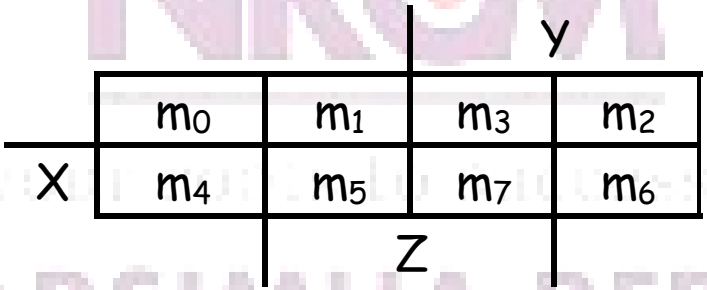
		y		
	0	1	0	0
X	0	1	1	1
		Z		

- In either case, the resulting K-map is shown below

Practice K-map 1

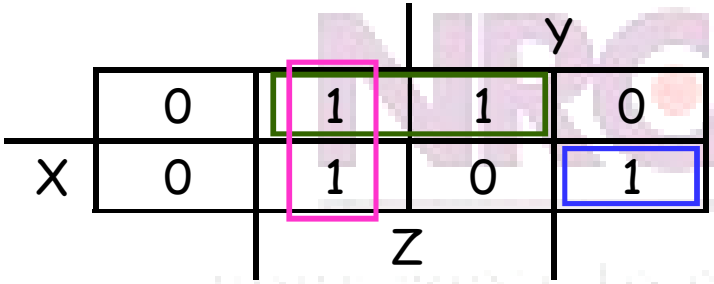
- Simplify the sum of minterms $m_1 + m_3 + m_5 + m_6$





Solutions for Practice K-map 1

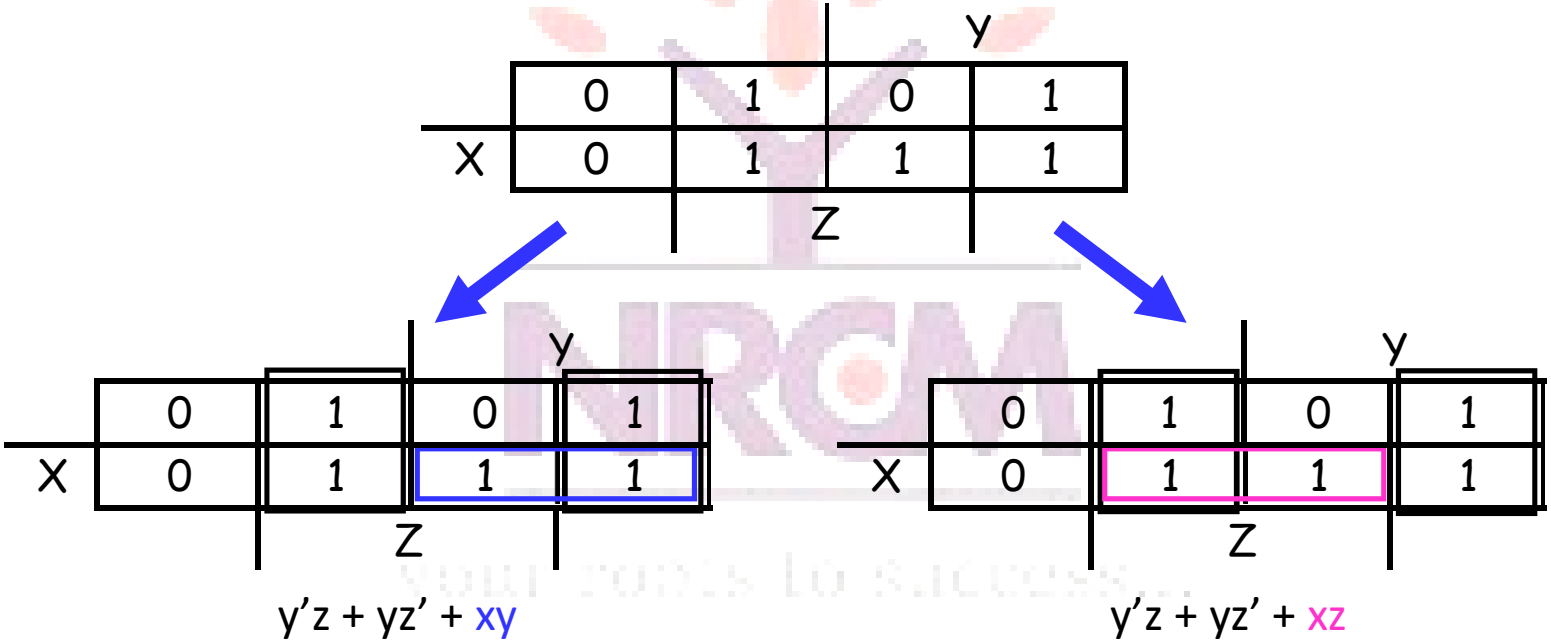
- Here is the filled in K-map, with all groups shown
  - The magenta and green groups overlap, which makes each of them as large as possible
  - Minterm m_6 is in a group all by its lonesome



- The final MSP here is x'z + y'z + xyz'

K-maps can be tricky!

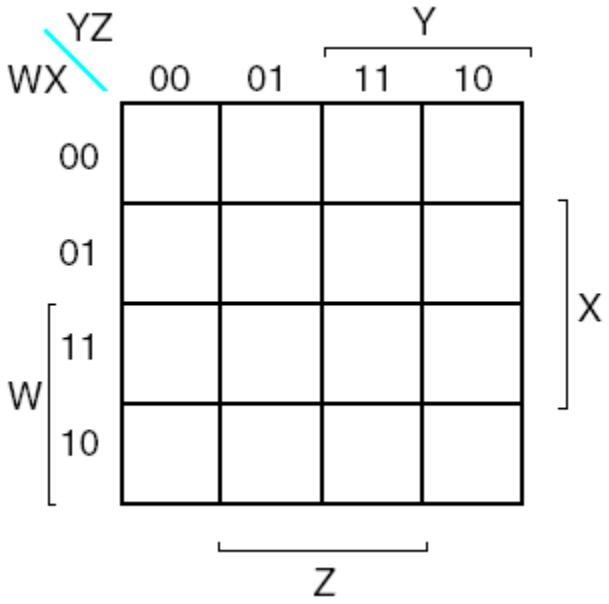
- There may not necessarily be a *unique* MSP. The K-map below yields two valid and equivalent MSPs, because there are two possible ways to include minterm m_7



- Remember that overlapping groups is possible, as shown above

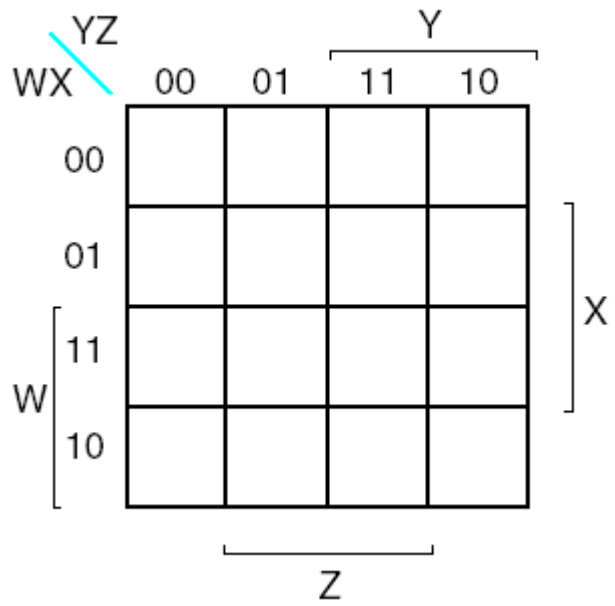
Four-variable K-maps – f(W,X,Y,Z)

- We can do four-variable expressions too!
  - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
  - Again, this ensures that adjacent squares have common literals



- Grouping minterms is similar to the three-variable case, but:
  - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
  - You can wrap around *all four* sides

Four-variable K-maps



		y	
w'x'y'z'	w'x'y'z	w'x'yz	w'x'yz'
w'xy'z'	w'xy'z	w'xyz	w'xyz'
wxy'z'	wxy'z	wxyz	wxyz'
wx'y'z'	wx'y'z	wx'yz	wx'yz'

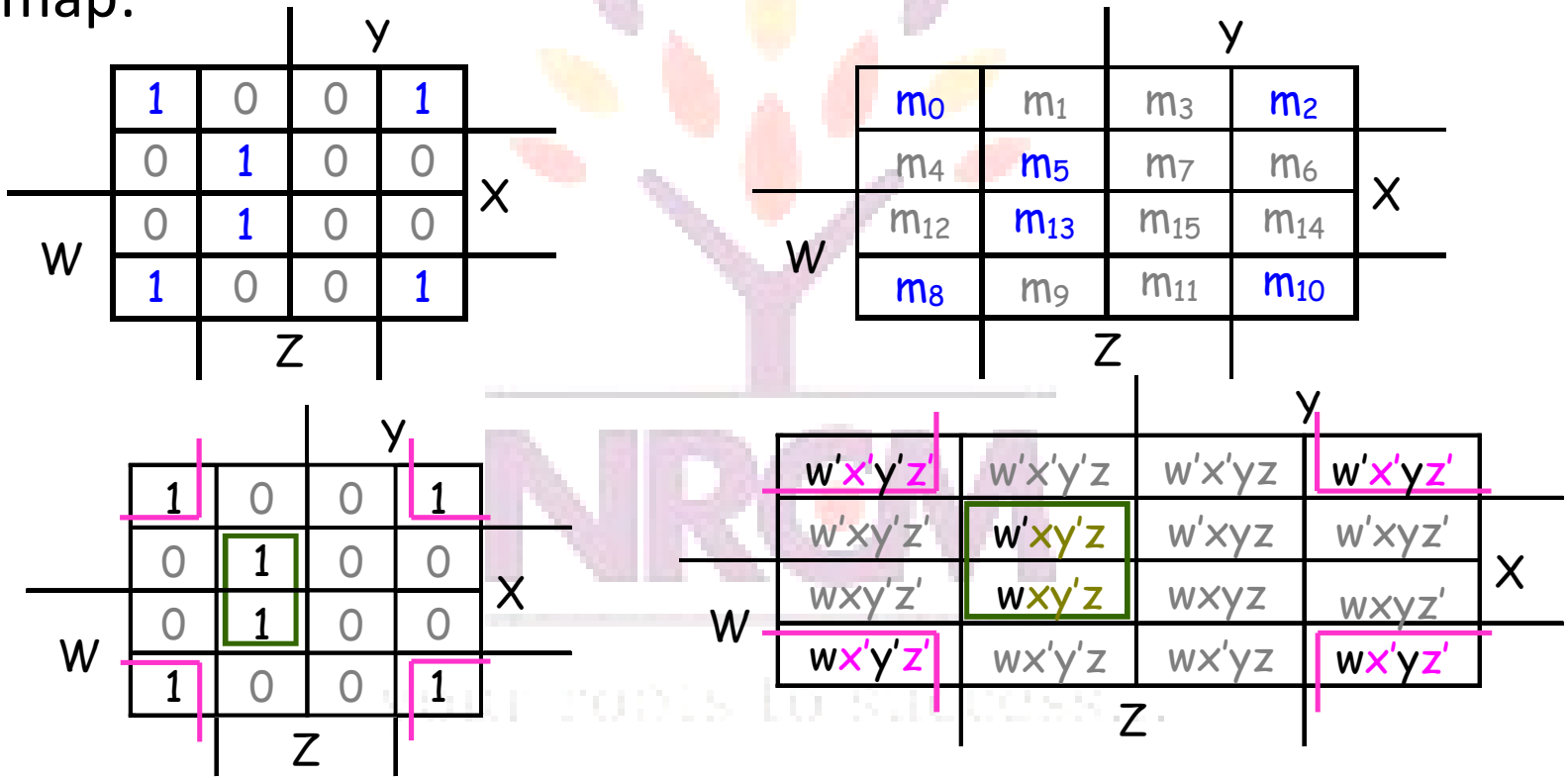
W, X, Z

		y	
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}

W, X, Z

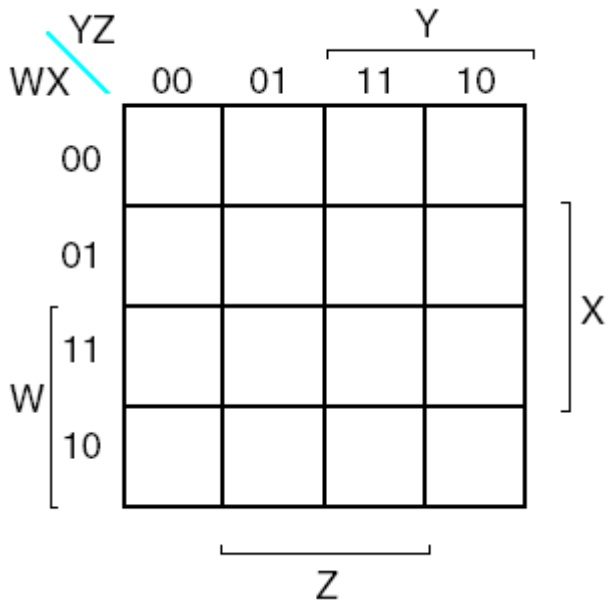
Example: Simplify $m_0+m_2+m_5+m_8+m_{10}+m_{13}$

- The expression is already a sum of minterms, so here's the K-map:

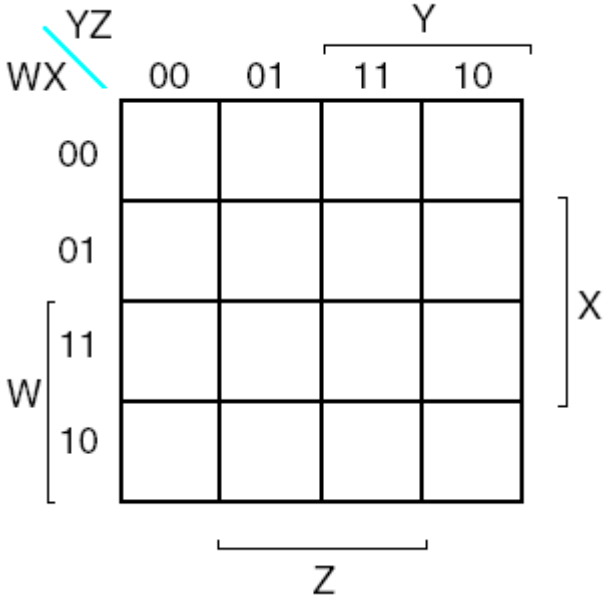


- We can make the following groups, resulting in the MSP x'z' + xy'z

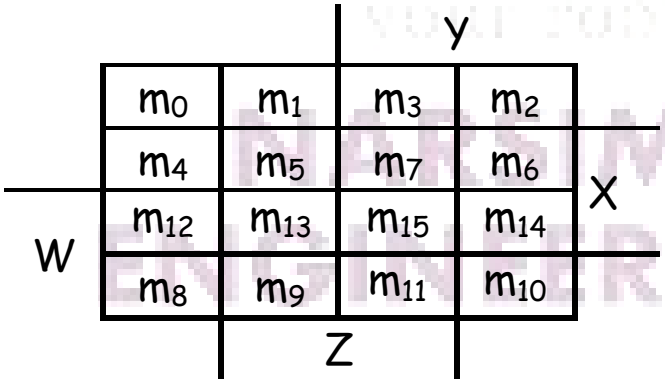
Five-variable K-maps – f(V,W,X,Y,Z)

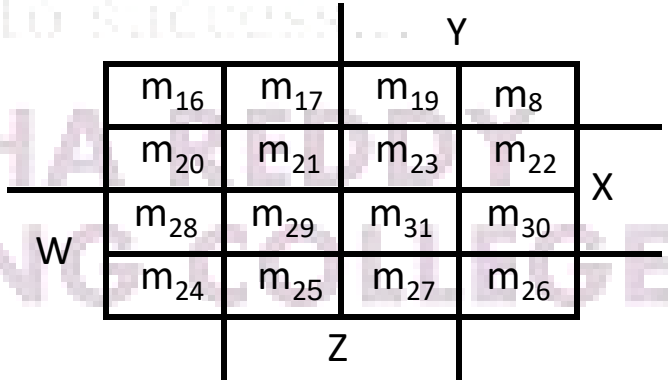


V= 0

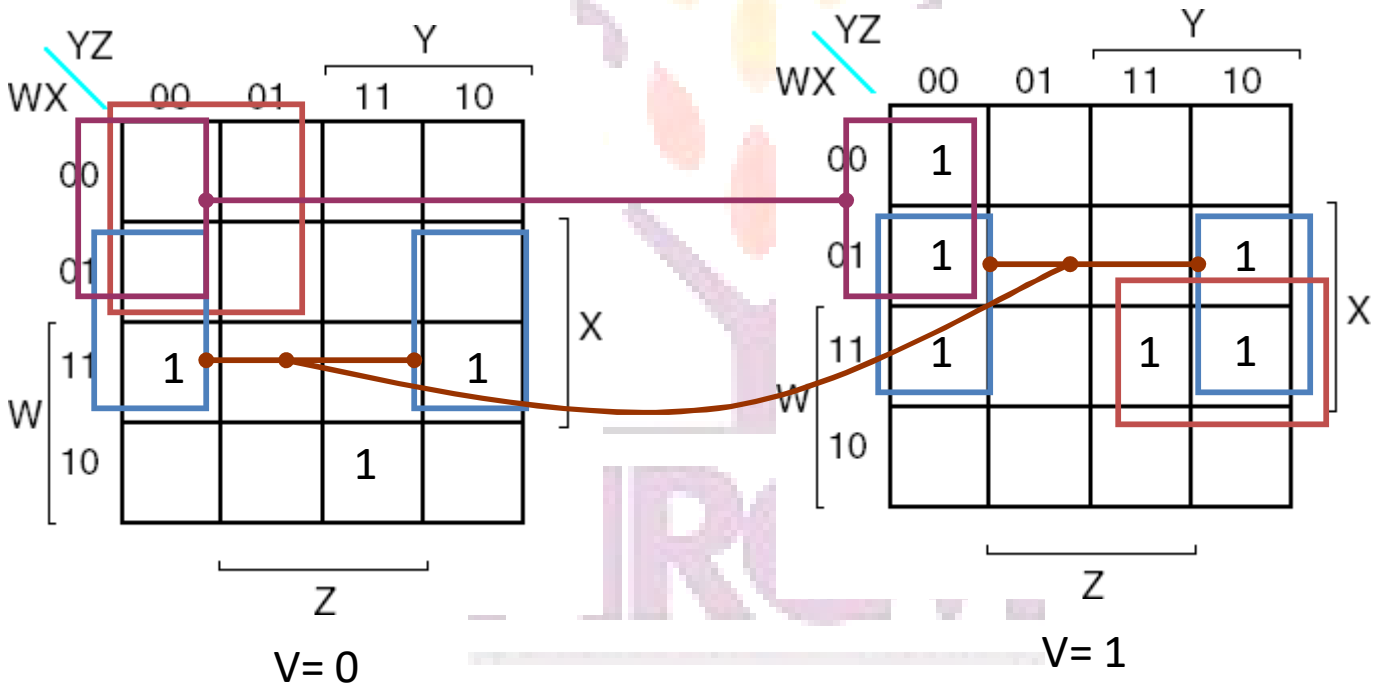


V= 1





Simplify f(V,W,X,Y,Z)=Σm(0,1,4,5,6,11,12,14,16,20,22,28,30,31)



f = XZ’ Σm(4,6,12,14,20,22,28,30)
+ V’W’Y’ Σm(0,1,4,5)
+ W’Y’Z’ Σm(0,4,16,20)
+ VWXY Σm(30,31)
+ V’WX’YZ m11

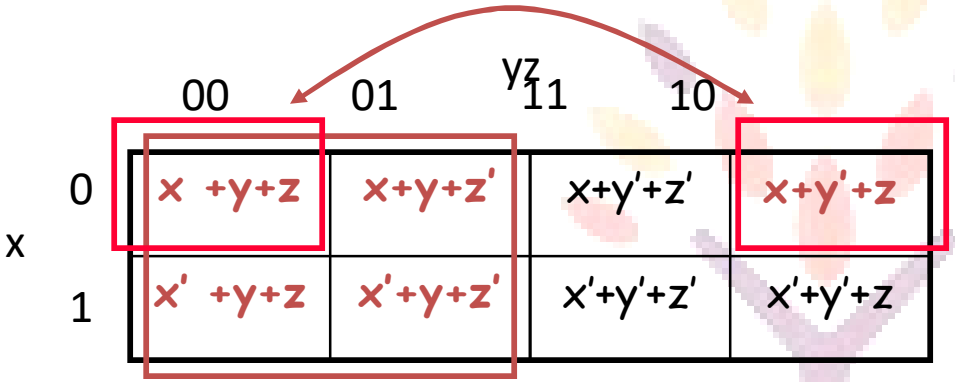
PoS Optimization

- Maxterms are grouped to find minimal PoS expression

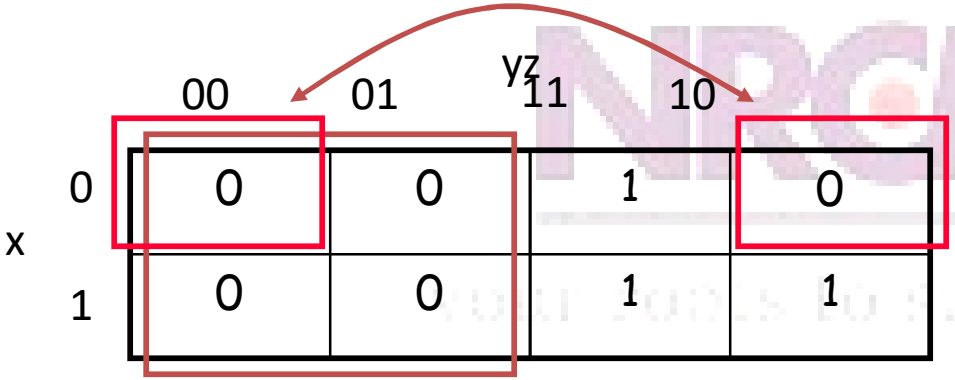
x \ yz	00	01	11	10
0	x +y+z	x+y+z′	x+y′+z′	x+y′+z
1	x′ +y+z	x′+y+z′	x′+y′+z′	x′+y′+z

Po

- F(W,X,Y,Z)= ∏ M(0,1,2,4,5)



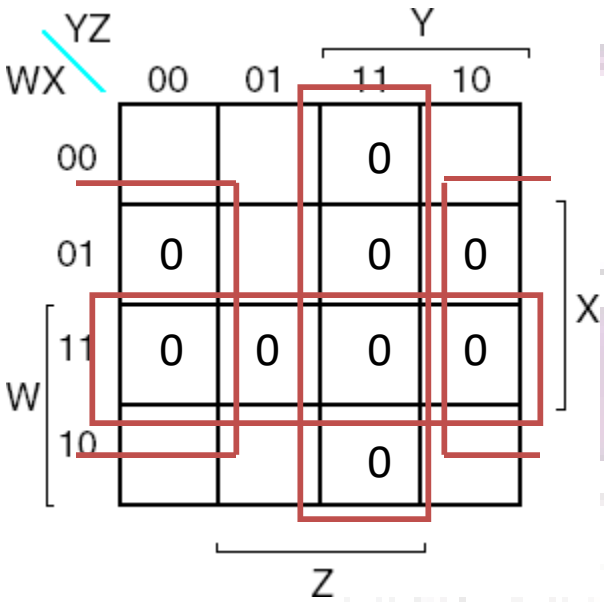
F(W,X,Y,Z)= Y . (X + Z)



Po

F(W,X,Y,Z)= Σm(0,1,2,5,8,9,10)

= ∏ M(3,4,6,7,11,12,13,14,15)



F(W,X,Y,Z)= (W' + X')(Y' + Z')(X' + Z)

Or,

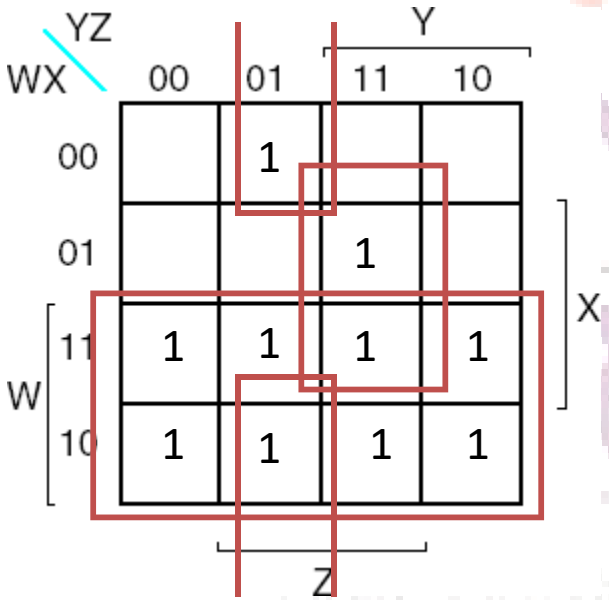
F(W,X,Y,Z)= X'Y' + X'Z' + W'Y'Z

Which one is the minimal one?

Sol

F(W,X,Y,Z)= ∏ M(0,2,3,4,5,6)

= Σm(1,7,8,9,10,11,12,13,14,15)



F(W,X,Y,Z)= W + XYZ + X'Y'Z

I d

- You don't always need all 2^n input combinations in an n-variable function
  - If you can guarantee that certain input combinations never occur
  - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

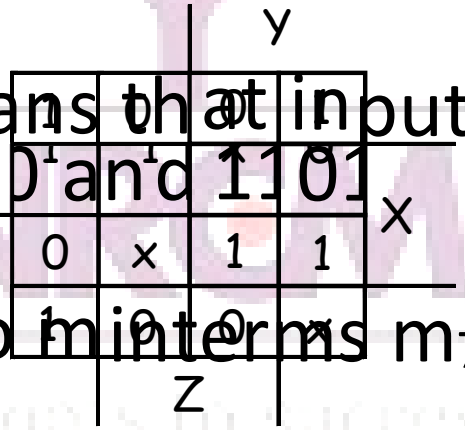
- Within a K-map, each X can be considered as either 0 or 1. You should pick the interpretation that allows for the most simplification.

Pra

- Find a MSP for

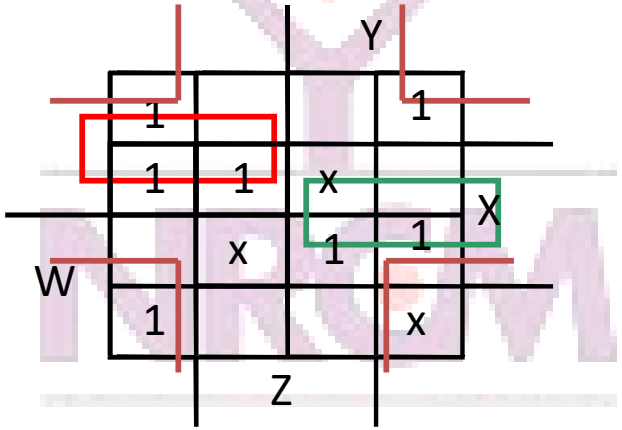
$$f(w,x,y,z) = \Sigma m(0,2,4,5,8,14,15),\ d(w,x,y,z) = \Sigma m(7,10,13)$$

This notation means that input combinations wxyz = 0111, 1010 and 1101 (corresponding to minterms m_7, m_{10} and m_{13}) are unused.



Solutions for Practice K-map

- Find a MSP for: $f(w,x,y,z) = \Sigma m(0,2,4,5,8,14,15)$, $d(w,x,y,z) = \Sigma m(7,10,13)$



$f(w,x,y,z) = x'z' + w'xy' + wxy$

K-map Summary

- K-maps are an alternative to algebra for simplifying expressions
  - The result is a MSP/MPS, which leads to a minimal two-level circuit
  - It's easy to handle don't-care conditions
  - K-maps are really only good for manual simplification of small expressions...
- Things to keep in mind:
  - Remember the correct order of minterms/maxterms on the K-map
  - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
  - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
  - There may be more than one valid solution

Tabulation Method – **STEP 1**

1. Partition Prime Implicants (or minterms) According to Number of 1's

2. Check Adjacent Classes for Cube Merging Building a New List

3. If Entry in New List Covers Entry in Current List – Disregard Current List Entry

4. If Current List = New List
   HALT
   Else
   Current List ← New List
   New List ← NULL
   Go To Step 1

STEP 1 – EXAMPLE

$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$

Minterm	Cube			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
8	1	0	0	0
3	0	0	1	1
5	0	1	0	1
10	1	0	1	0
11	1	0	1	1
13	1	1	0	1
15	1	1	1	1

STEP 1 – EXAMPLE

$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube			
0,1	0	0	0	-
0,2	0	0	-	0
0,8	-	0	0	0
1,3	0	0	-	1
1,5	0	-	0	1
2,3	0	0	1	-
2,10	-	0	1	0
8,10	1	0	-	0
3,11	-	0	1	1
5,13	-	1	0	1
10,11	1	0	1	-
11,15	1	-	1	1
13,15	1	1	-	1

STEP 1 – EXAMPLE

$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube				
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,3	0	0	-	1	✓
1,5	0	-	0	1	
2,3	0	0	1	-	✓
2,10	-	0	1	0	✓
8,10	1	0	-	0	✓
3,11	-	0	1	1	✓
5,13	-	1	0	1	
10,11	1	0	1	-	✓
11,15	1	-	1	1	
13,15	1	1	-	1	

Minterm	Cube			
0,1,2,3	0	0	-	-
0,8,2,10	-	0	-	0
2,3,10,11	-	0	1	-

STEP 1 – EXAMPLE

$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube				
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,3	0	0	-	1	✓
1,5	0	-	0	1	PI=D
2,3	0	0	1	-	✓
2,10	-	0	1	0	✓
8,10	1	0	-	0	✓
3,11	-	0	1	1	✓
5,13	-	1	0	1	PI=E
10,11	1	0	1	-	✓
11,15	1	-	1	1	PI=F
13,15	1	1	-	1	PI=G

Minterm	Cube				
0,1,2,3	0	0	-	-	PI=A
0,8,2,10	-	0	-	0	PI=C
2,3,10,11	-	0	1	-	PI=B

$f^{on} = \{A,B,C,D,E,F,G\} = \{00\text{--}, \text{-}01\text{-}, \text{-}0\text{-}0, 0\text{-}01, \text{-}101, 1\text{-}11, 11\text{-}1\}$

STEP 2 – Construct Cover Table

- PIs Along Vertical Axis (in order of # of literals)
- Minterms Along Horizontal Axis

	0	1	2	3	5	8	10	11	13	15
A	x	x	x	x						
B			x	x			x	x		
C	x		x			x	x			
D		x			x					
E					x				x	
F								x		x
G									x	x

NOTE: Table 4.2 in book is incomplete

STEP 2 – Finding the Minimum Cover

- Extract All Essential Prime Implicants, EPI
- EPIs are the PI for which a Single x Appears in a Column

	0	1	2	3	5	8	10	11	13	15
A	x	x	x	x						
B			x	x			x	x		
C	x		x			x	x			
D		x			x					
E					x				x	
F								x		x
G									x	x

- *C* is an EPI so: $f^{on}=\{C, \ldots\}$
- Row C and Columns 0, 2, 8, and 10 can be Eliminated Giving Reduced Cover Table
- Examine Reduced Table for New EPIs

STEP 3 – Reduced Table

Distinguished Column

	0	1	2	3	5	8	10	11	13	15
A	x	x	x	x						
B			x	x			x	x		
C	x		x			x	x			
D		x			x					
E					x				x	
F								x		x
G									x	x

Essential row

	1	3	5	11	13	15
A	x	x				
B		x		x		
D	x		x			
E			x		x	
F				x		x
G					x	x

- The Row of an EPI is an *Essential row*
- The Column of the Single x in the *Essential Row* is a *Distinguished Column*

Row and Column Dominance

- If Row P has x's Everywhere Row Q Does
  Then Q Dominates P if P has fewer x's

- If Column *i* has x's Everywhere *j* Does
  Then *j* Dominates *i* if *i* has fewer x's

- If Row P is equal to Row Q and Row Q does not cost more than Row P, eliminate Row P, or if Row P is dominated by Row Q and Row Q Does not cost more than Row P, eliminate Row P

- If Column *i* is equal to Column *j*, eliminate Column *i* or if Column *i* dominates Column *j*, eliminate Column *i*

STEP 3 – The Reduced Cover Table

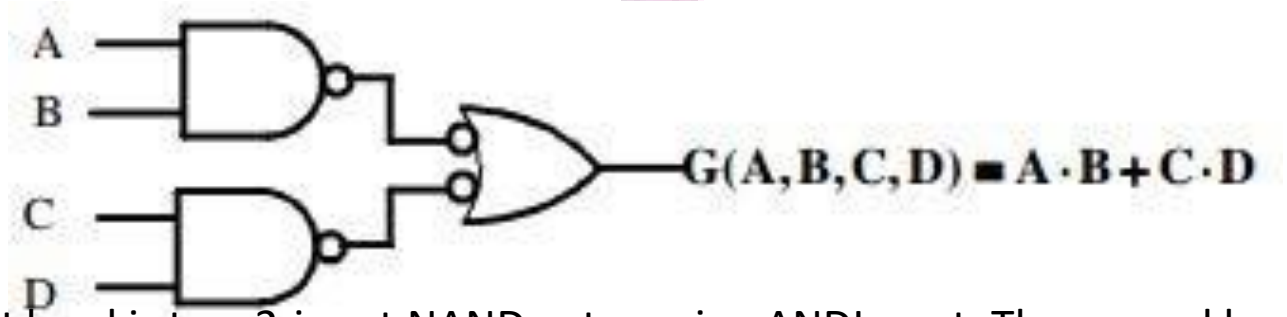
- Initially, Columns 0, 2, 8 and 10 Removed

	1	3	5	11	13	15
A	x	x				
B		x		x		
D	x		x			
E			x		x	
F				x		x
G					x	x

- No EPIs are Present
- No Row Dominance Exists
- No Column Dominance Exists
- This is *Cyclic Cover* Table
- Must Solve Exactly OR Use a Heuristic

NAND Function Implementation

- NAND gates can implement a simplified Sum-ofProducts form. Constructing two level NAND-NAND gate circuits



The first level is two 2-input NAND gates using ANDInvert. The second level is one 2-input NAND gate using Invert-OR. Using the NAND relationship, we

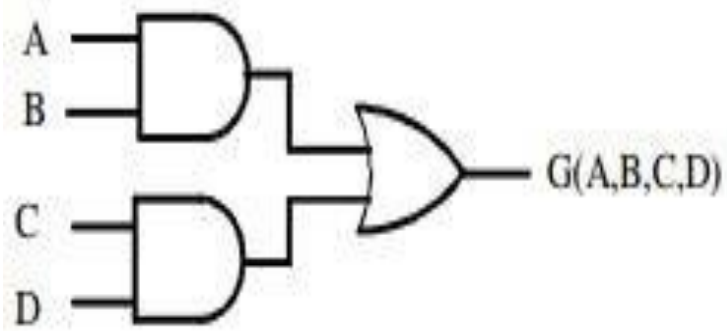
$$G(A,B,C,D) = \overline{\overline{A\cdot B}\cdot\overline{C\cdot D}}$$
$$= \overline{\overline{A\cdot B}} + \overline{\overline{C\cdot D}}$$
$$= A\cdot B + C\cdot D$$

Logic Family Characteristics

- **Complementary metal oxide semiconductor (CMOS)**
  - most widely used family for large-scale devices
  - combines high speed with low power consumption
  - usually operates from a single supply of 5 – 15 V
  - excellent noise immunity of about 30% of supply voltage
  - can be connected to a large number of gates (about 50)
  - many forms – some with t_{PD} down to 1 ns
  - power consumption depends on speed (perhaps 1 mW)

NAND Implementation (Cont.)

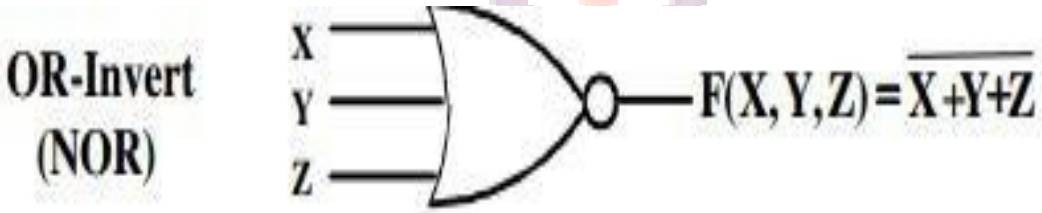
In the implementation, note that the bubbles are on opposite ends of the same line. Thus, they can be combined and deleted:



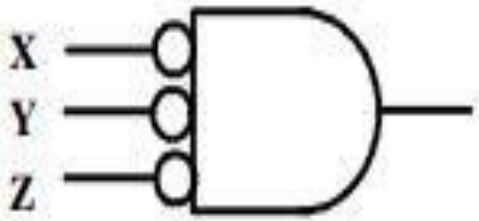
This form of the implementation is the Sum-of-Products form.

NOR Gates

The basic positive logic NOR gate (Not-OR) is denoted by the following symbol:



This is called the OR-Invert, since it is logically an OR function followed by an invert. By DeMorgan's Law we have the following Invert-AND symbol for a NOR gate:



General Implementations (Cont.)

Given a two level implementation desired, use the previous transfromations to get it into one of the below forms. Then follow the steps to transform the function to the desired form:

For Type:	Use:
AND-OR (SOP Form)	Circle 1's in the K-Map and minimize (Also use for NAND-NAND)
AND-NOR (SOP complemented)	Circle 0's in the K-Map and minimize
OR-AND (POS Form)	Circle 0's in the K-Map and minimize SOP. Use DeMorgan's to transform to POS. (Also use for NOR-NOR)
OR-NAND (POS complemented)	Circle 1's in the K-Map and minimize SOP. Use DeMorgan's to transform to POS.

Multi-level NAND Implementations

- Add inverters in two-level implementation into the cost picture
- Attempt to “combine” inverters to reduce the term count
- Attempt to reduce literal + tem count by factoring expression into POSOP or SOPOS

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

- **Transistor-transistor logic (TTL)**
  - based on bipolar transistors
  - one of the most widely used families for small- and medium-scale devices – rarely used for VLSI
  - typically operated from 5V supply
  - typical noise immunity about 1 – 1.6 V
  - many forms, some optimised for speed, power, etc.
  - high speed versions comparable to CMOS (~ 1.5 ns)
  - low-power versions down to about 1 mW/gate

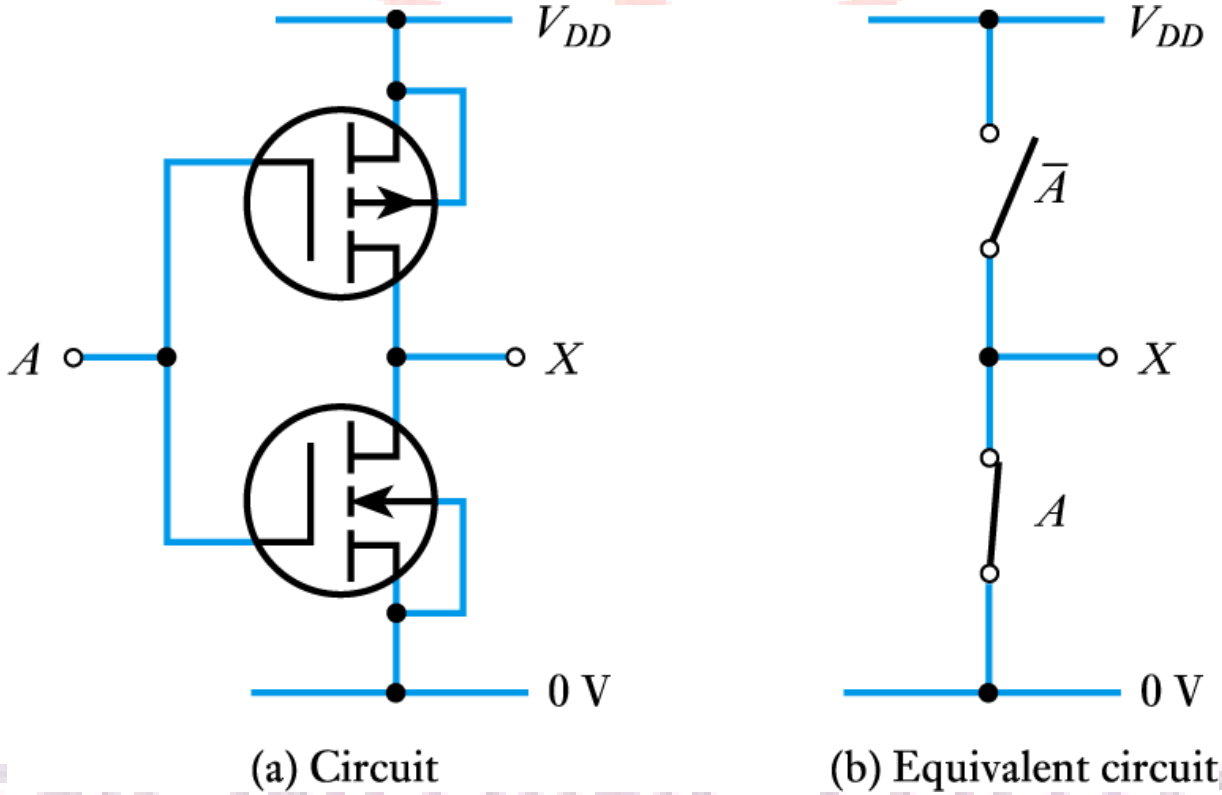
- **Emitter-coupled logic (ECL)**
  - based on bipolar transistors, but removes problems of storage time by preventing the transistors from saturating
  - very fast operation - propagation delays of 1ns or less
  - high power consumption, perhaps 60 mW/gate
  - low noise immunity of about 0.2-0.25 V
  - used in some high speed specialist applications, but now largely replaced by high speed CMOS

A Comparison of Logic Families

Parameter	CMOS	TTL	ECL
Basic gate	NAND/NOR	NAND	OR/NOR
Fan-out	>50	10	25
Power per gate (mW)	1 @ 1 MHz	1 - 22	4 - 55
Noise immunity	Excellent	Very good	Good
t_{PD} (ns)	1 - 200	1.5 – 33	1 - 4

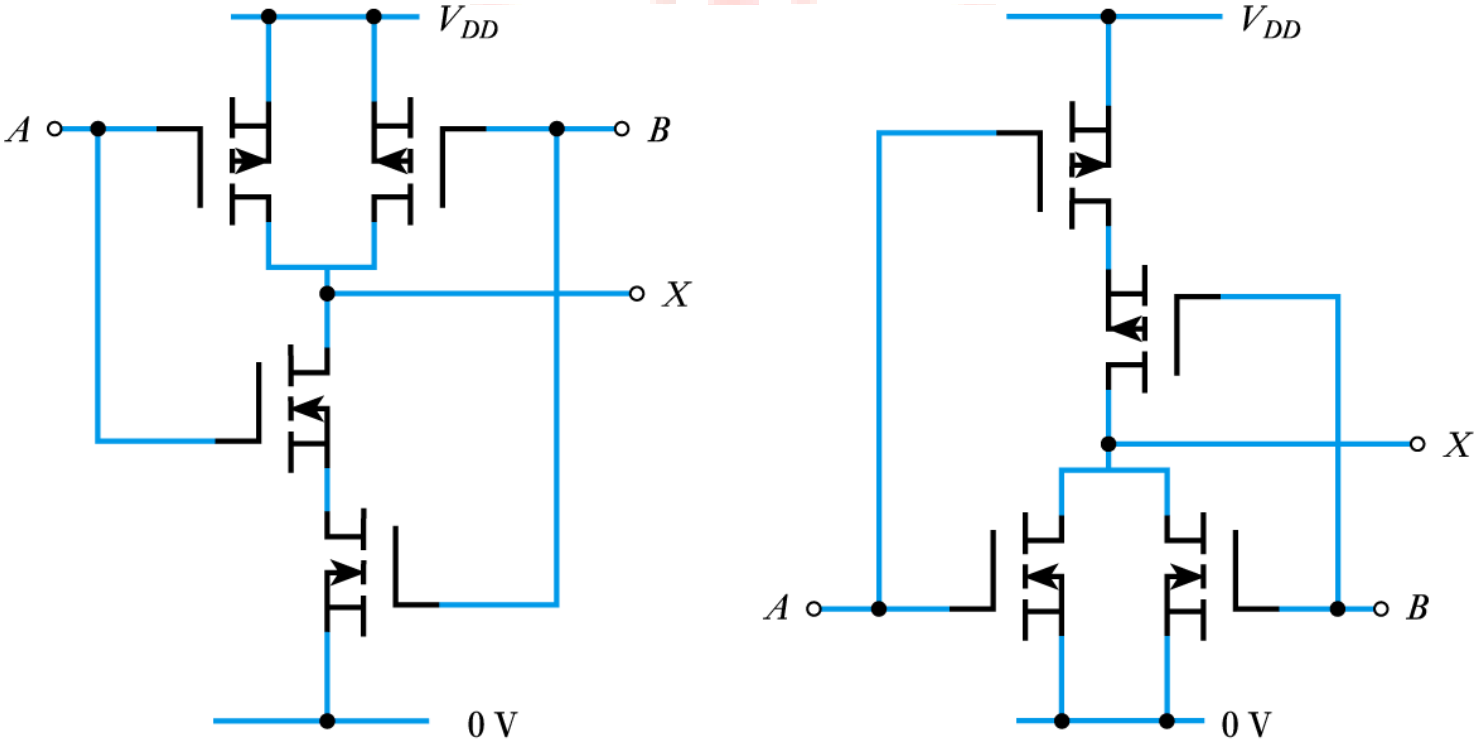
Complementary Metal Oxide Semiconductor

- **A CMOS inverter**



(a) Circuit (b) Equivalent circuit

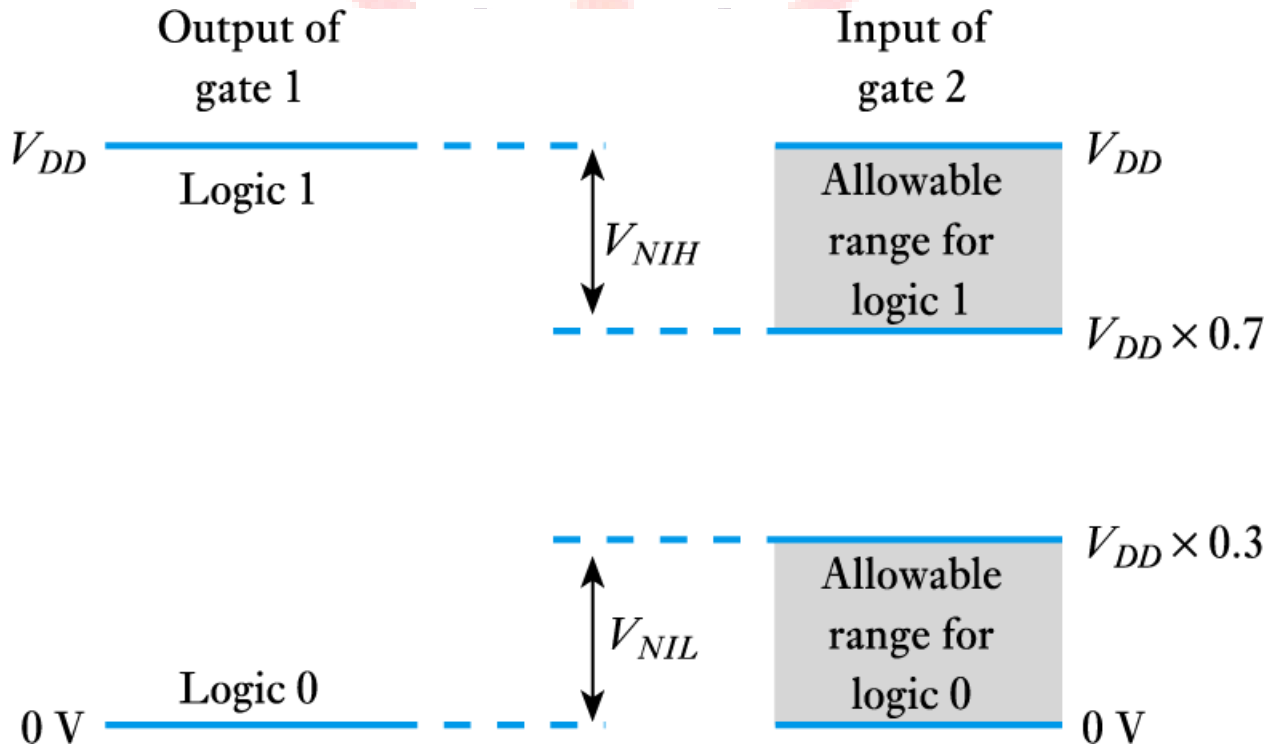
- **CMOS gates**



(a) NAND gate

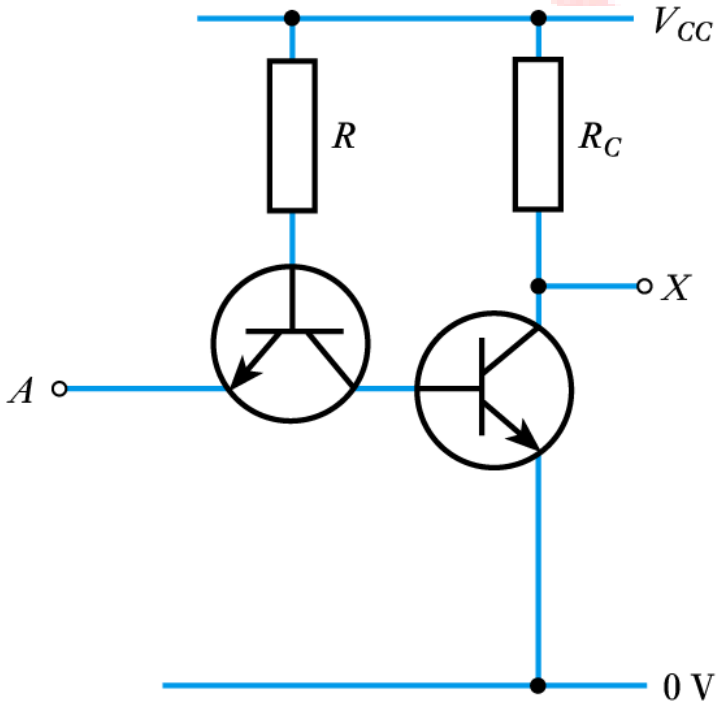
(b) NOR gate

- **CMOS logic levels and noise immunity**

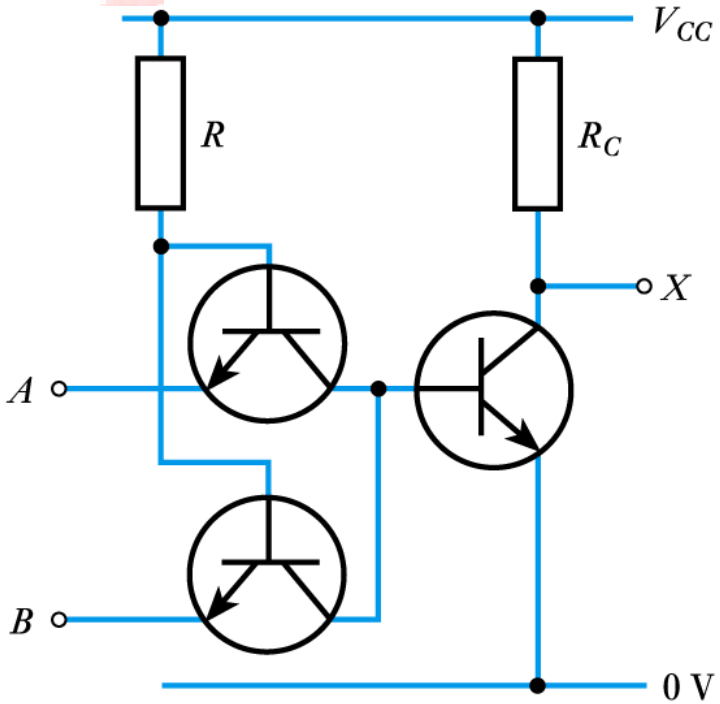


Transistor-Transistor Logic

- **Discrete TTL inverter and NAND gate circuits**

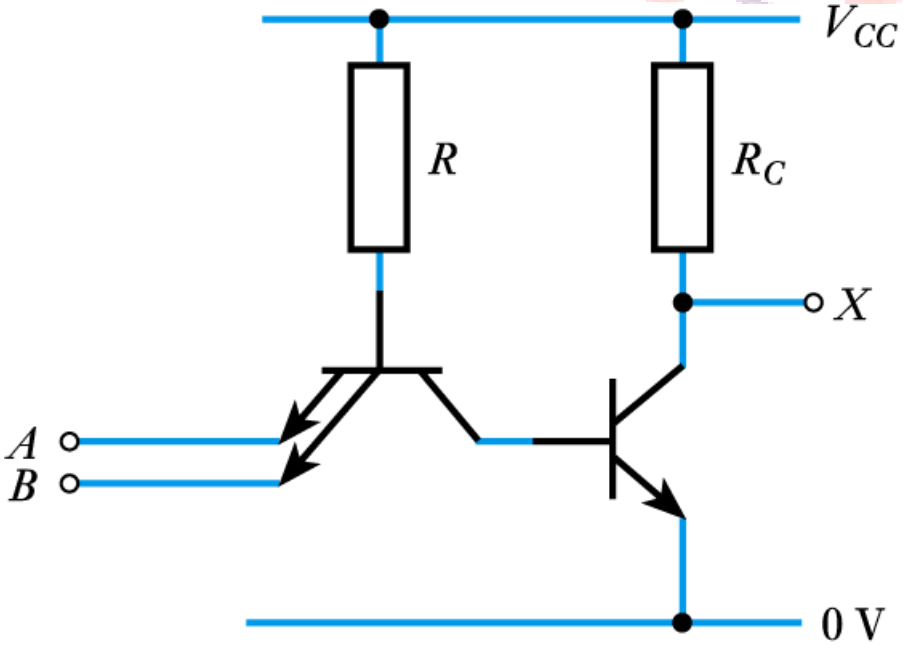


(a) A TTL inverter

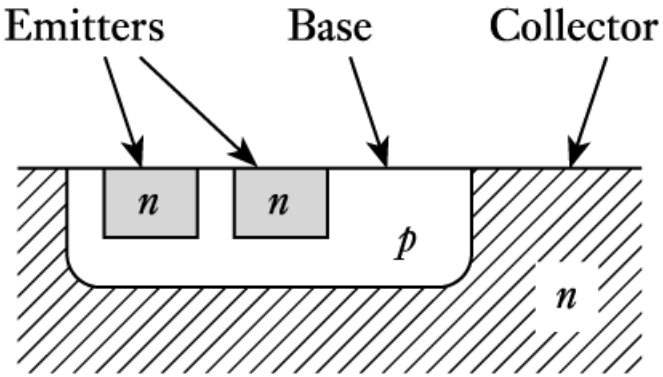


(b) A TTL NAND gate

- **A basic integrated circuit TTL NAND gate**

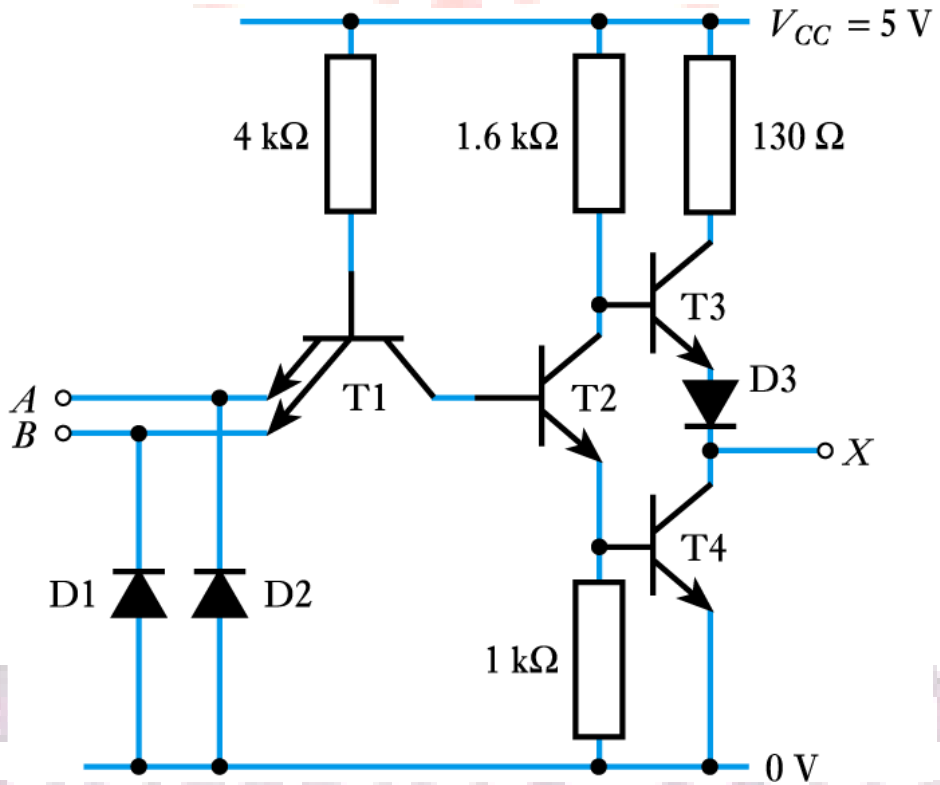


(a) Circuit

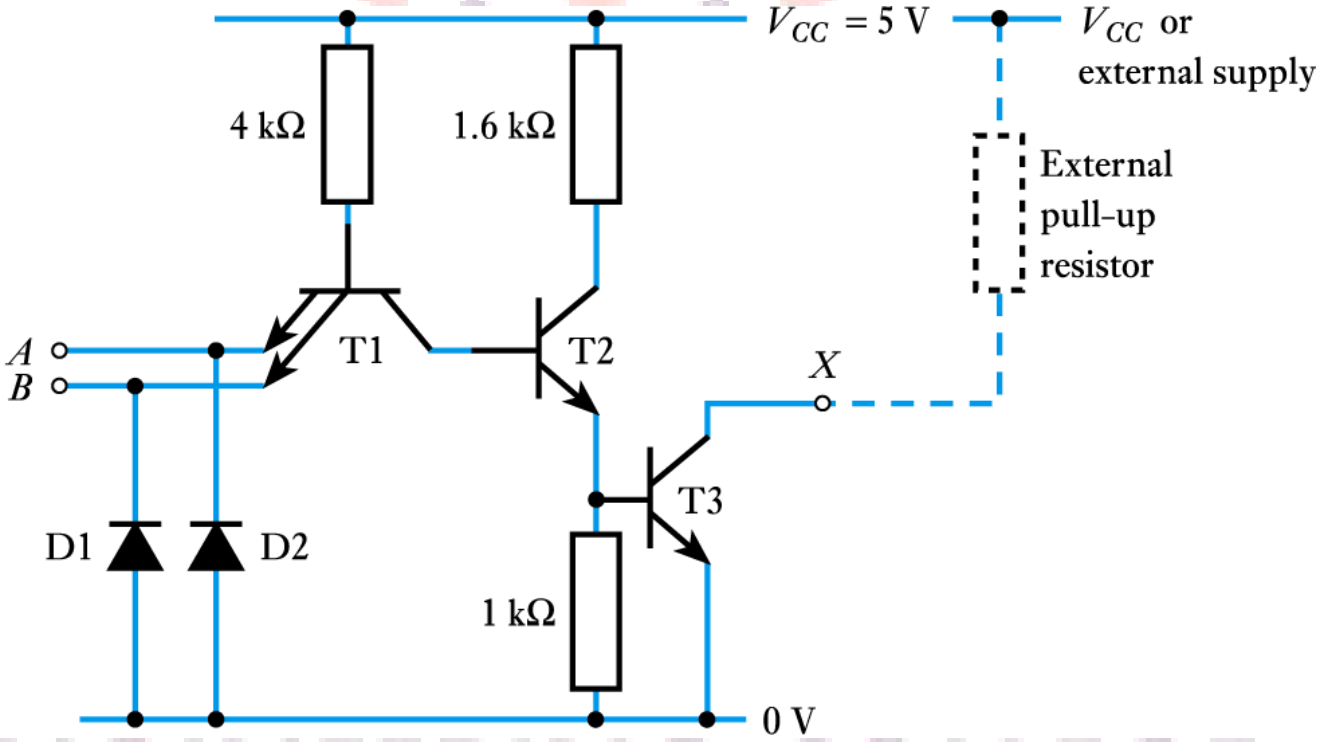


(b) Multi-emitter transistor

- **A standard TTL NAND gate**



- **A TTL NAND gate with open collector output**



Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of input variables, logic gates, and output variables.

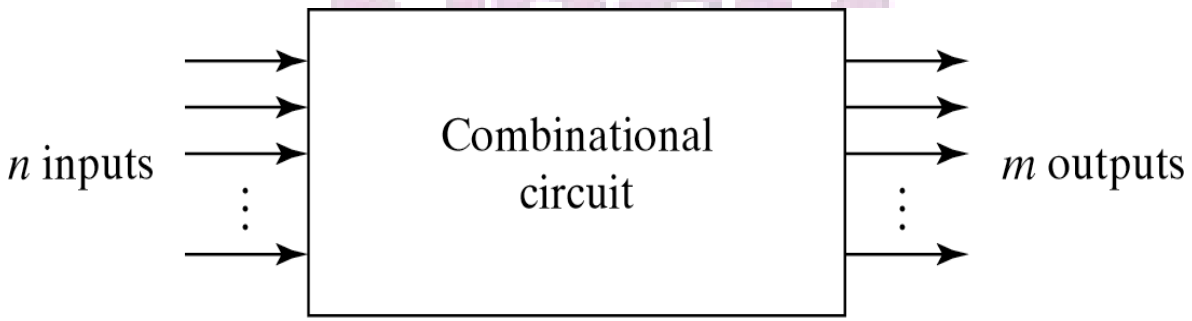


Fig. 4-1 Block Diagram of Combinational Circuit

Analysis procedure

To obtain the output Boolean functions from a logic diagram, proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

$F_2 = AB + AC + BC$; $T_1 = A + B + C$; $T_2 = ABC$; $T_3 = F_2'T_1$;

$F_1 = T_3 + T_2$

$F_1 = T_3 + T_2 = F_2'T_1 + ABC = A'BC' + A'B'C + AB'C' + ABC$

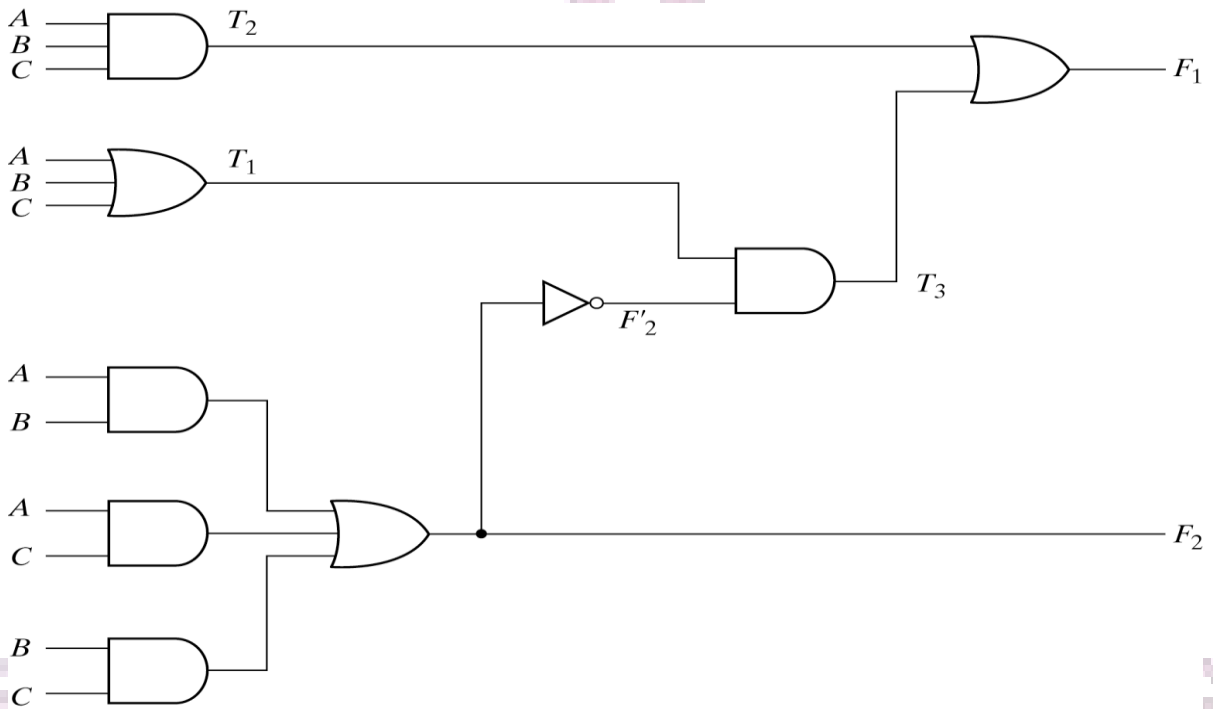


Fig. 4-2 Logic Diagram for Analysis Example

Derive truth table from logic diagram

- We can derive the truth table in Table 4-1 by using the circuit of Fig.4-2.

Table 4-1
Truth Table for the Logic Diagram of Fig. 4-2

A	B	C	F_2	F_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design procedure

1. Table4-2 is a Code-Conversion example, first, we can list the relation of the BCD and Excess-3 codes in the truth table.

Table 4-2
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Karnaugh map

2. For each symbol of the Excess-3 code, we use 1's to draw the map for simplifying Boolean function.

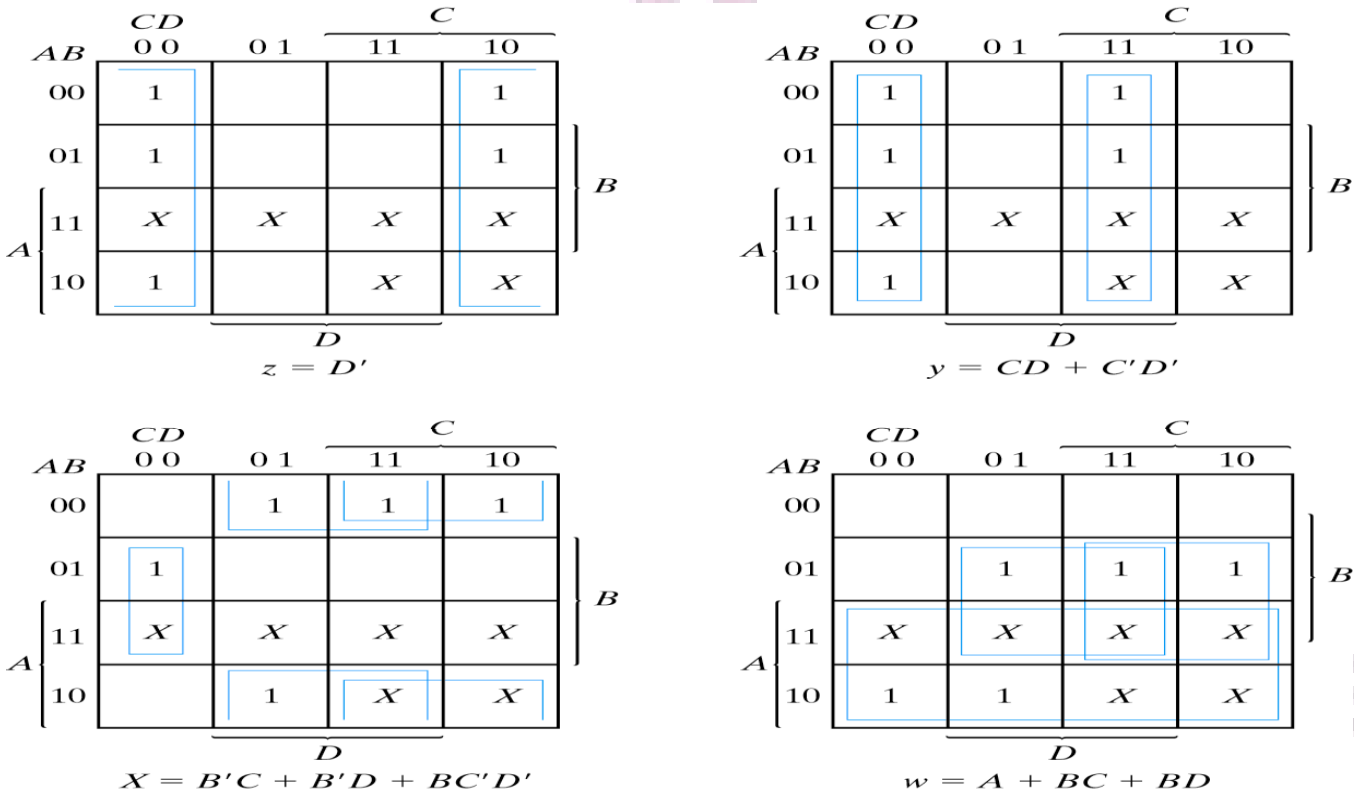


Fig. 4-3 Maps for BCD to Excess-3 Code Converter

z = D’; y = CD + C’D’ = CD + (C + D)’

x = B’C + B’D + BC’D’ = B’(C + D) + B(C + D)’

w = A + BC + BD = A + B(C + D)

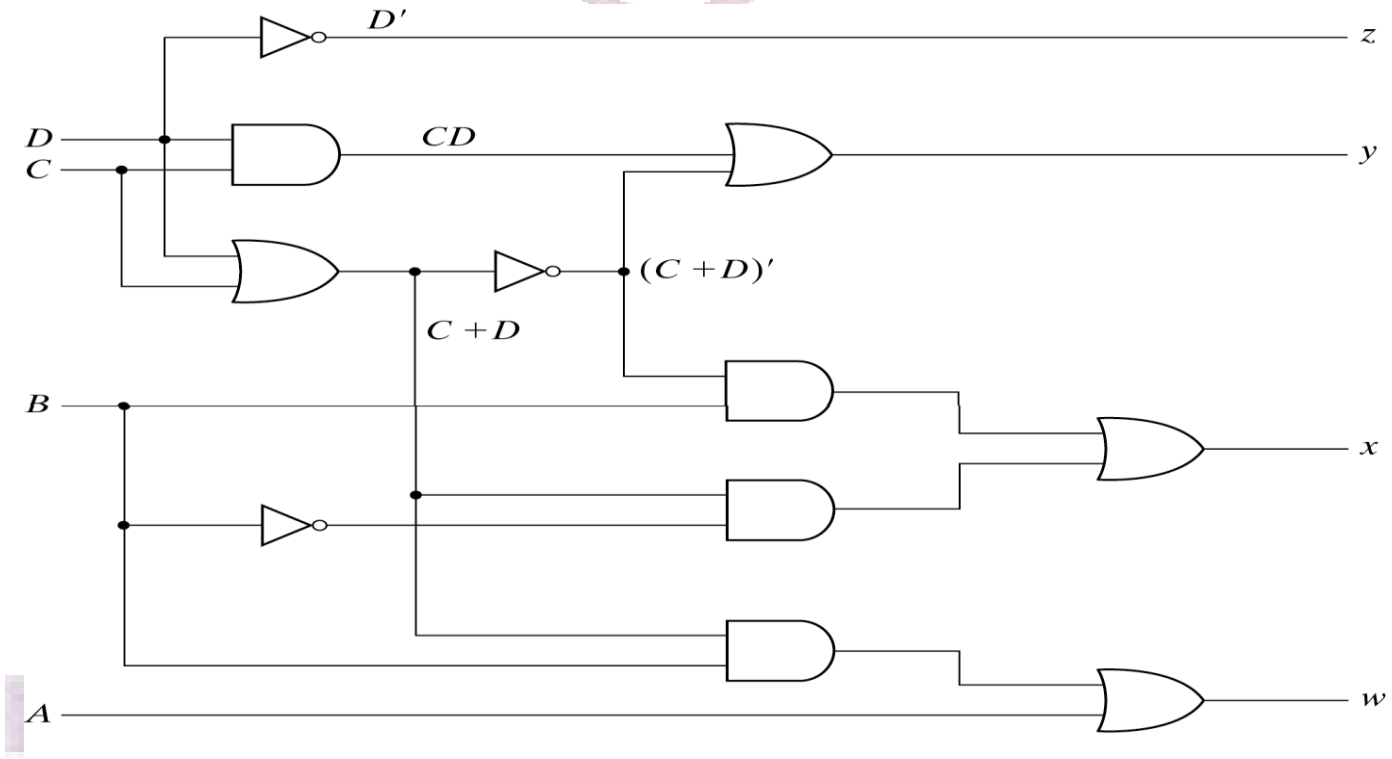


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter

- A combinational circuit that performs the addition of two bits is called a half adder.
- The truth table for the half adder is listed below:

Table 4-3
Half Adder

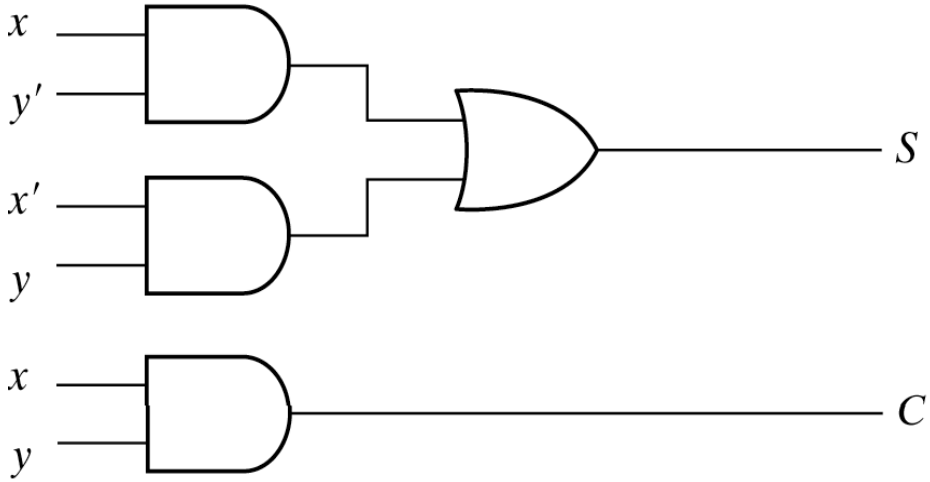
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S: Sum
C: Carry

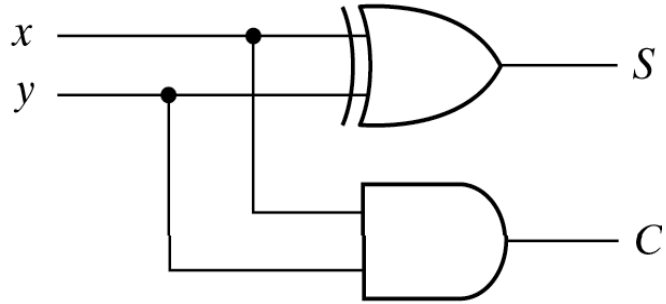
$S = x'y + xy'$

$C = xy$

Implementation of Half-Adder



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

Fig. 4-5 Implementation of Half-Adder

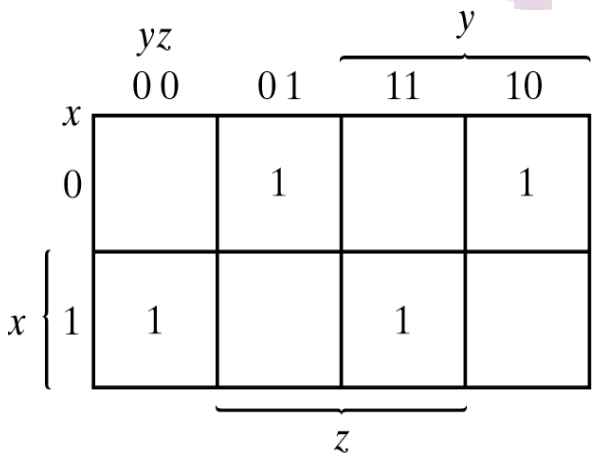
Full-Adder

- One that performs the addition of three bits(two significant bits and a previous carry) is a full adder.

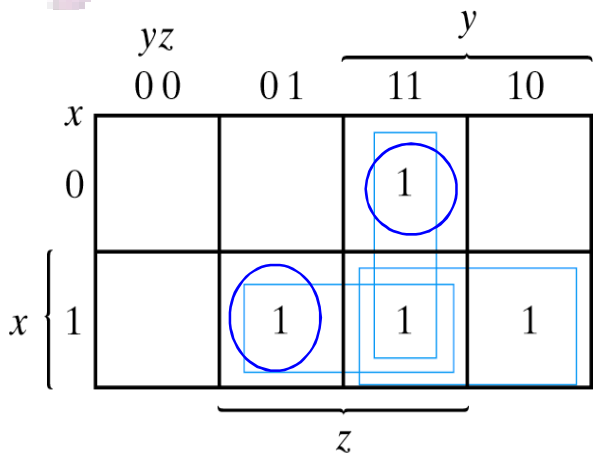
Table 4-4
Full Adder

x	y	z	C	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

Simplified Expressions



$S = x'y'z + x'yz' + xy'z' + xyz$



$C = xy + xz + yz$
$= xy + xy'z + x'yz$

Fig. 4-6 Maps for Full Adder

$S = x'y'z + x'yz' + xy'z' + xyz$

$C = xy + xz + yz$

Full adder implemented in SOP

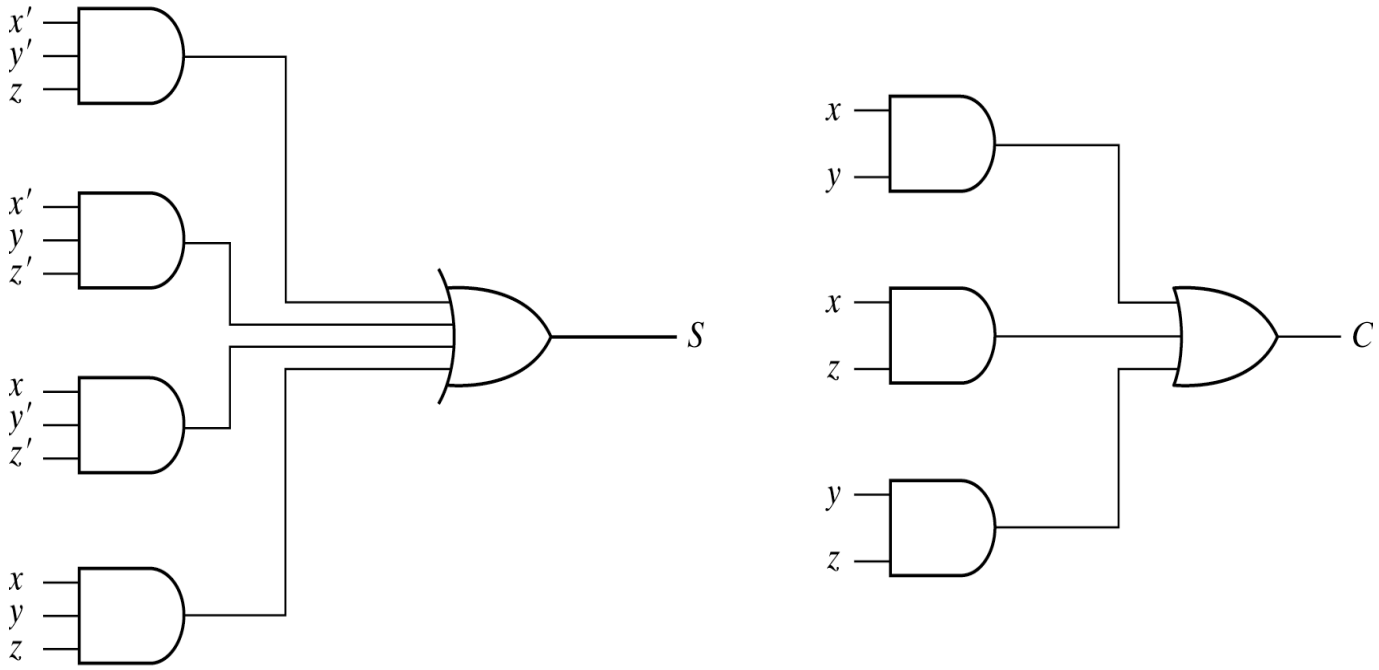


Fig. 4-7 Implementation of Full Adder in Sum of Products

Another implementation

- Full-adder can also implemented with two half adders and one OR gate (Carry Look-Ahead adder).

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= xy'z' + x'yz' + xyz + x'y'z$$
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

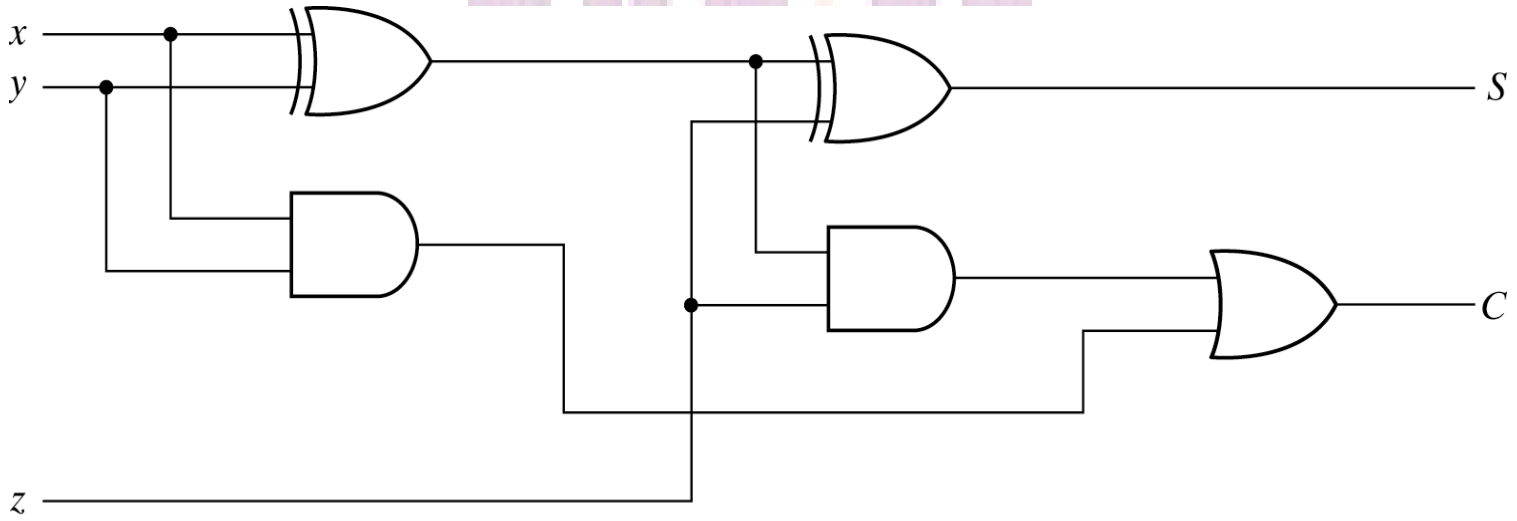
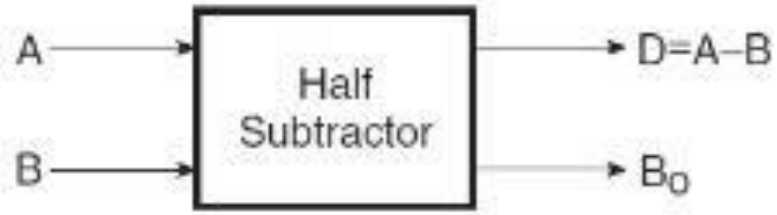


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

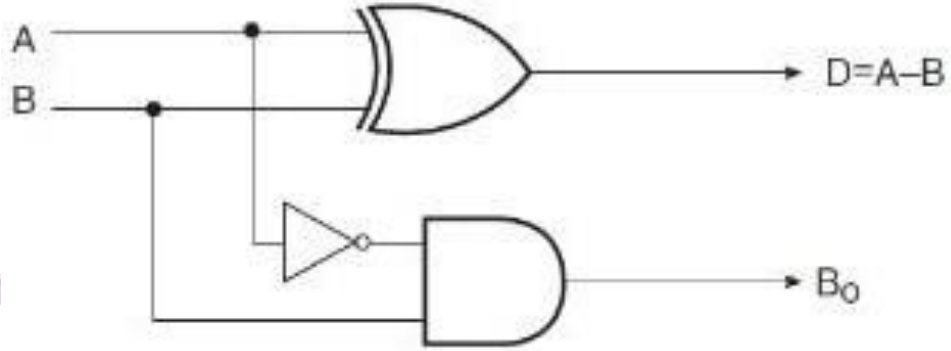
Half Subtractor

Truth table

A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

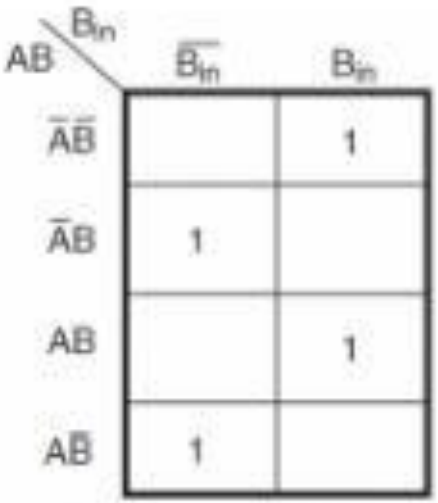


Logic Circuit

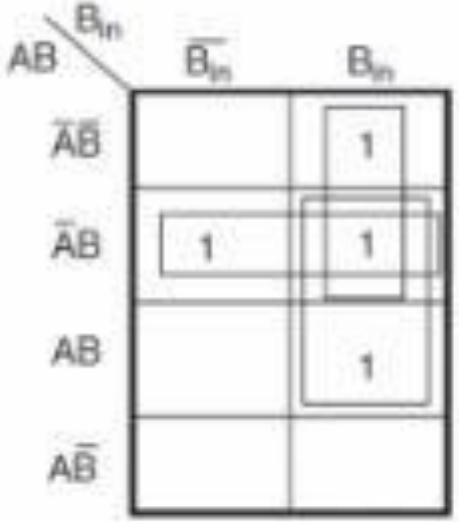


Full Subtractor

Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_0)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

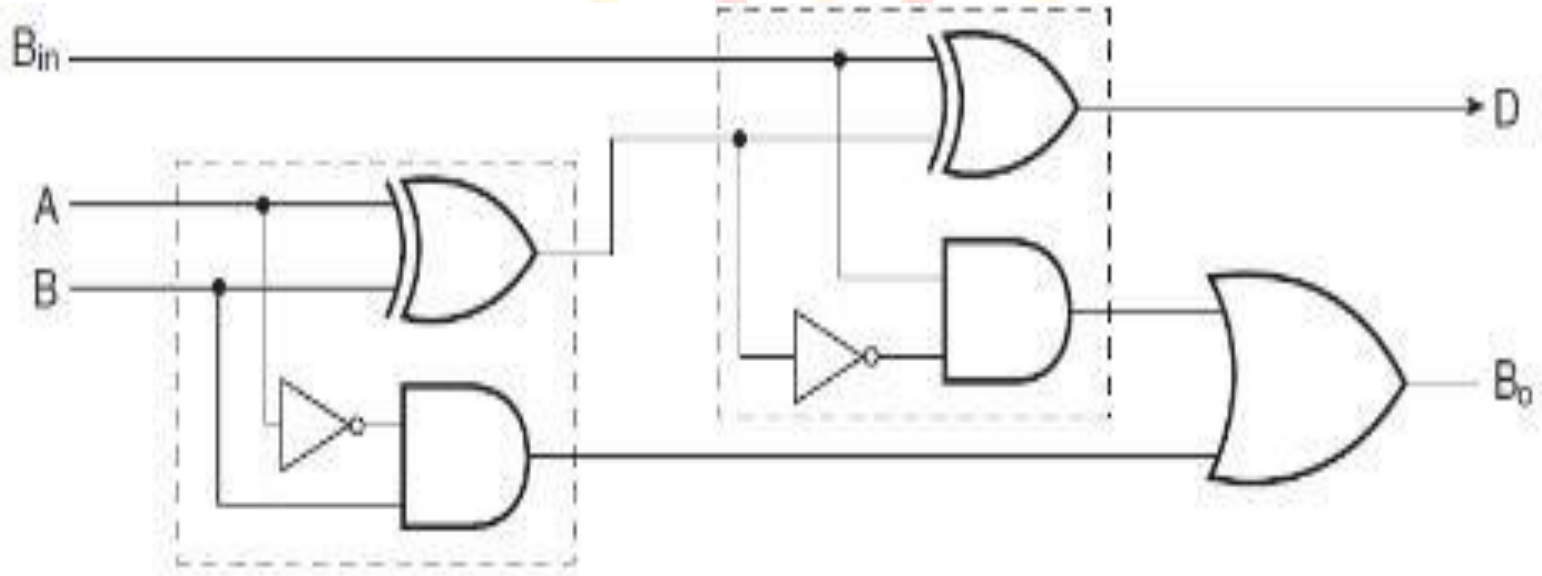


$$D = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + A.\overline{B}.\overline{B}_{in} + A.B.B_{in}$$



$$B_o = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + \overline{A}.B.B_{in} + A.B.B_{in}$$

Full Subtractor



Binary adder

- This is also called Ripple Carry Adder ,because of the construction with full adders are connected in cascade.

Subscript i:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

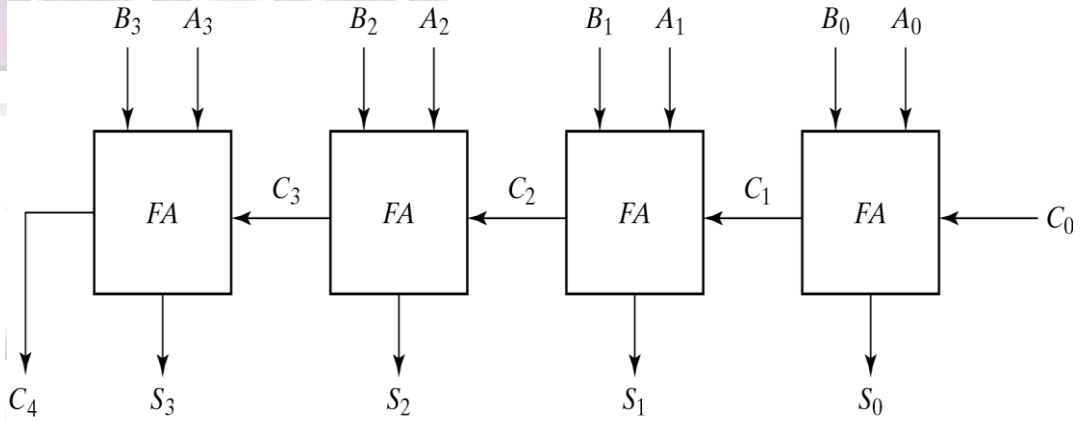


Fig. 4-9 4-Bit Adder

- Fig.4-9 causes a unstable factor on carry bit, and produces a longest propagation delay.
- The signal from C_i to the output carry C_{i+1}, propagates through an AND and OR gates, so, for an n-bit RCA, there are 2n gate levels for the carry to propagate from input to output.

- Because the propagation delay will affect the output signals on different time, so the signals are given enough time to get the precise and stable outputs.
- The most widely used technique employs the principle of carry look-ahead to improve the speed of the algorithm.

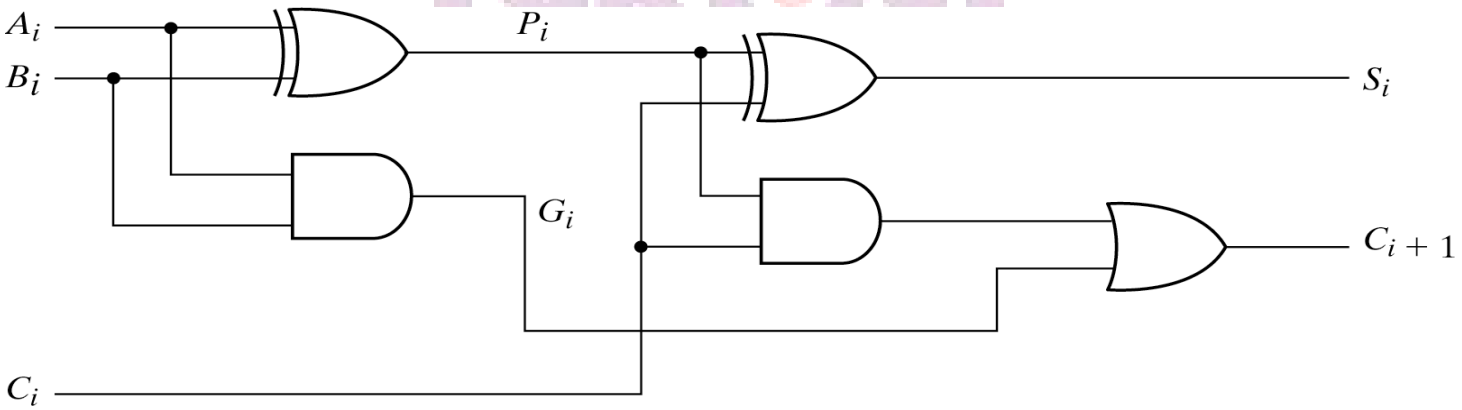


Fig. 4-10 Full Adder with P and G Shown

Boolean functions

$P_i = A_i \oplus B_i$ steady state value

$G_i = A_iB_i$ steady state value

Output sum and carry

$S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_iC_i$

G_i : carry generate P_i : carry propagate

C_0 = input carry

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

- C_3 does not have to wait for C_2 and C_1 to propagate.

Logic diagram of carry look-ahead generator

- C_3 is propagated at the same time as C_2 and C_1.

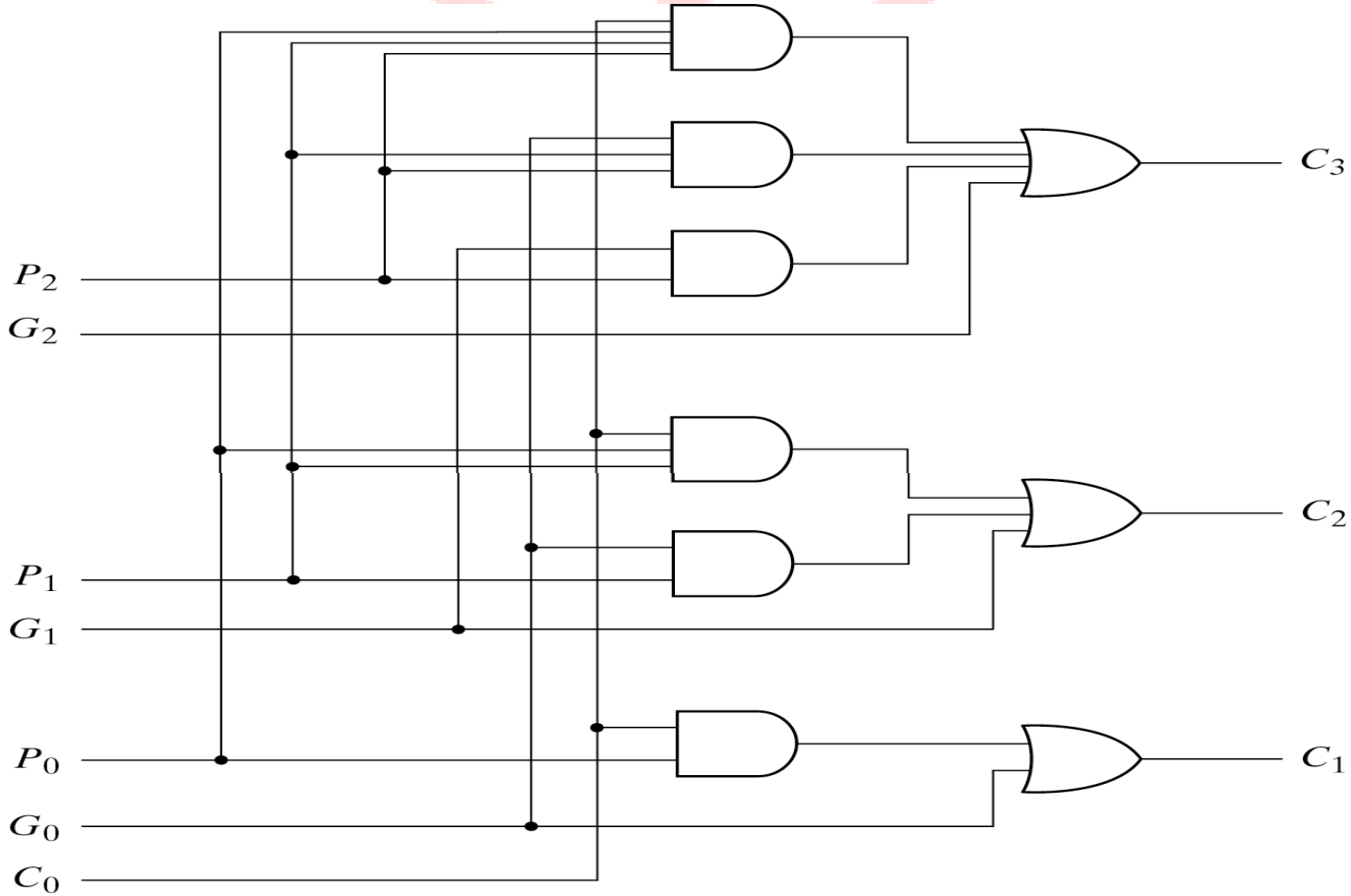


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

4-bit adder with carry lookahead

- Delay time of n-bit CLAA = XOR + (AND + OR) + XOR

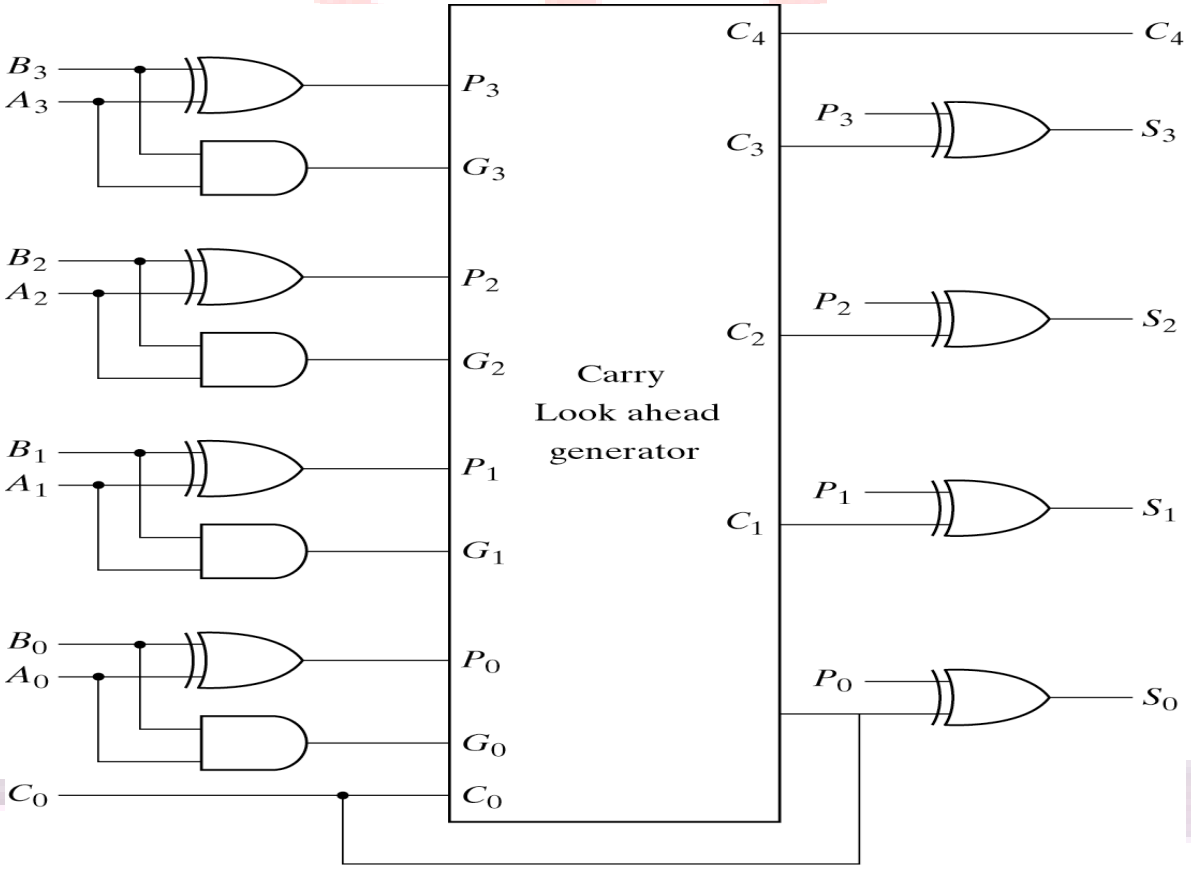


Fig. 4-12 4-Bit Adder with Carry Lookahead

M = 1→subtractor ; M = 0→adder

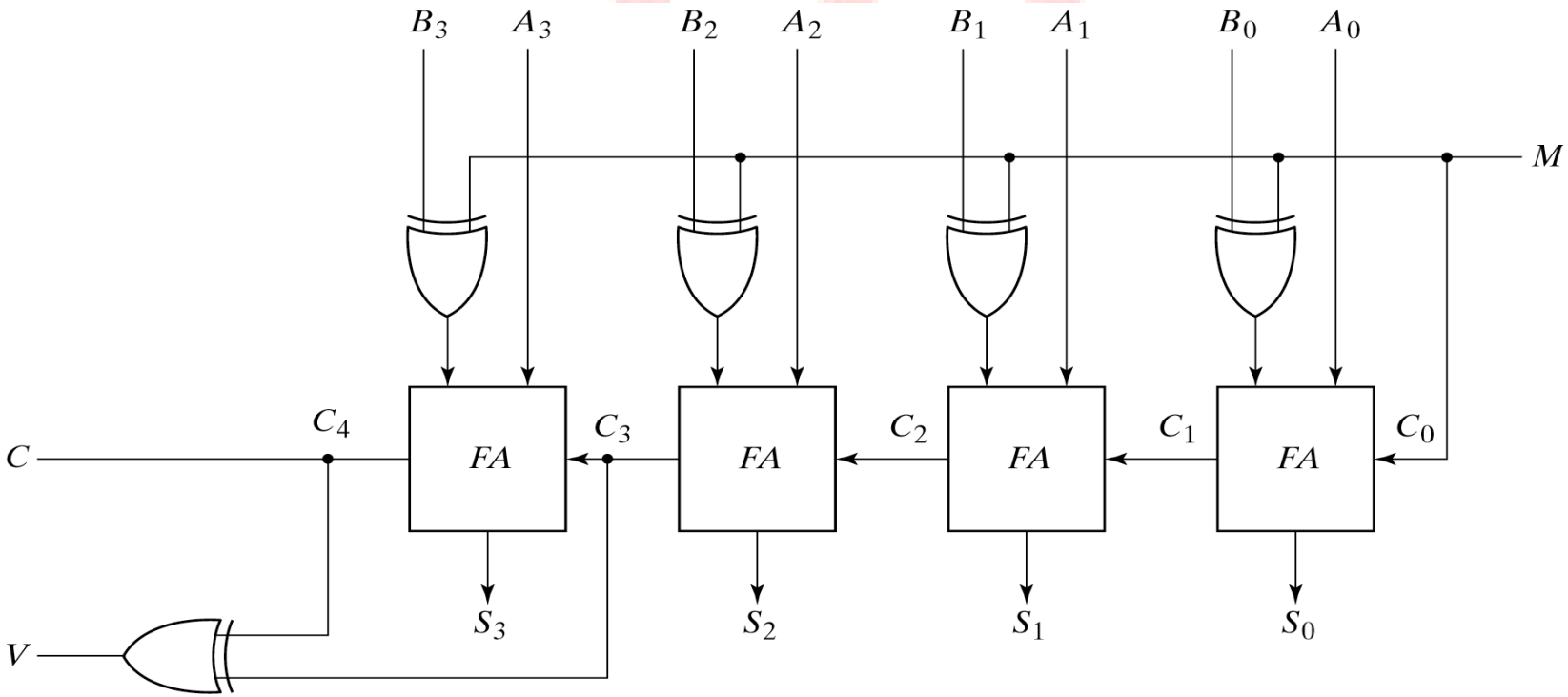


Fig. 4-13 4-Bit Adder Subtractor

4.5 Decimal adder

BCD adder can't exceed 9 on each input digit. K is the carry.

Table 4-5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Rules of BCD adder

- When the binary sum is greater than 1001, we obtain a non-valid BCD representation.
- The addition of binary 6(0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1.

$$C = K + Z_8Z_4 + Z_8Z_2$$

Implementation of BCD adder

- A decimal parallel adder that adds n decimal digits needs n BCD adder stages.
- The output carry from one stage must be connected to the input carry of the next higher-order stage.

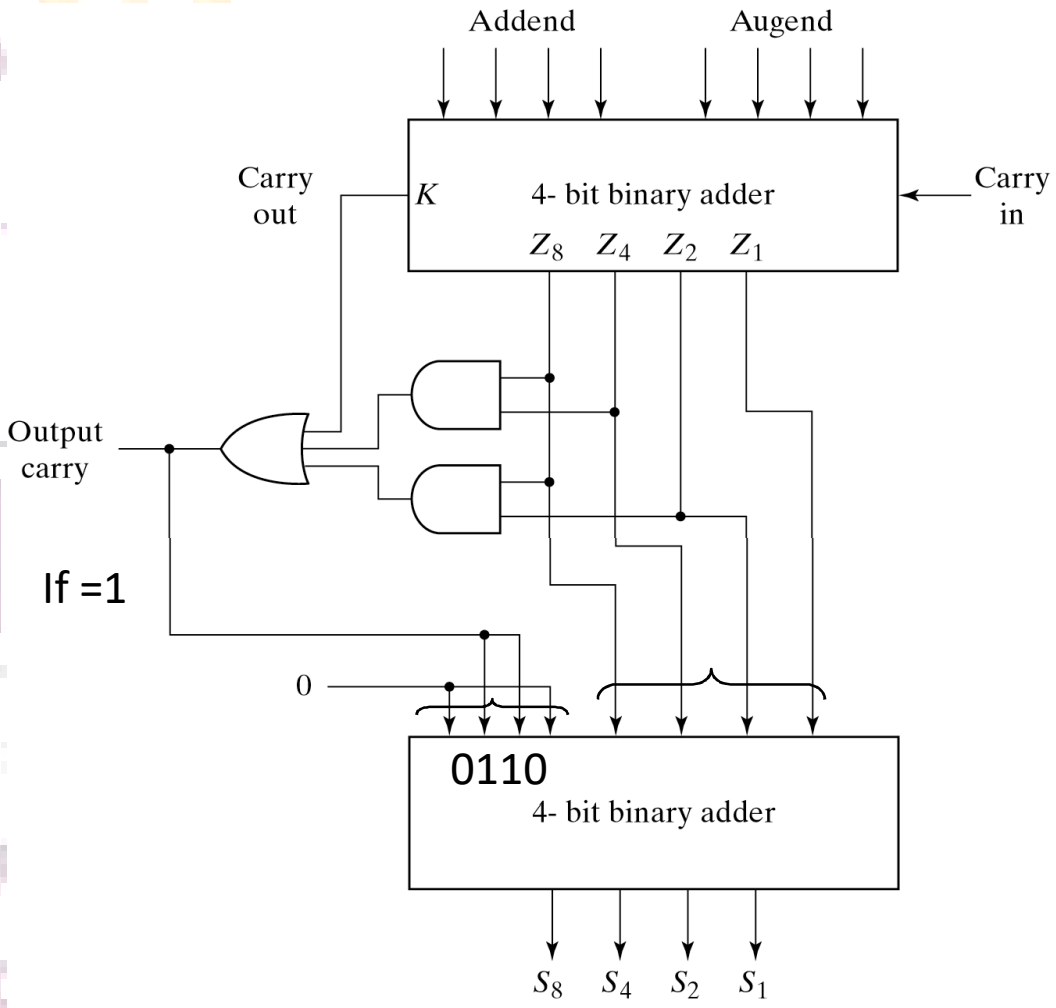


Fig. 4-14 Block Diagram of a BCD Adder

4-6. Binary multiplier

- Usually there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products.

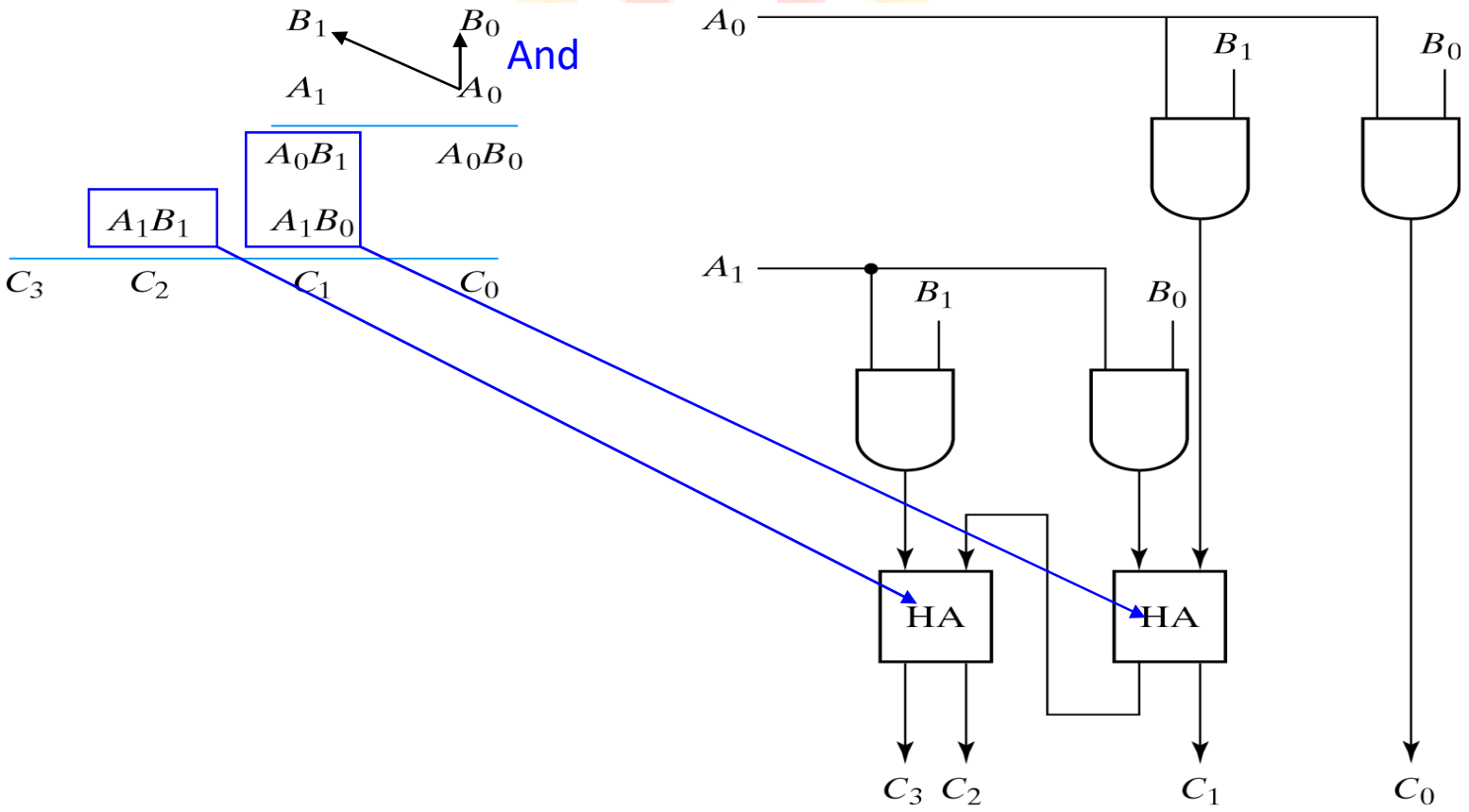


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

4-bit by 3-bit binary multiplier

- For J multiplier bits and K multiplicand bits we need (J X K) AND gates and (J – 1) K-bit adders to produce a product of J+K bits.
- K=4 and J=3, we need 12 AND gates and two 4-bit adders.

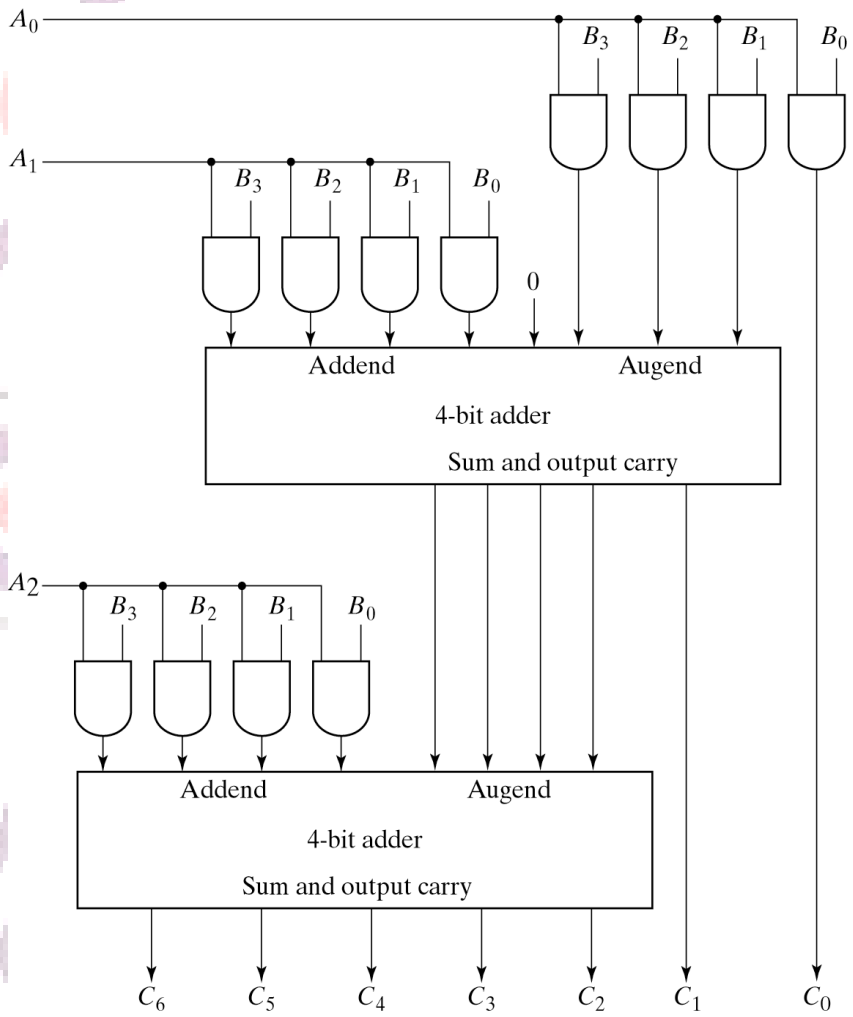


Fig. 4-16 4-Bit by 3-Bit Binary Multiplier

- The equality relation of each pair of bits can be expressed logically with an exclusive-NOR function as:

$A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$

$x_i = A_iB_i + A_i'B_i'$ for i = 0, 1, 2, 3

$(A = B) = x_3x_2x_1x_0$

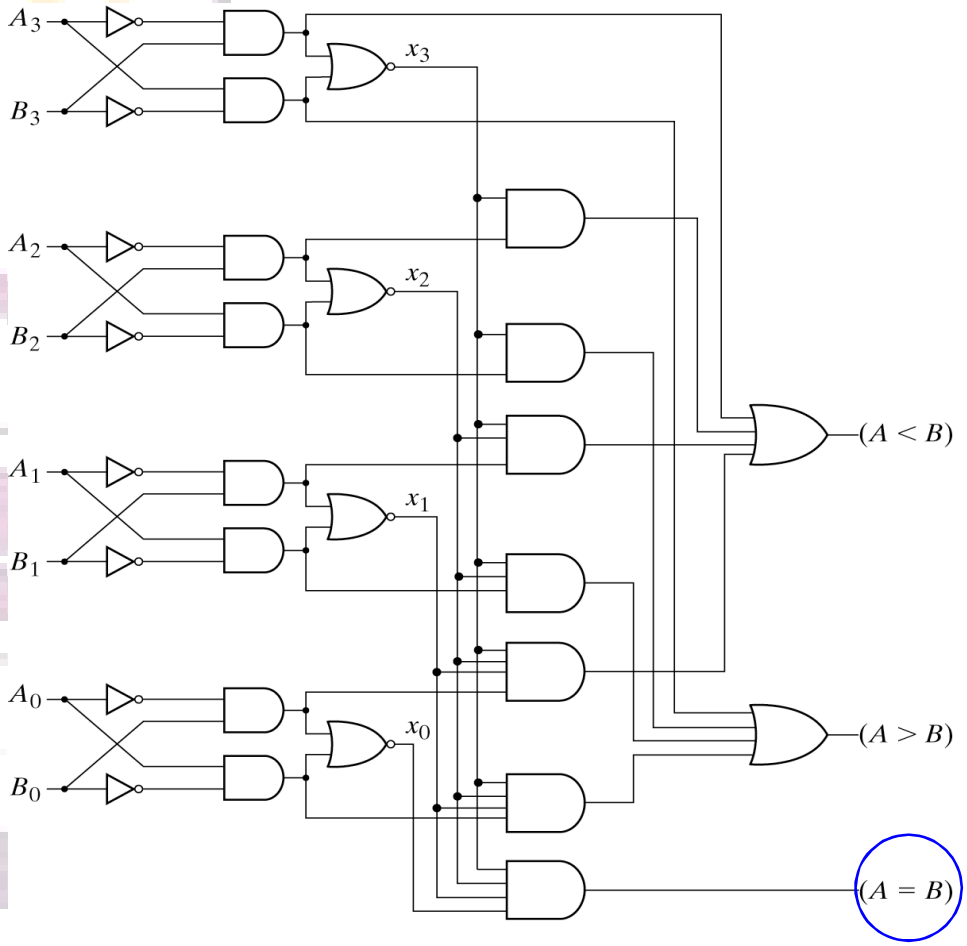


Fig. 4-17 4-Bit Magnitude Comparator

- We inspect the relative magnitudes of pairs of MSB. If equal, we compare the next lower significant pair of digits until a pair of unequal digits is reached.
- If the corresponding digit of A is 1 and that of B is 0, we conclude that A>B.

(A>B)=

$A_3B'_3+x_3A_2B'_2+x_3x_2A_1B'_1+x_3x_2x_1A_0B'_0$

(A<B)=

$A'_3B_3+x_3A'_2B_2+x_3x_2A'_1B_1+x_3x_2x_1A'_0B_0$

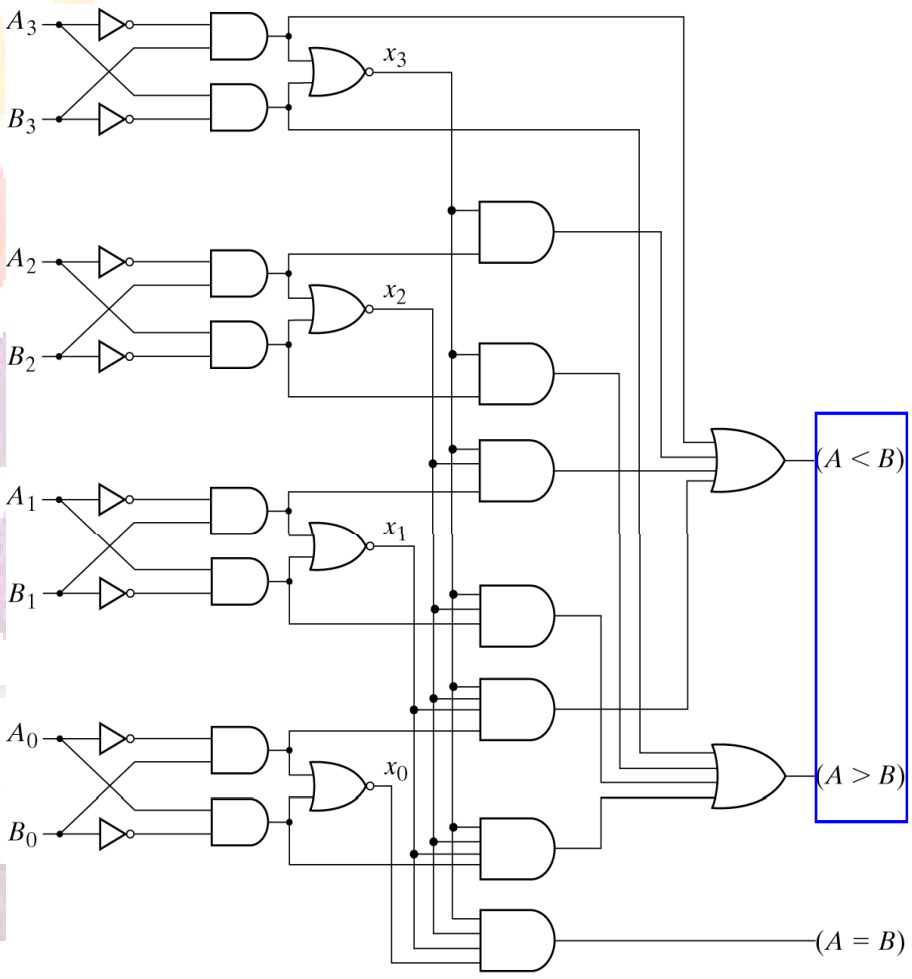


Fig. 4-17 4-Bit Magnitude Comparator

- The decoder is called n-to-m-line decoder, where $m \leq 2^n$.
- the decoder is also used in conjunction with other code converters such as a BCD-to-seven_segment decoder.
- 3-to-8 line decoder: For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

Implementation and truth table

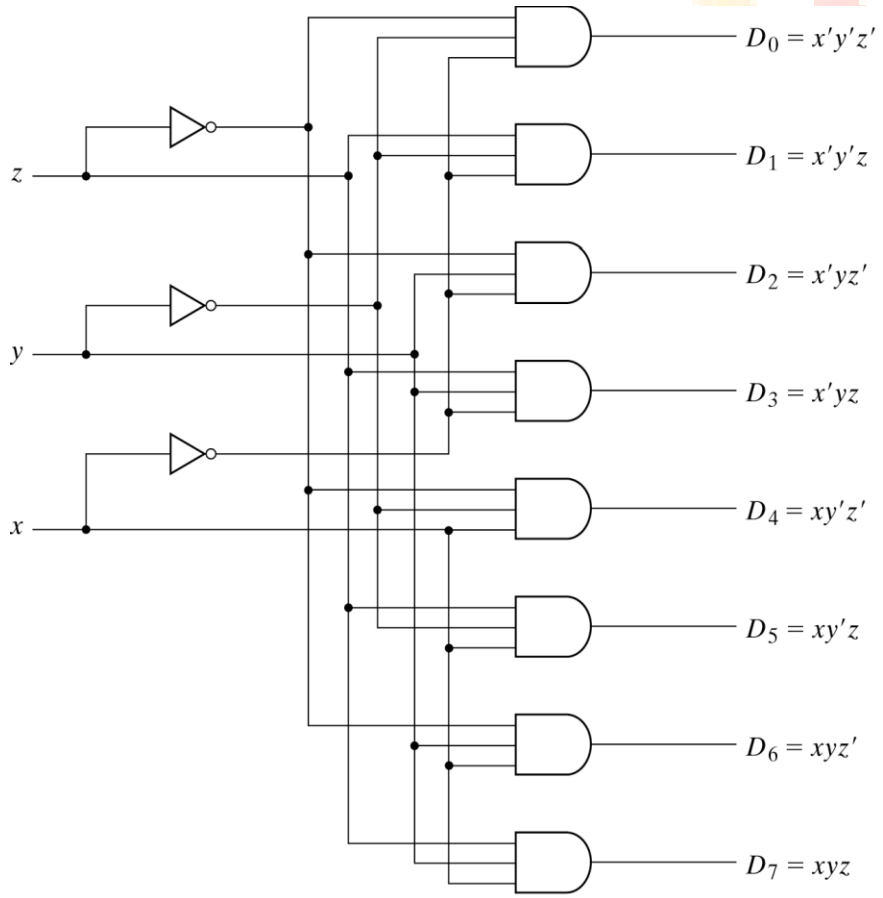


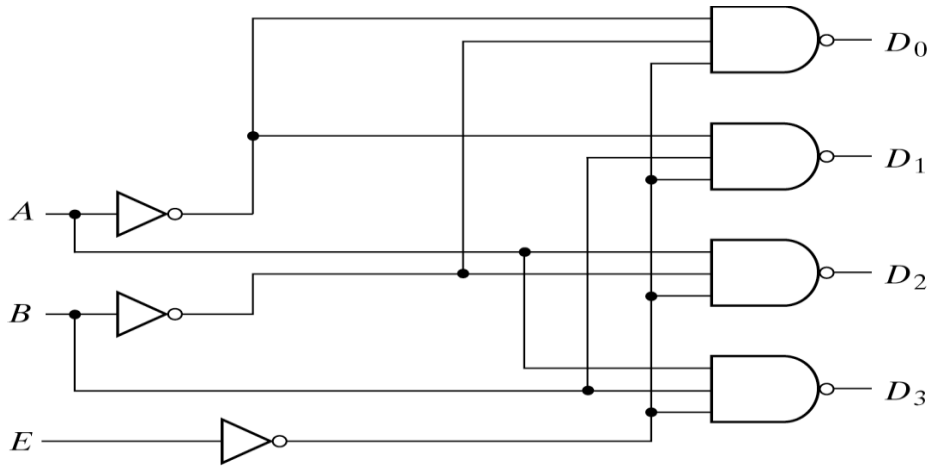
Fig. 4-18 3-to-8-Line Decoder

Table 4-6
Truth Table of a 3-to-8-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

Decoder with enable input

- Some decoders are constructed with NAND gates, it becomes more economical to generate the decoder minterms in their complemented form.
- As indicated by the truth table , only one output can be equal to 0 at any given time, all other outputs are equal to 1.



(a) Logic diagram

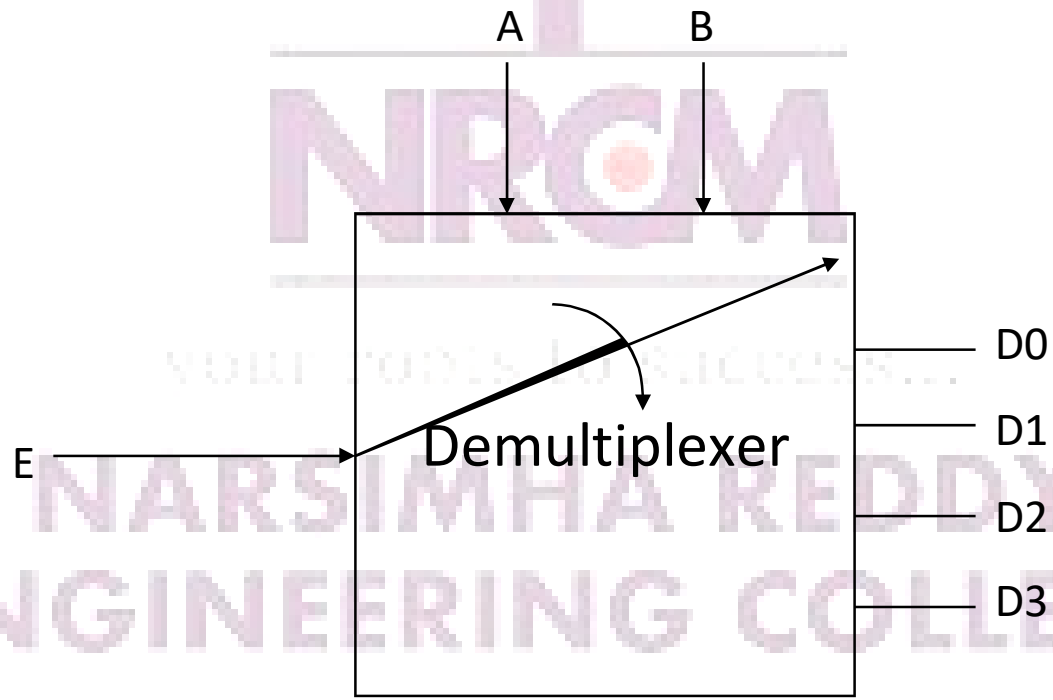
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

Fig. 4-19 2-to-4-Line Decoder with Enable Input

Demultiplexer

- A decoder with an enable input is referred to as a decoder/demultiplexer.
- The truth table of demultiplexer is the same with decoder.



3-to-8 decoder with enable implement the 4-to-16 decoder

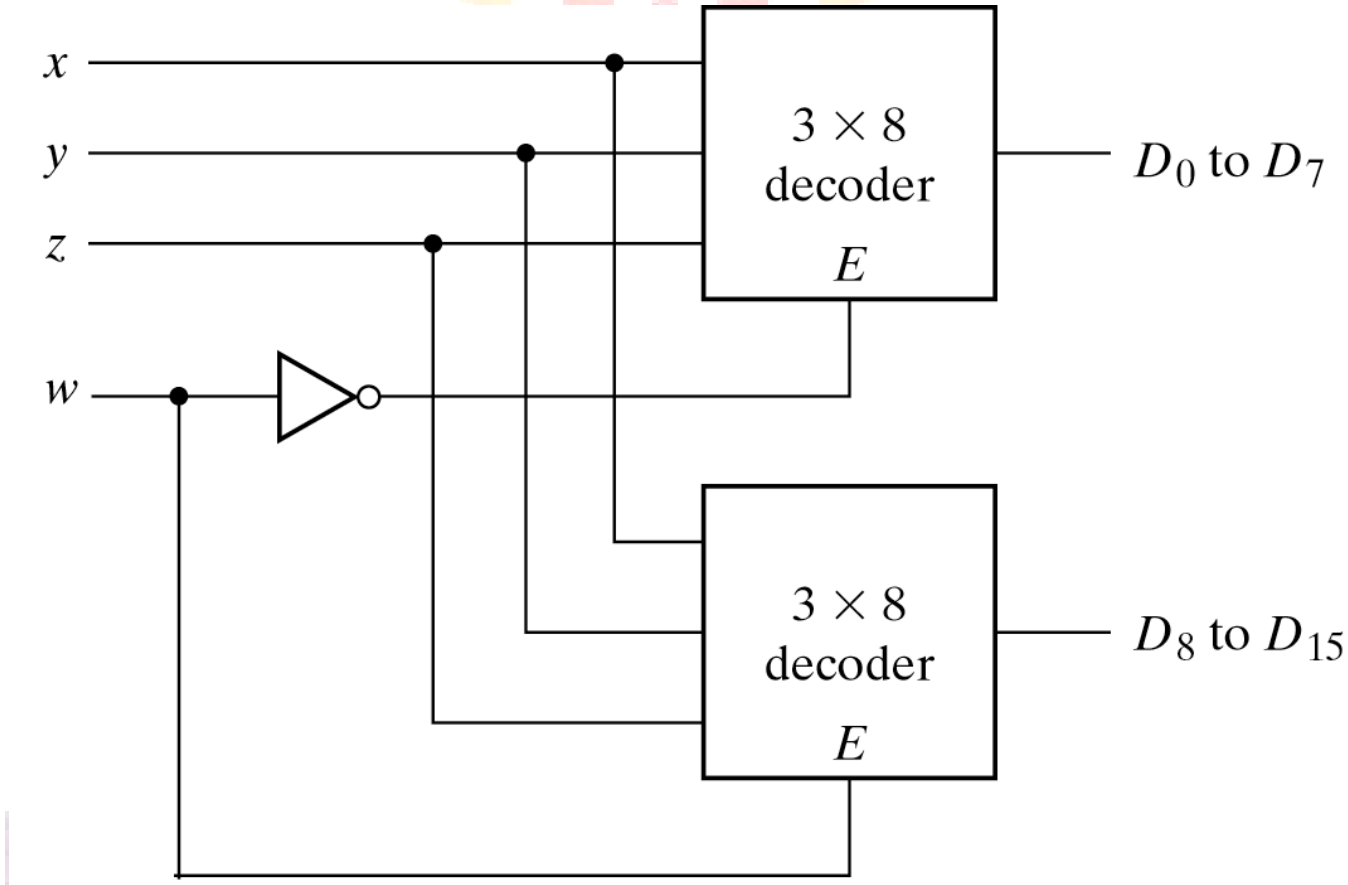


Fig. 4-20 4 × 16 Decoder Constructed with Two 3 × 8 Decoders

Implementation of a Full Adder with a Decoder

- From table 4-4, we obtain the functions for the combinational circuit in sum of minterms:

  $S(x, y, z) = \sum(1, 2, 4, 7)$

  $C(x, y, z) = \sum(3, 5, 6, 7)$

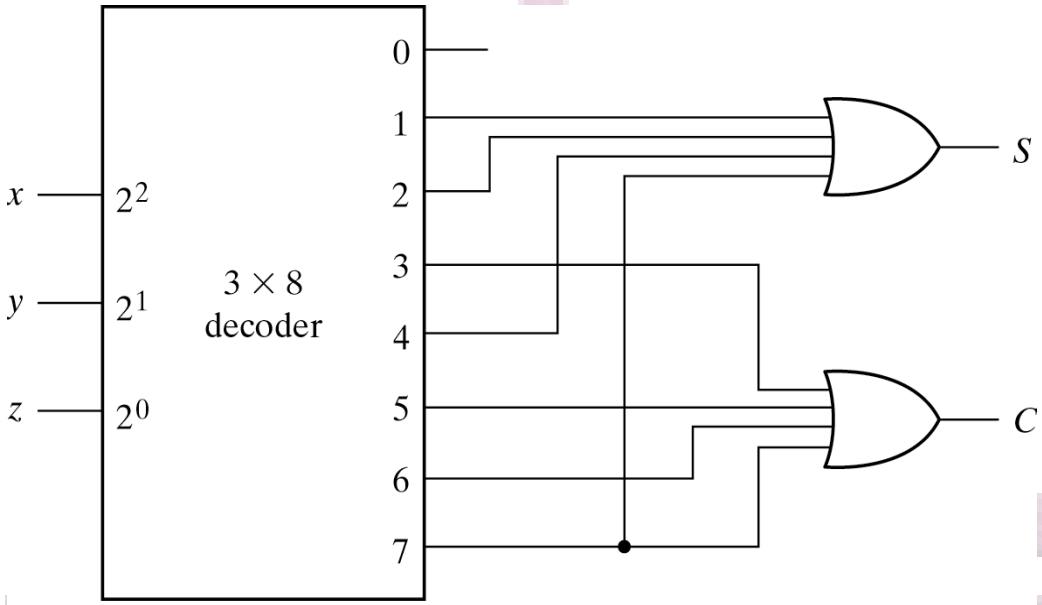


Fig. 4-21 Implementation of a Full Adder with a Decoder

4.9 Encoders

- An encoder is the inverse operation of a decoder.
- We can derive the Boolean functions by table 4-7

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

Table 4-7
Truth Table of 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

- If two inputs are active simultaneously, the output produces an undefined combination. We can establish an input priority to ensure that only one input is encoded.
- Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; the output is the same as when D_0 is equal to 1.
- The discrepancy tables on Table 4-7 and Table 4-8 can resolve aforesaid condition by providing one more output to indicate that at least one input is equal to 1.

V=0→no valid inputs

V=1→valid inputs

X's in output columns represent don't-care conditions

X's in the input columns are useful for representing a truth table in condensed form.

Instead of listing all 16 minterms of four variables.

Table 4-8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

- Implementation of table 4-8

$x = D_2 + D_3$

$y = D_3 + D_1D'_2$

$V = D_0 + D_1 + D_2 + D_3$

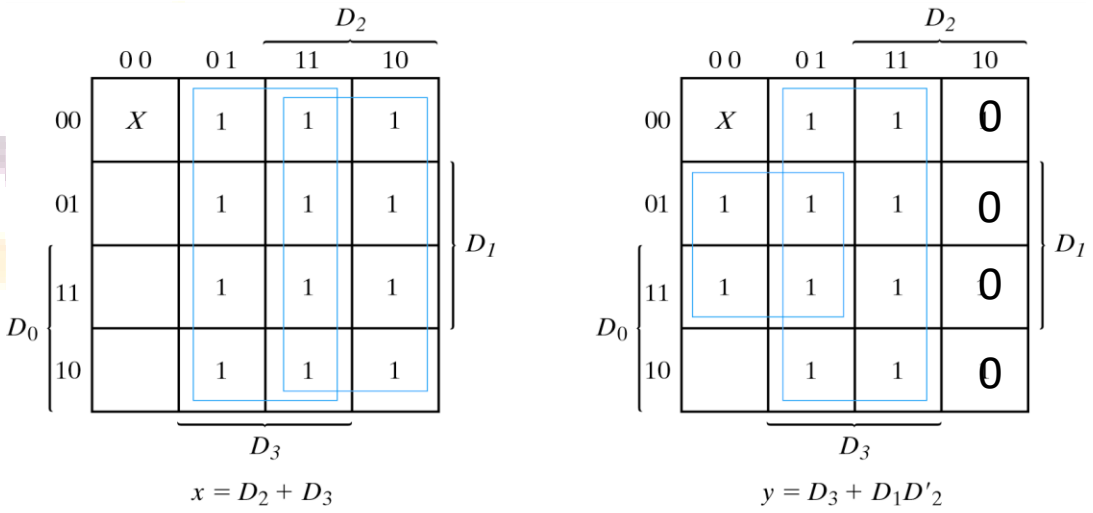


Fig. 4-22 Maps for a Priority Encoder

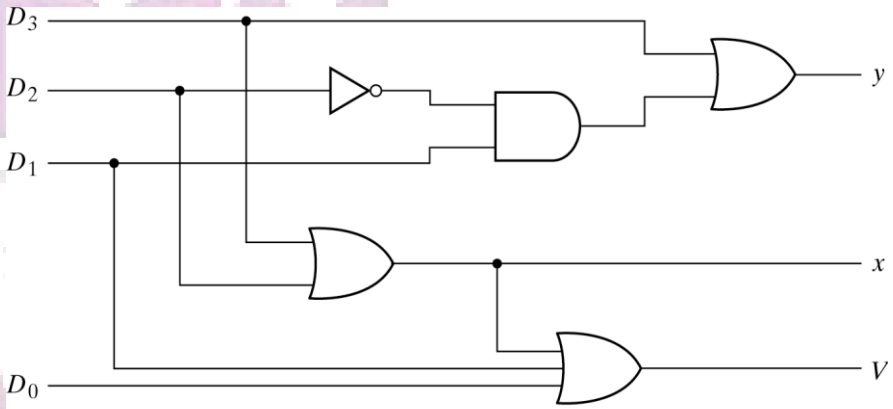


Fig. 4-23 4-Input Priority Encoder

4.10 Multiplexers

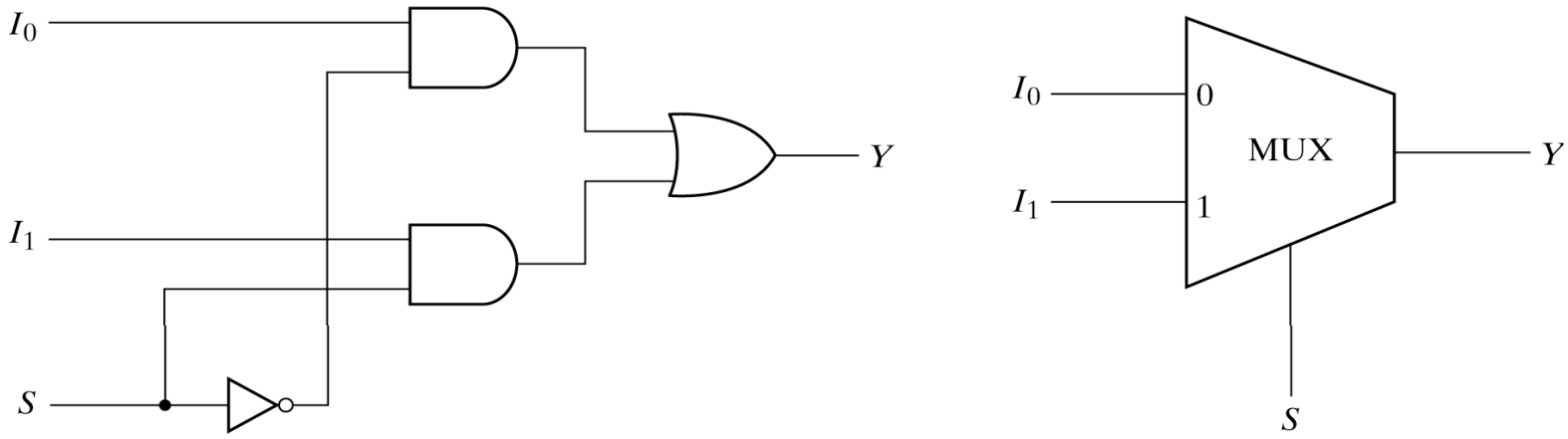
S = 0, Y = I_0

S = 1, Y = I_1

Truth Table→

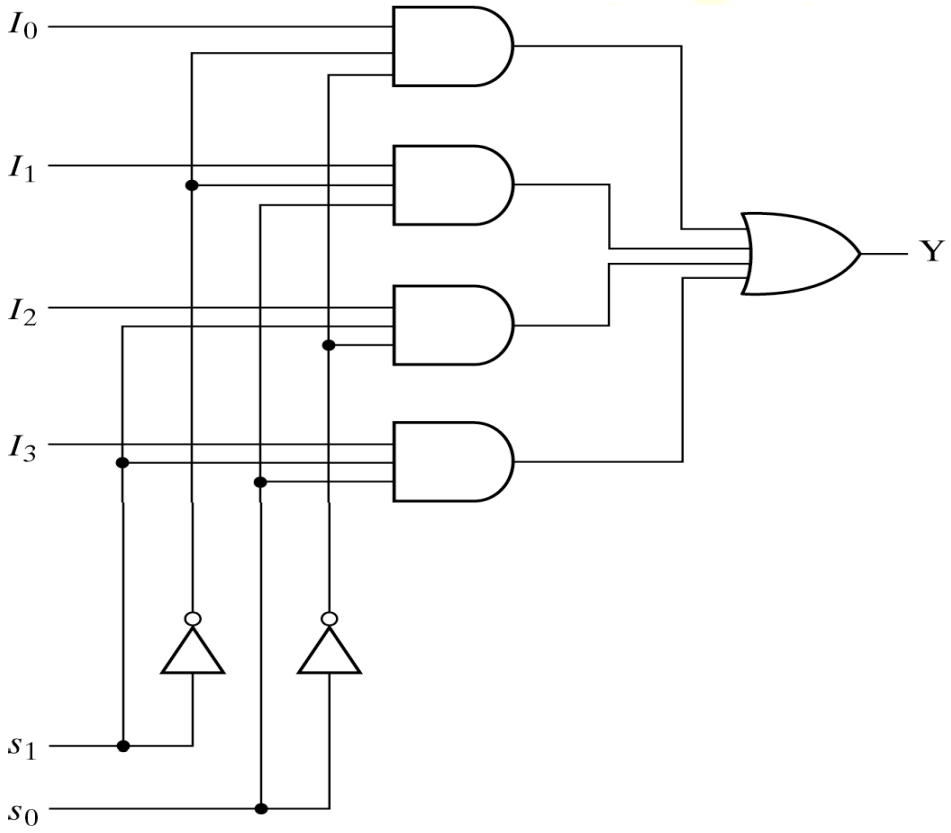
S	Y
0	I_0
1	I_1

$Y = S'I_0 + SI_1$



(a) Logic diagram (b) Block diagram

Fig. 4-24 2-to-1-Line Multiplexer



(a) Logic diagram

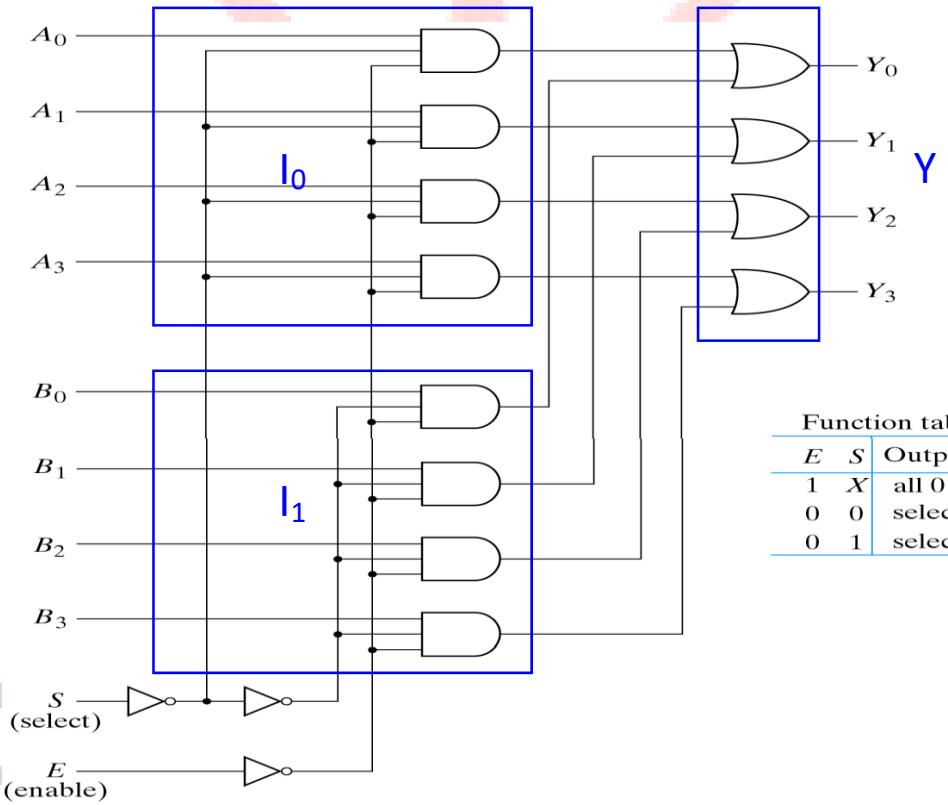
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Fig. 4-25 4-to-1-Line Multiplexer

er

- Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. Compare with Fig4-24.



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Fig. 4-26 Quadruple 2-to-1-Line Multiplexer

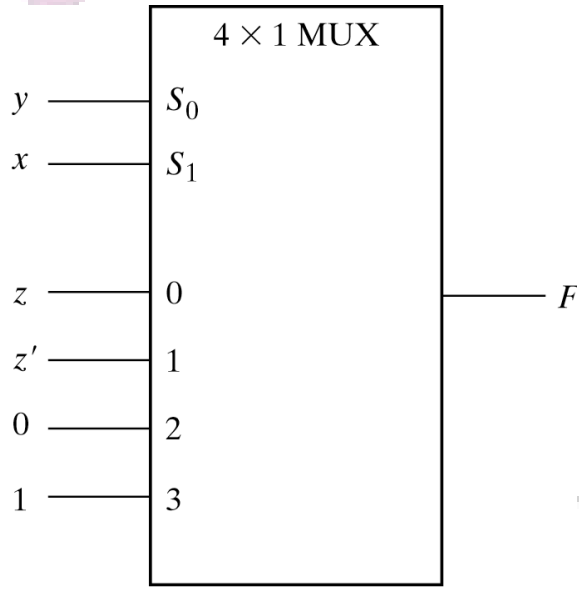
ɔn

- A more efficient method for implementing a Boolean function of n variables with a multiplexer that has n-1 selection inputs.

$$F(x, y, z) = \Sigma(1,2,6,7)$$

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Fig. 4-27 Implementing a Boolean Function with a Multiplexer

‹er

$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

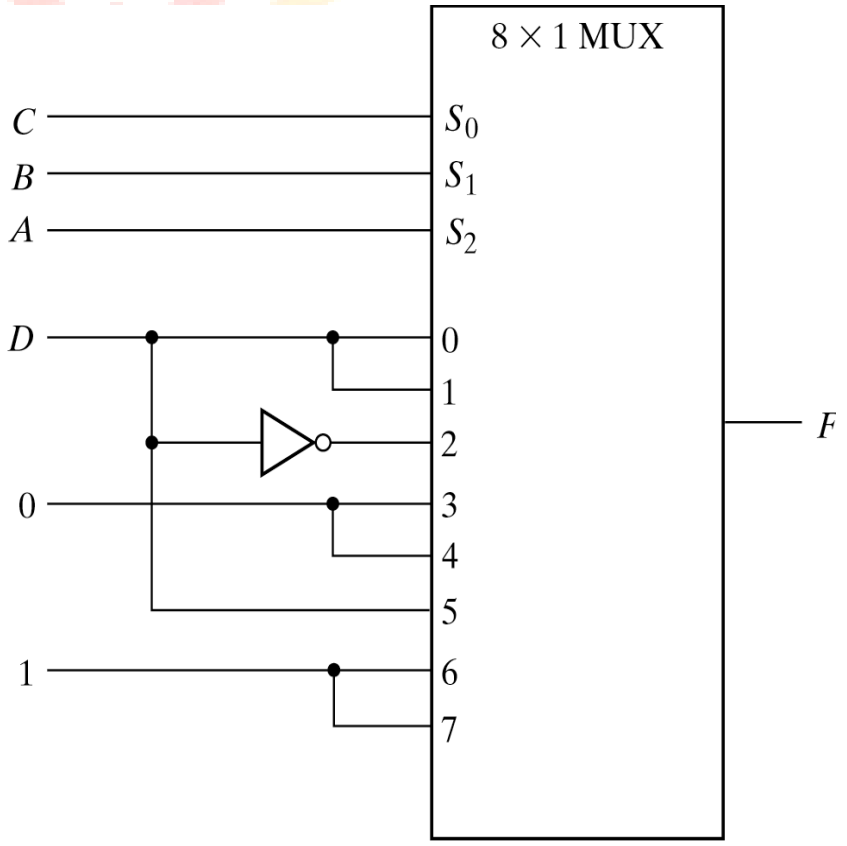


Fig. 4-28 Implementing a 4-Input Function with a Multiplexer

Three-State Gates

- A multiplexer can be constructed with three-state gates.

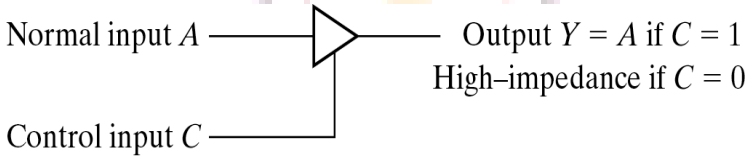
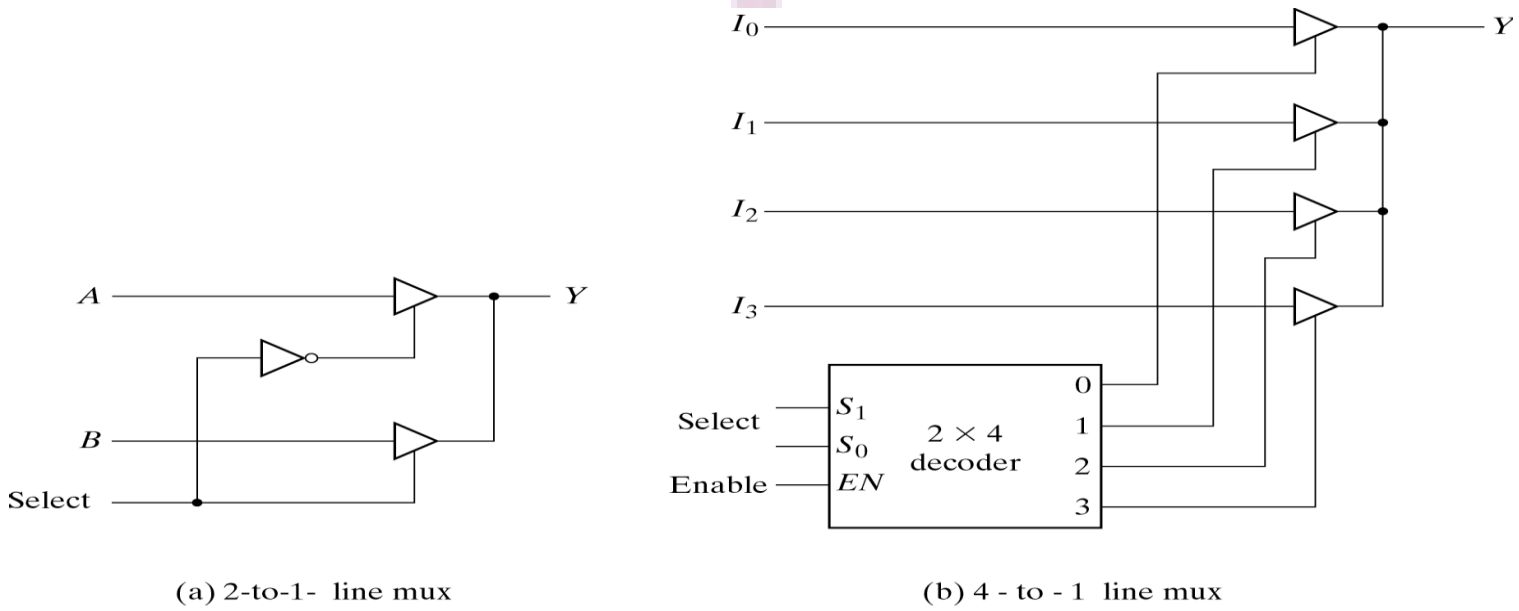


Fig. 4-29 Graphic Symbol for a Three-State Buffer



(a) 2-to-1- line mux

(b) 4 - to - 1 line mux

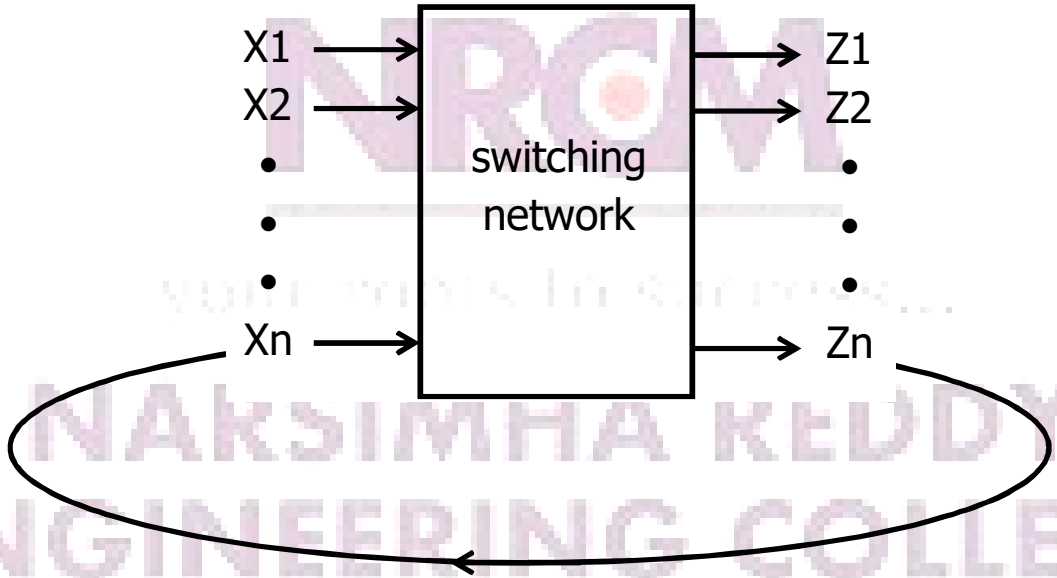
Fig. 4-30 Multiplexers with Three-State Gates

Sequential Circuits

- A **sequential circuit** is one whose outputs depend not only on its current inputs, but also on the past sequence of inputs.
- In other words, sequential circuits must be able to ”**remember**” (i.e., store) the past history of the inputs in order to produce the present output.
- The information about the previous inputs history is called the **state** of the system.
- A circuit that uses n binary state variables to **store** its past history can take up to 2^n different states.
- Since n is always finite, sequential circuits are also called **finite state machines** (FSM).

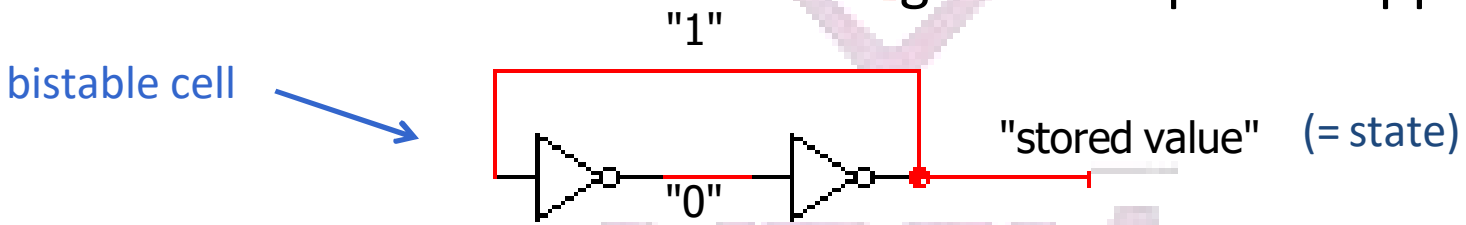
In short, sequential circuits are …

- circuits consisting of ordinary gates and feedback loops

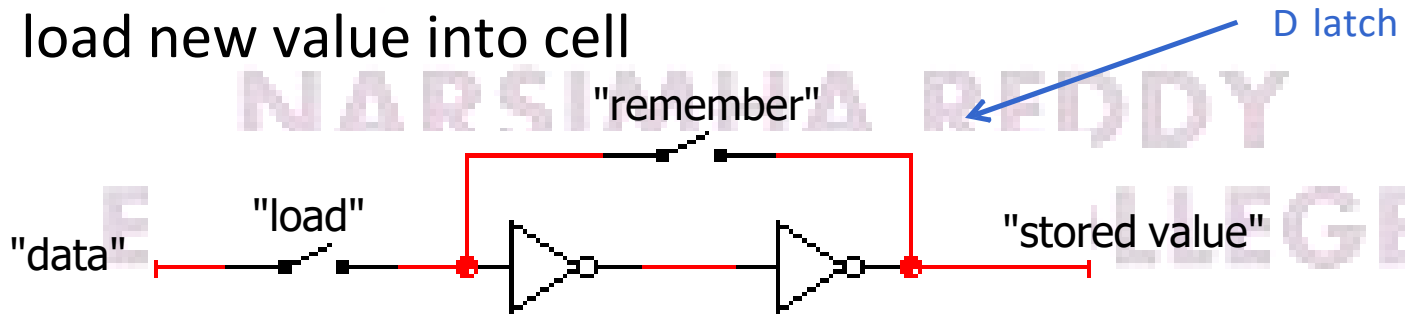


The simplest sequential circuit

- Two inverters and a feedback loop form a “static” storage cell
  - The cell will hold value as long as it has power applied



- How to get a new value into the storage cell?
  - selectively break feedback path
  - load new value into cell



Latches and Flip-Flops

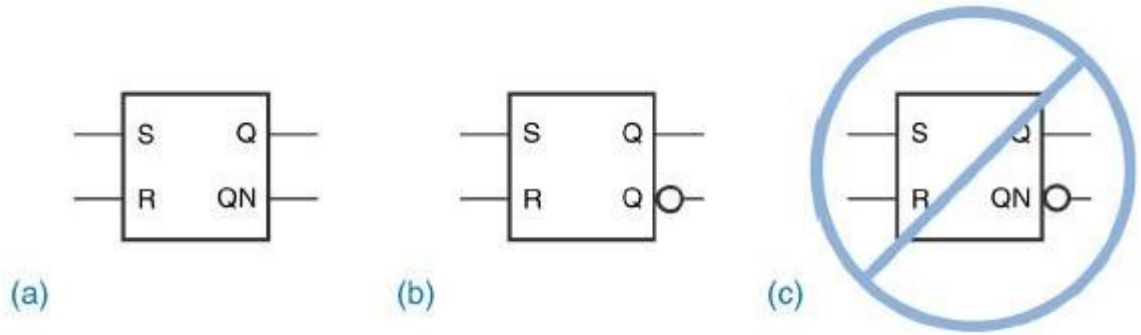
- The two most popular varieties of storage cells used to build sequential circuits are: latches and flip-flops.
  - **Latch:** level sensitive storage element
  - **Flip-Flop:** edge triggered storage element

- Common examples of latches:
  S-R latch, \S-\R latch, D latch (= gated D latch)

- Common examples of flip-flops:
  D-FF, D-FF with enable, Scan-FF, JK-FF, T-FF

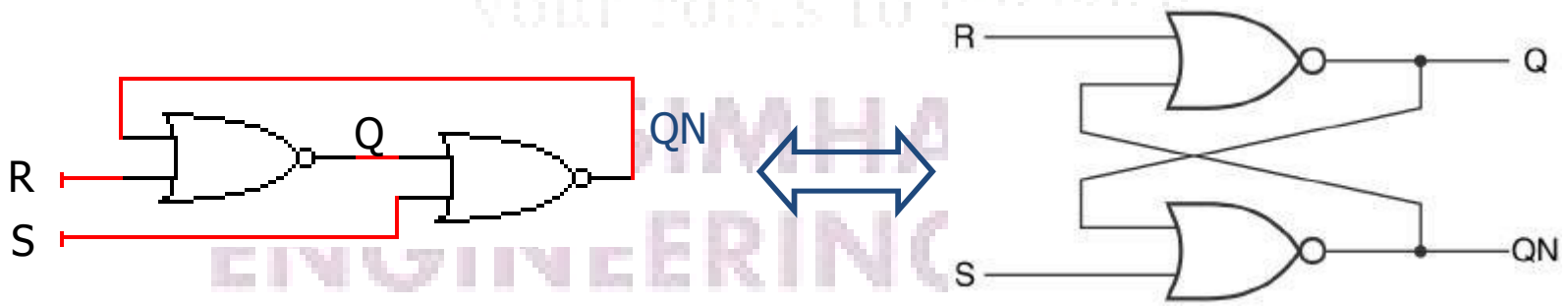
S-R (Set-Reset) Latch

X Y	NOR
0 0	1
0 1	0
1 0	0
1 1	0

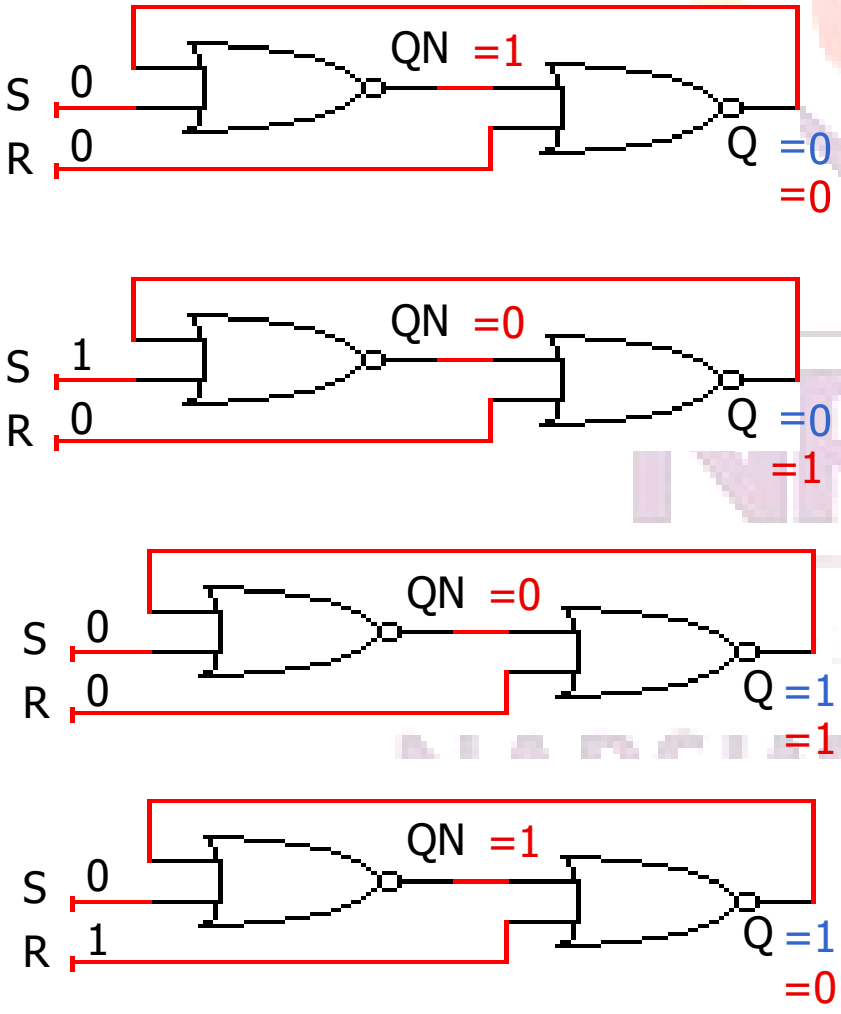


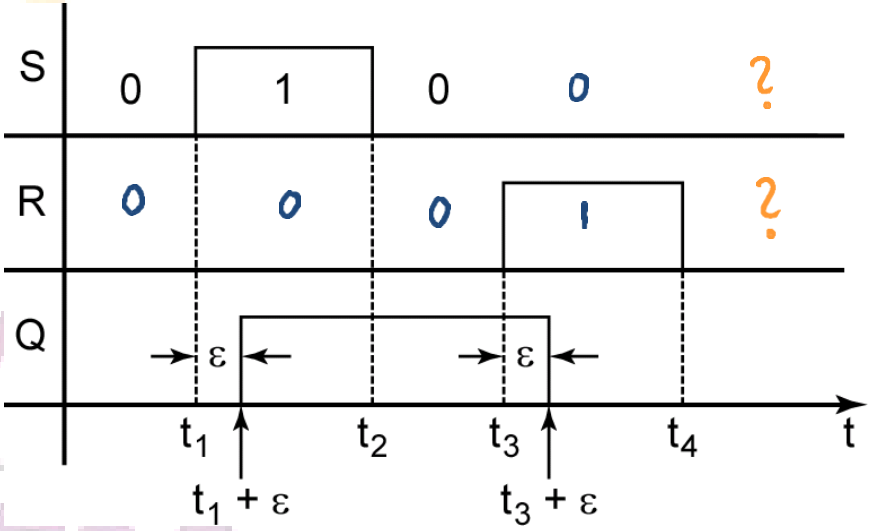
Symbols for an S-R latch: (a) without bubble; (b) preferred for bubbleto-bubble design; (c) incorrect because of double negation.

S-R latch: similar to inverter pair, with capability to force output to 0 (reset=1) or 1 (set=1)

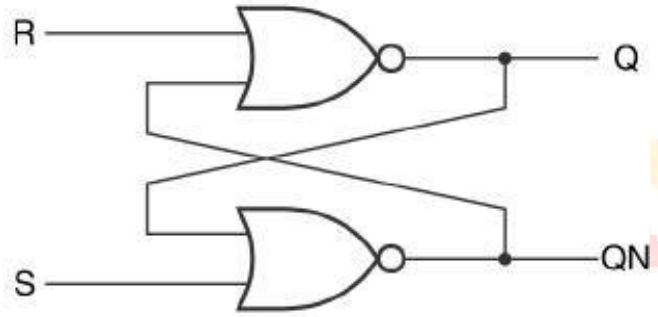


S-R latch operation

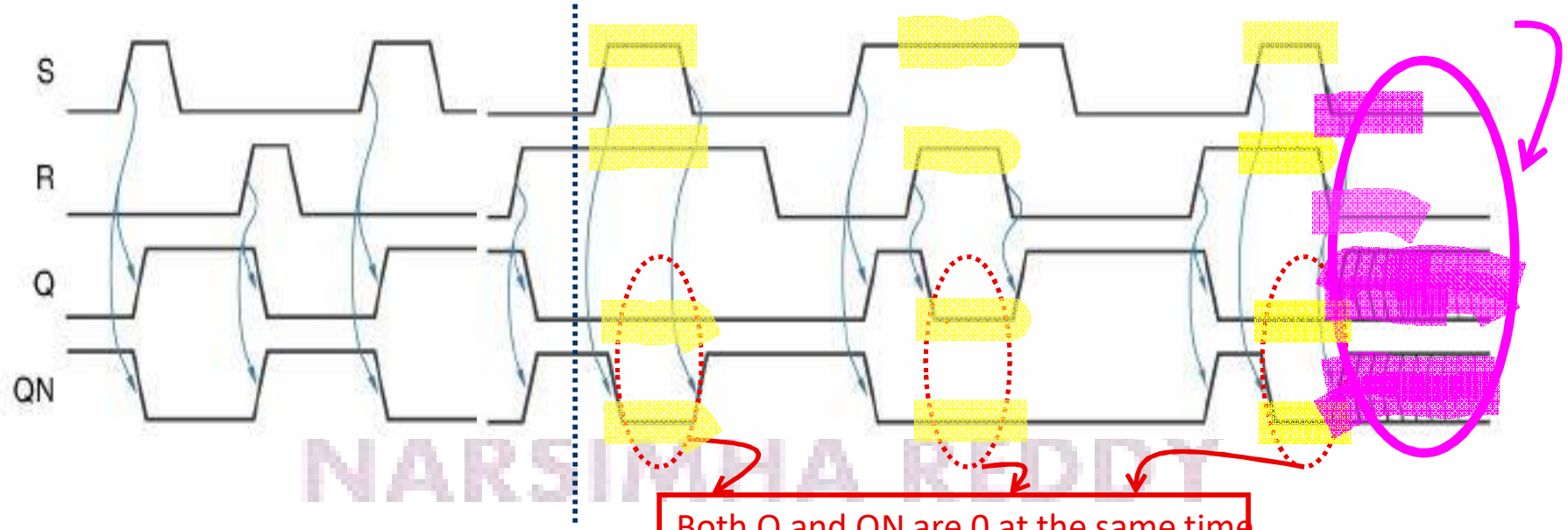


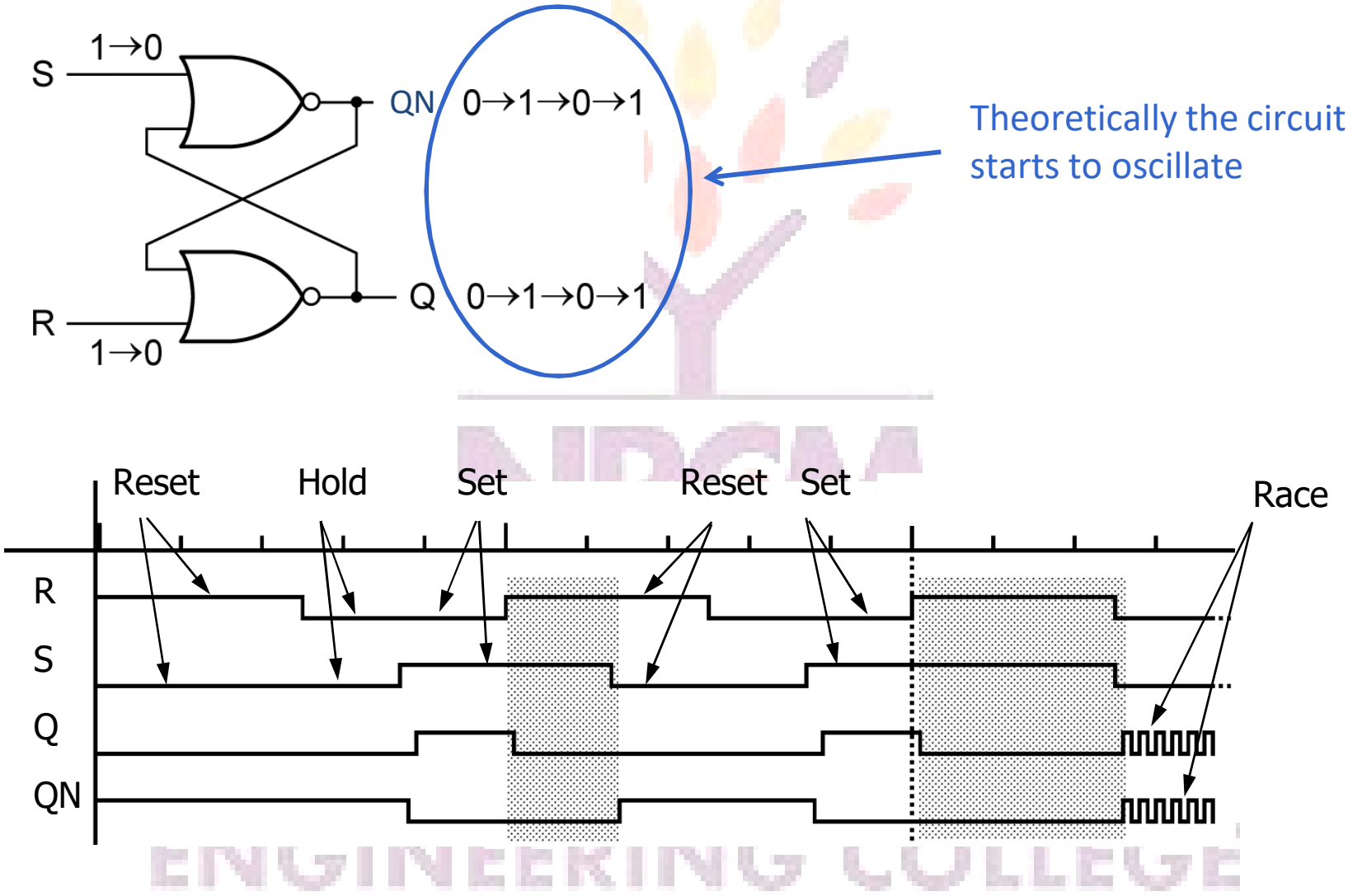


$S(t)$	$R(t)$	$Q(t)$	$Q(t+\epsilon)$	$QN(t+\epsilon)$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0		
1	1	1		

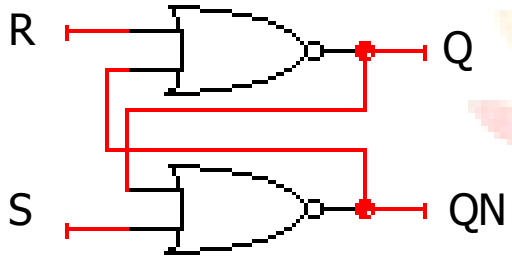


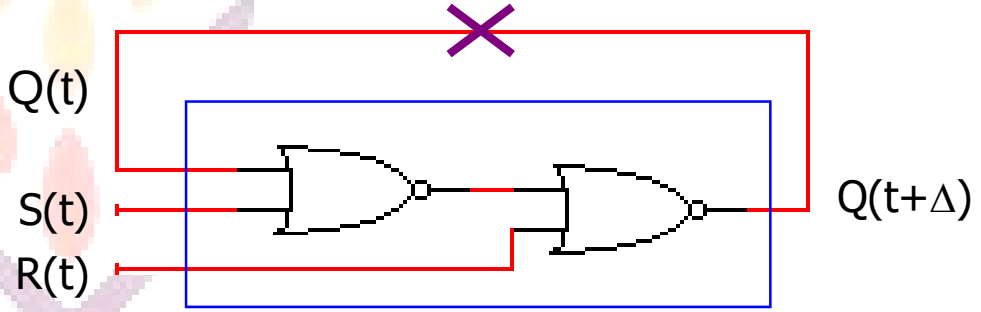
S	R	Q	QN	
0	0	last Q	last QN	(hold)
0	1	0	1	(reset)
1	0	1	0	(set)
1	1	0	0	(forbidden)




S
1→0
QN
0→1→0→1
Theoretically the circuit starts to oscillate
R
1→0
Q
0→1→0→1
Reset
Hold
Set
Reset
Set
Race
R
S
Q
QN

- Break feedback path





S(t)	R(t)	Q(t)	Q(t+Δ)	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	X	not allowed
1	1	1	X	

Q(t) \ S(t)	0	0	X	1
0	0	0	X	1
1	1	0	X	1

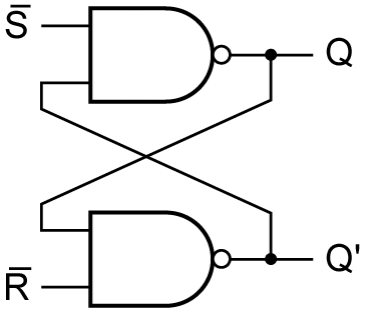
R(t)

next state equation:

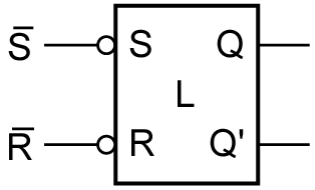
$Q(t+\Delta) = S(t) + R'(t)\,Q(t)$

$Q^+ = Q^* = S + R'\,Q$

a.k.a. characteristic equation

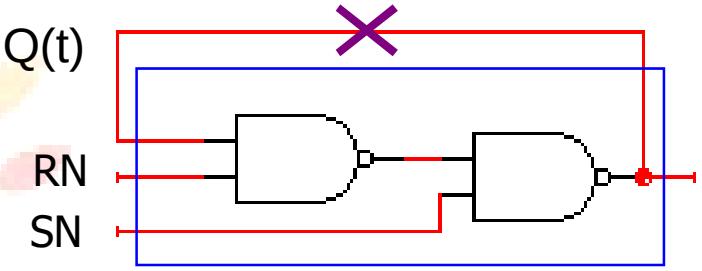


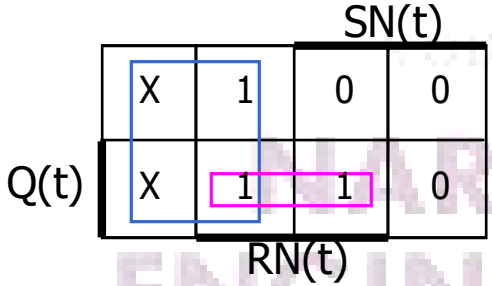
(a)



(b)

SN(t)	RN(t)	Q(t)	Q(t+Δ)	
1	1	0	0	hold
1	1	1	1	
1	0	0	0	reset
1	0	1	0	
0	1	0	1	set
0	1	1	1	
0	0	0	X	not allowed
0	0	1	X	

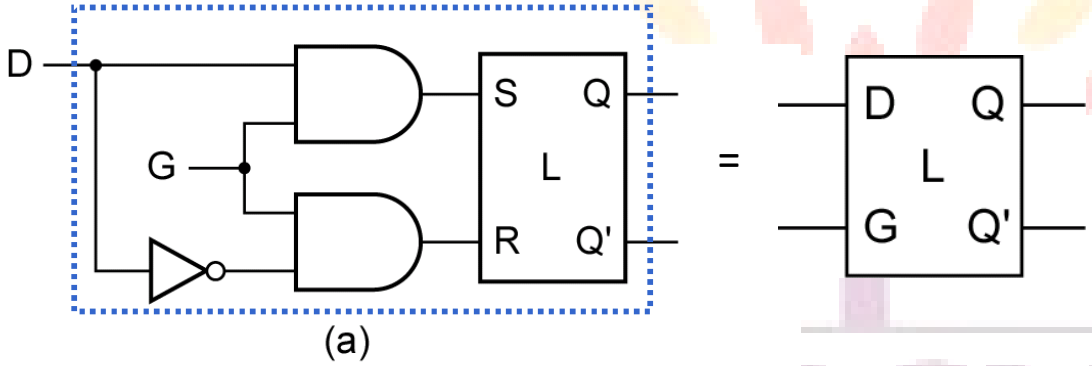




next state equation:

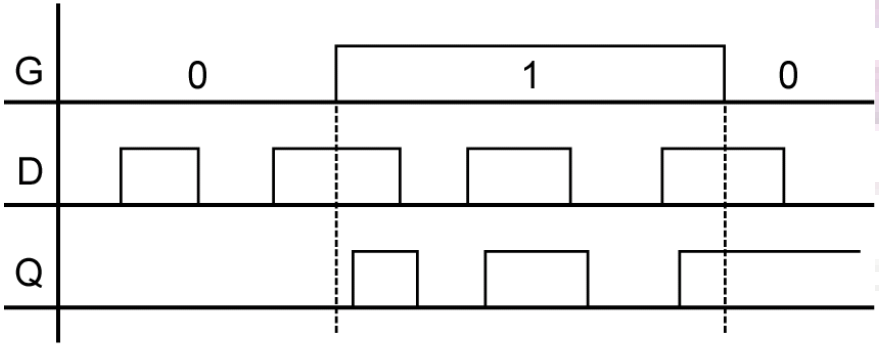
$Q(t+\Delta) = S'(t) + R(t)\,Q(t)$

$Q^+ = Q^* = S' + R\,Q$

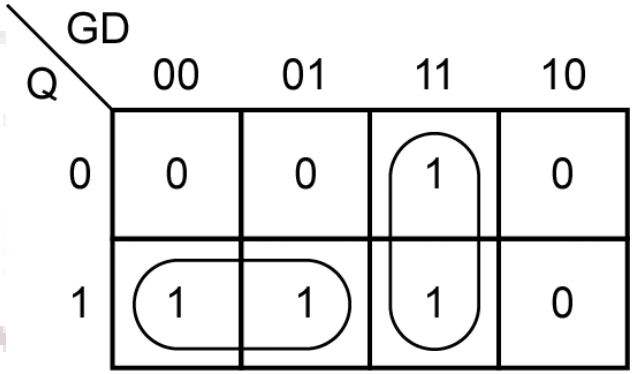


(a)

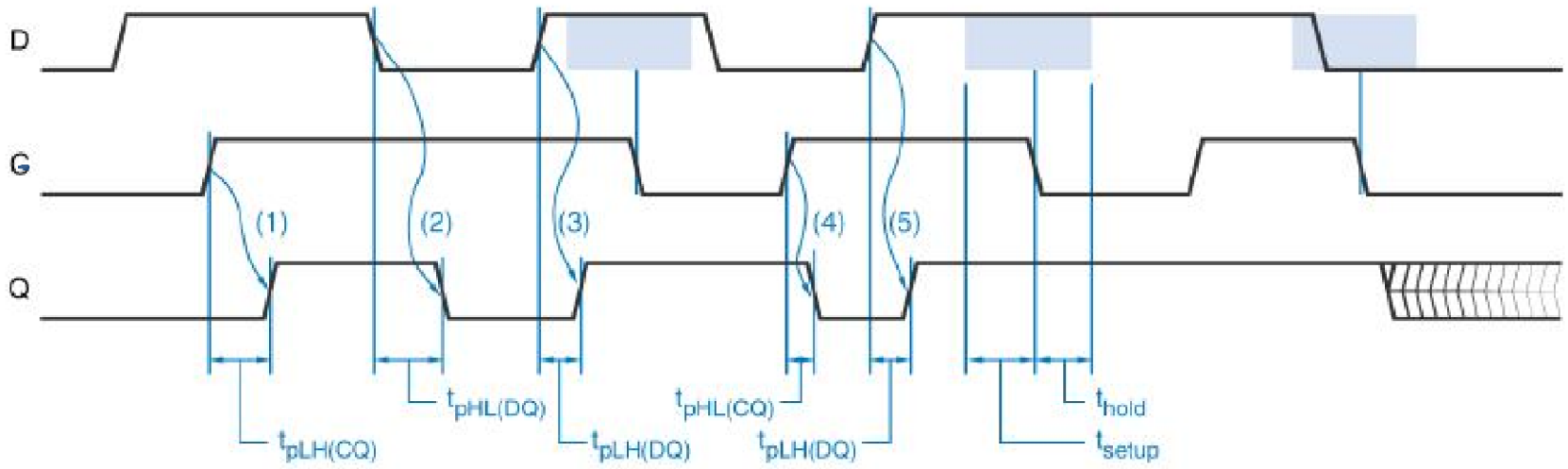
G	D	Q	QN
1	0	0	1
1	1	1	0
0	x	last Q	last QN



(b)

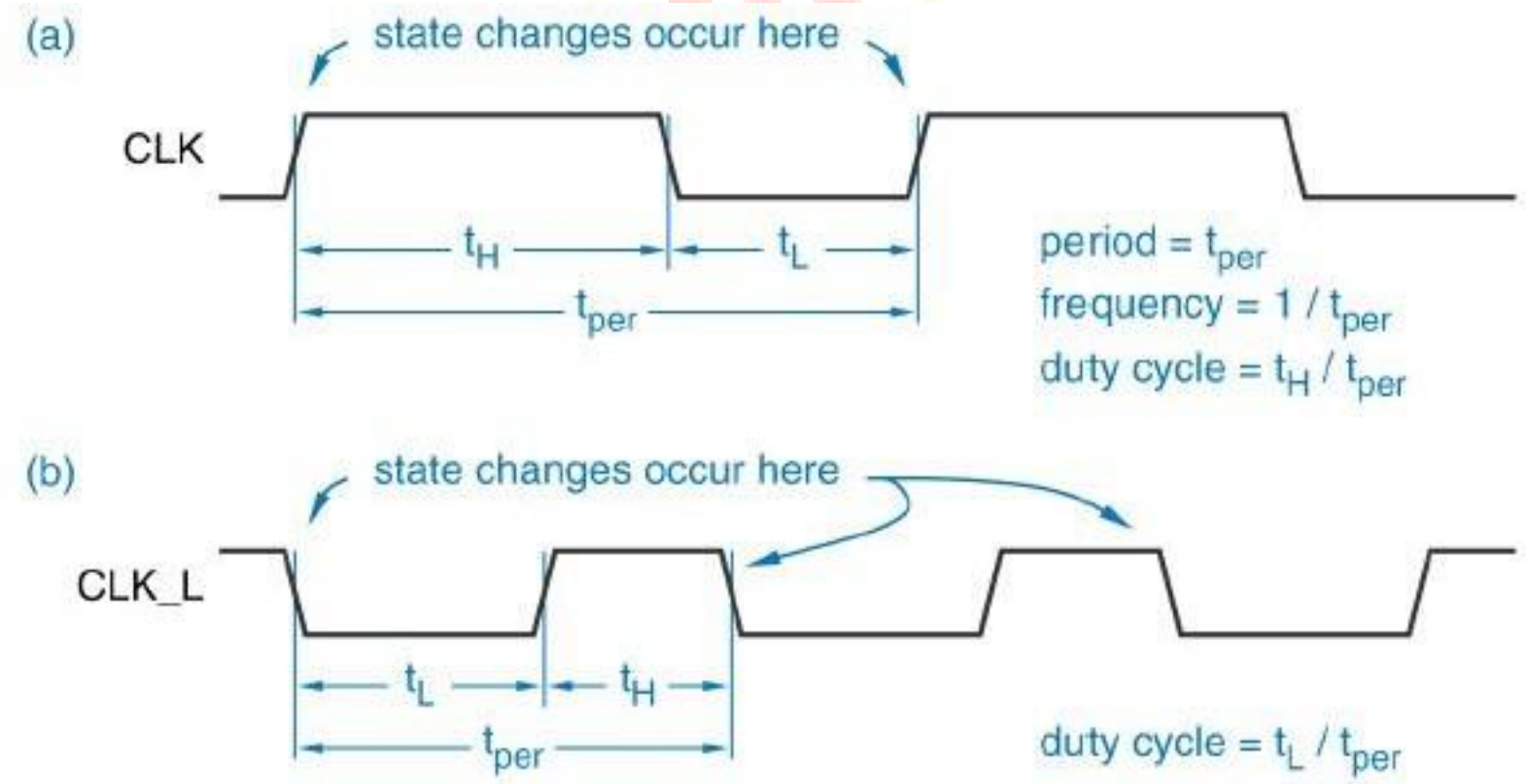


$Q^+ = G'Q + GD$



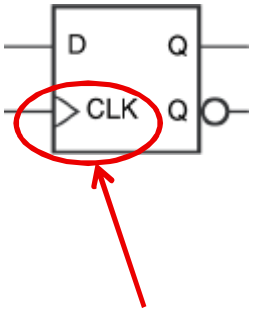
- The D Latch eliminates the S=R=1 problem of the SR latch
- However, violations of setup and hold time still cause metastability

- Clocks are regular periodic signals used to specify state changes



Clock signals: (a) active high; (b) active low.

D Flip-Flop (positive edge triggered)



Notice: the little triangle !

Functional Table

D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

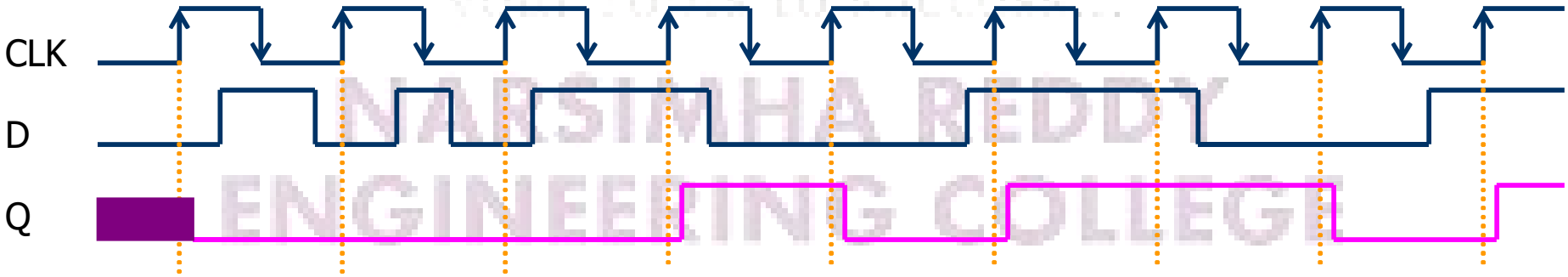
Truth Table

DQ	Q^+
0 0	0
0 1	0
1 0	1
1 1	1

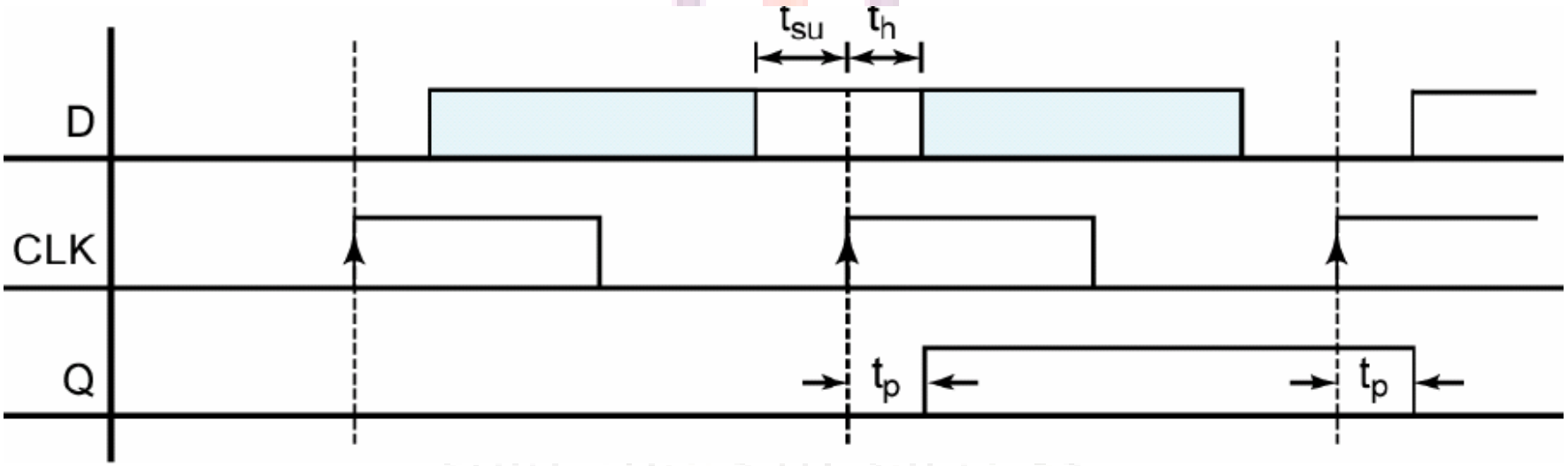
More compact Truth Table

D	Q^+
0	0
1	1

Next state equation: $Q^+ = D$



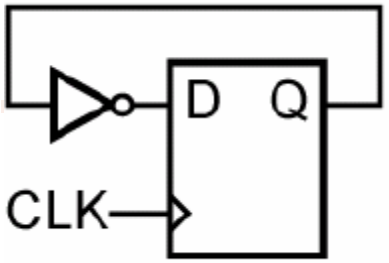
Setup and hold times for an edge-triggered DFF



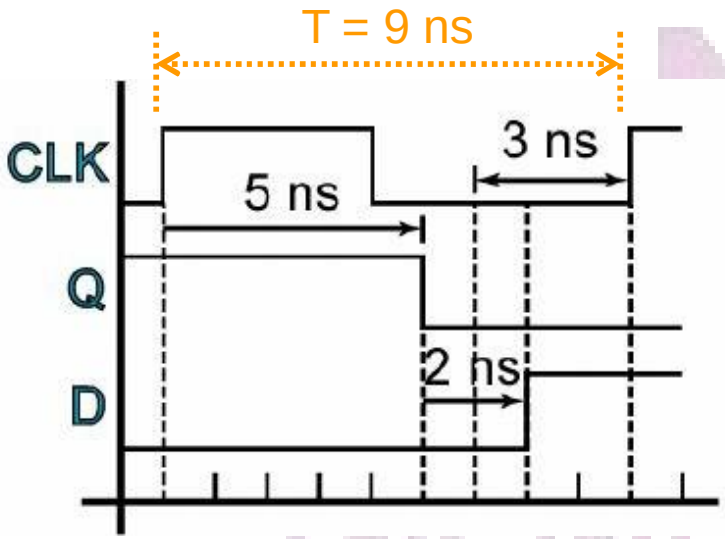
Setup and Hold Times for an Edge-Triggered D Flip-Flop

Minimum clock period T ?

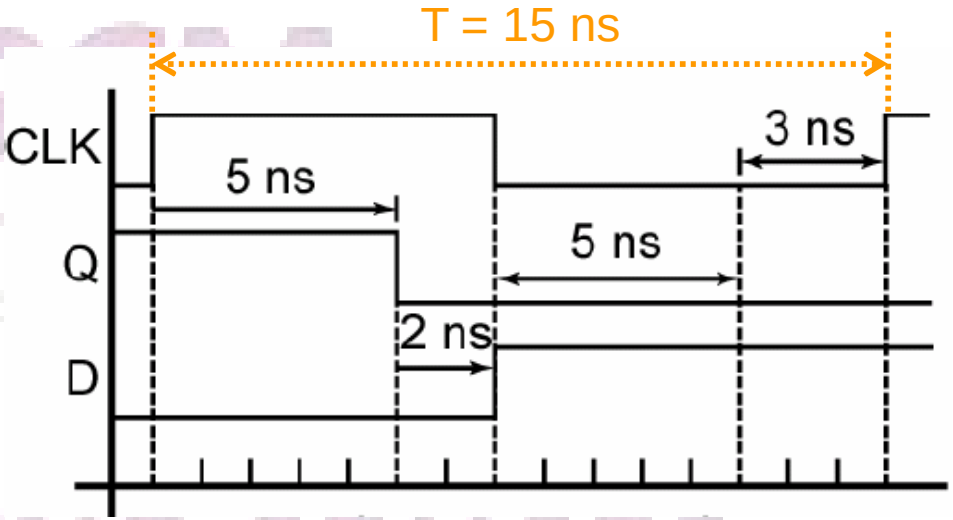
t_{pINV} = 2 ns
t_{pFF} = 5 ns
t_{suFF} = 3 ns



Simple flip-flop circuit



Example with T = 9 ns
Setup time not satisfied

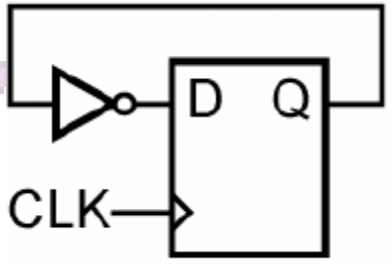


Example with T = 15 ns
Setup time satisfied

Minimum clock period T ? (cont'd)

Observation:

t_{hFF} doesn't affect this calculation

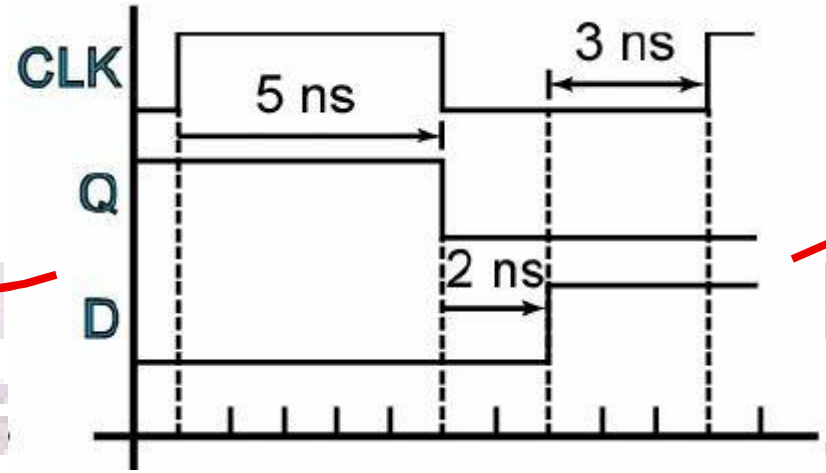


Simple flip-flop circuit

t_{pINV} = 2 ns

t_{pFF} = 5 ns

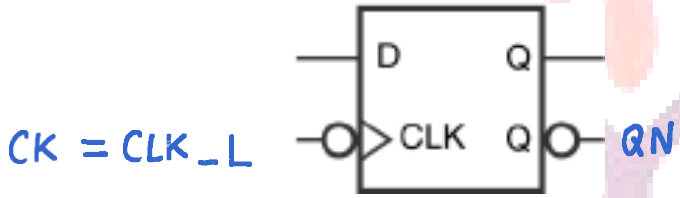
t_{suFF} = 3 ns



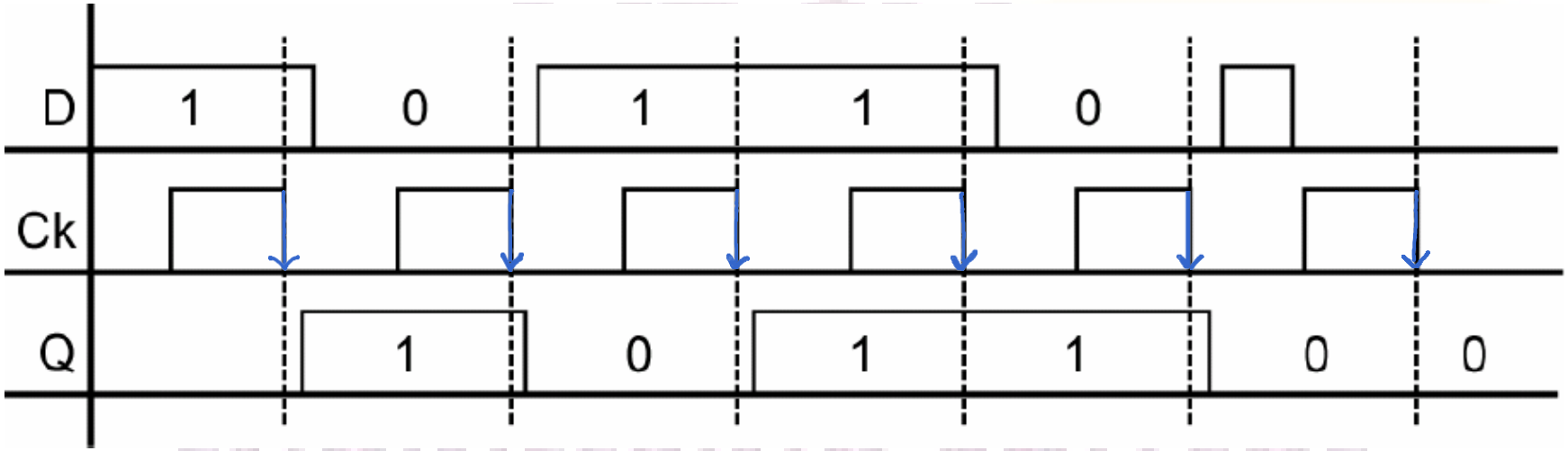
T_{min} = 10 ns

Determination of Minimum Clock Period

D Flip-Flop (negative edge triggered)



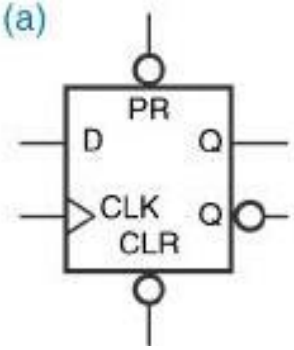
D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN



inputs sampled on falling edge; outputs change after falling edge

DFF with asynchronous preset and clear

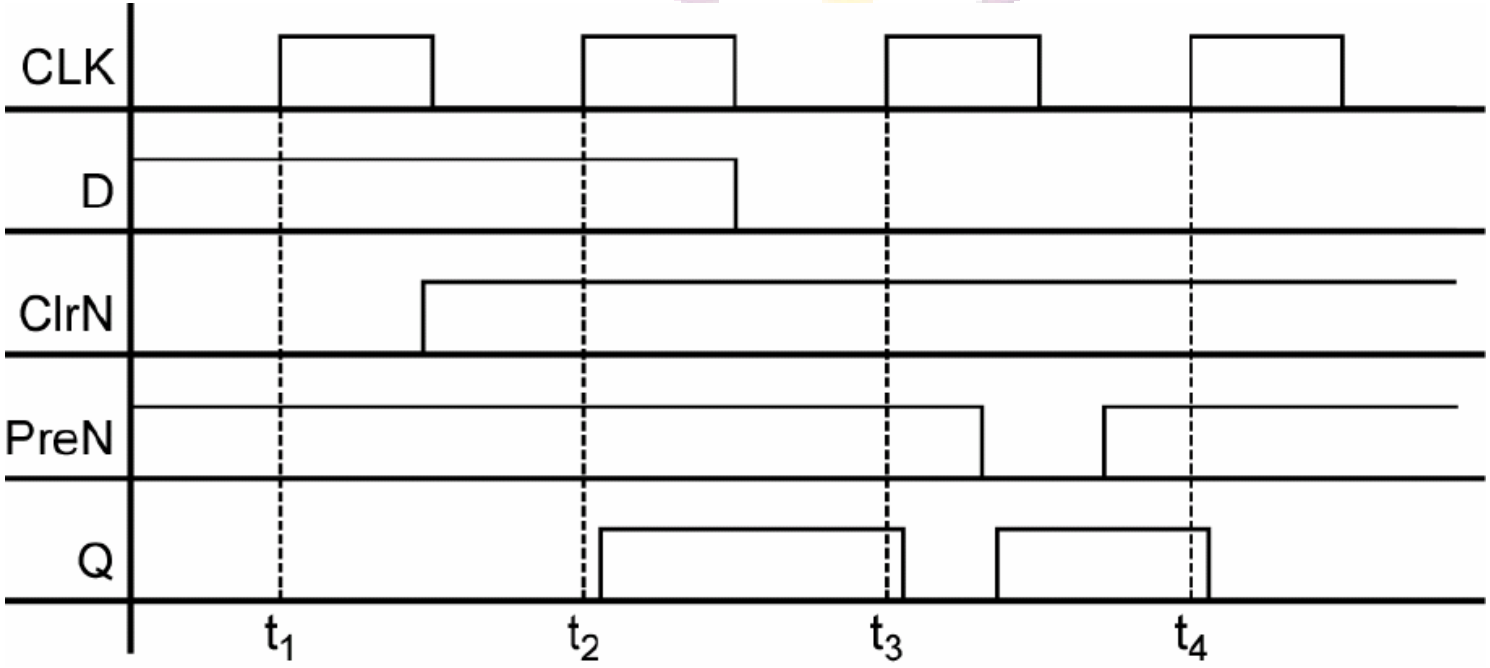
(a)

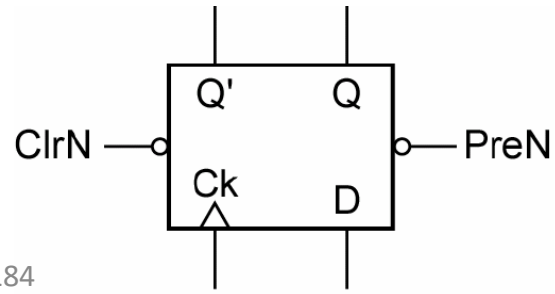


INPUTS				OUTPUTS		FUNCTION
$\overline{CLR}$	$\overline{PR}$	D	CK	Q	$\overline{Q}$	
L	H	X	X	L	H	CLEAR
H	L	X	X	H	L	PRESET
L	L	X	X	H	H	FORBIDDEN
H	H	L	↑	L	H	sample data
H	H	H	↑	H	L	sample data
H	H	X	↓	Q_n	$\overline{Q}_n$	NO CHANGE

X : Don't Care

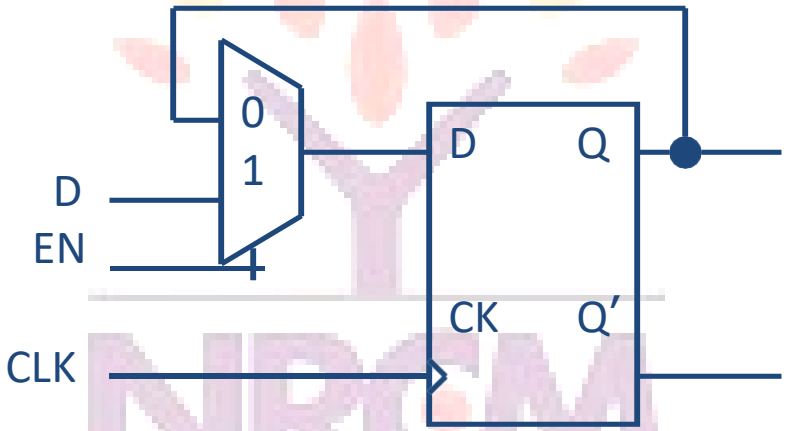
DFF with asynchronous preset and clear (cont'd)





CK	D	PreN	ClrN	Q^+
x	x	0	0	(not allowed)
x	x	0	1	1
x	x	1	0	0
↑	0	1	1	0
↑	1	1	1	1
0,1, ↓	x	1	1	Q (no change)

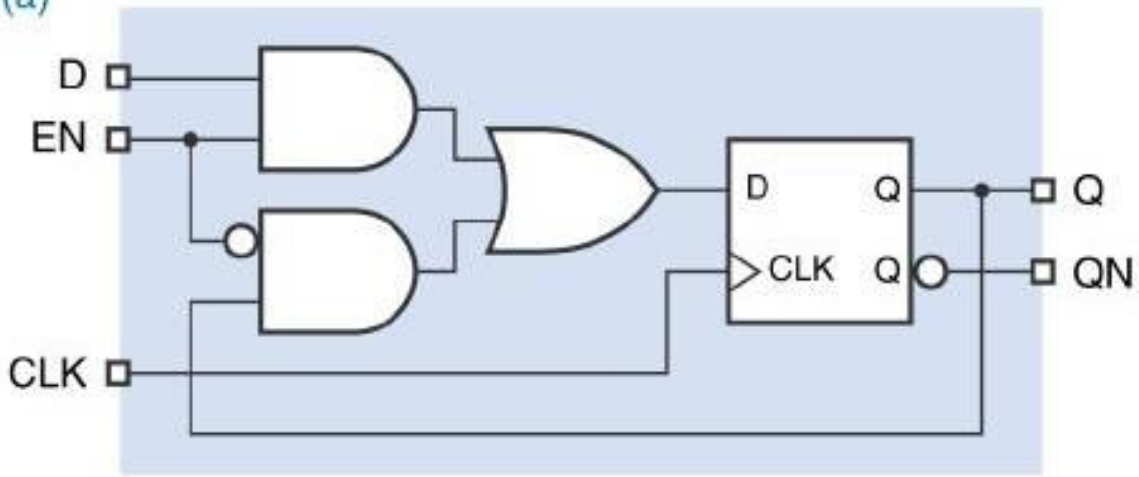
DFF with enable



Reliable alternative

DFF with enable (cont'd)

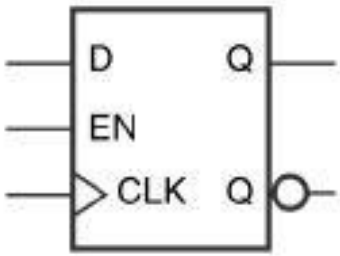
(a)



(b)

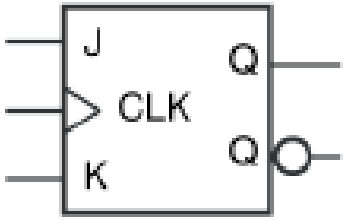
D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

(c)

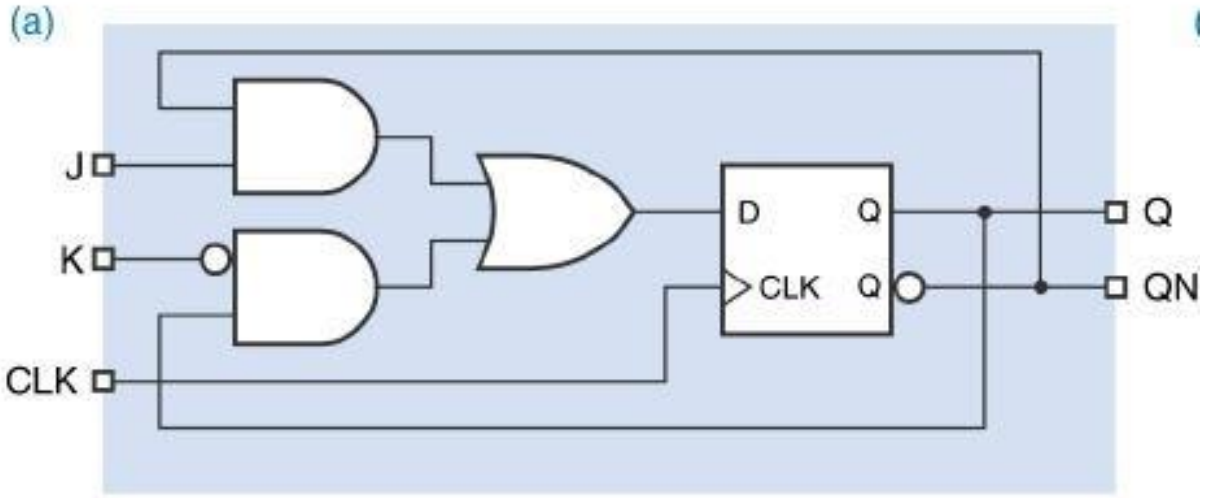


Positive-edge-triggered D flip-flop with enable: (a) circuit design; (b) function table; (c) logic symbol.

ered)



=



Functional Table

J	K	CLK	Q	QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN
0	0	↓	last Q	last QN
0	1	↓	0	1
1	0	↓	1	0
1	1	↓	last QN	last Q

Truth Table

JKQ	Q^+
000	0
001	1
010	0
011	0
100	1
101	1
110	1
111	0

More Compact Truth Table

J K	Q^+
0 0	Q
0 1	0
1 0	1
1 1	Q′

Next state equation:

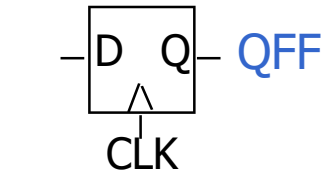
$$Q^+ = JQ' + K'Q$$

ops

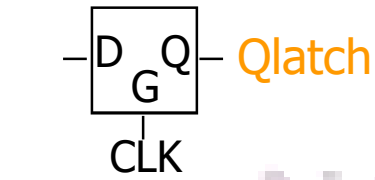
Device Type	Characteristic Equation
S-R latch	$Q^* = S + R' \cdot Q$
D latch	$Q^* = D$
D flip-flop	$Q^* = D$
D flip-flop with enable	$Q^* = EN \cdot D + EN' \cdot Q$
J-K flip-flop	$Q^* = J \cdot Q' + K' \cdot Q$
T flip-flop	$Q^* = Q' \cdot T + T \cdot Q$

Latch and flip-flop characteristic equations.

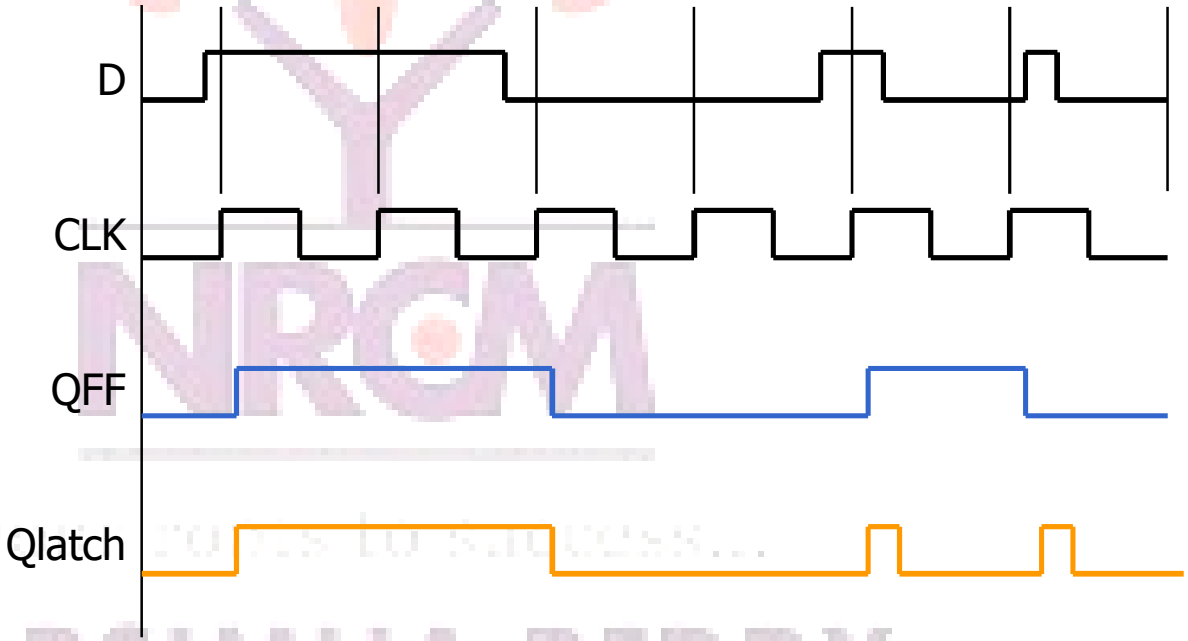
Comparison of latches and flip-flops



positive edge-triggered flip-flop



transparent (level-sensitive) latch

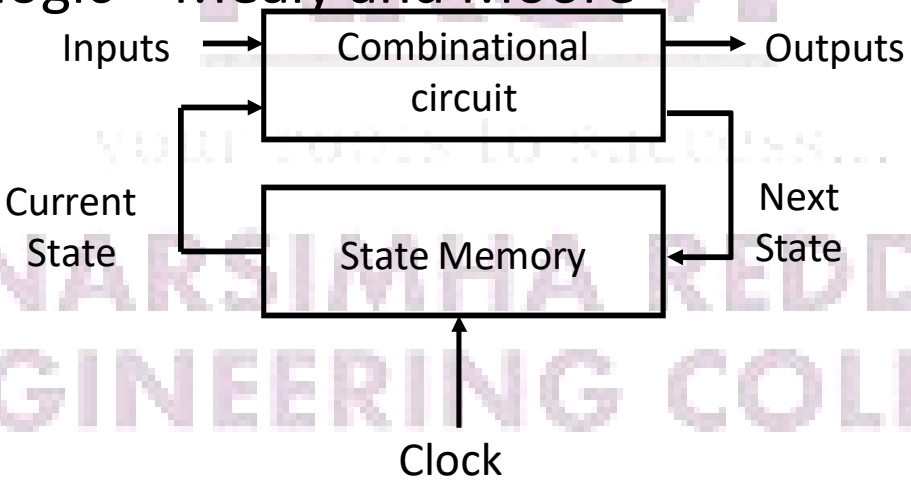


behavior is the same unless input changes while the clock is high

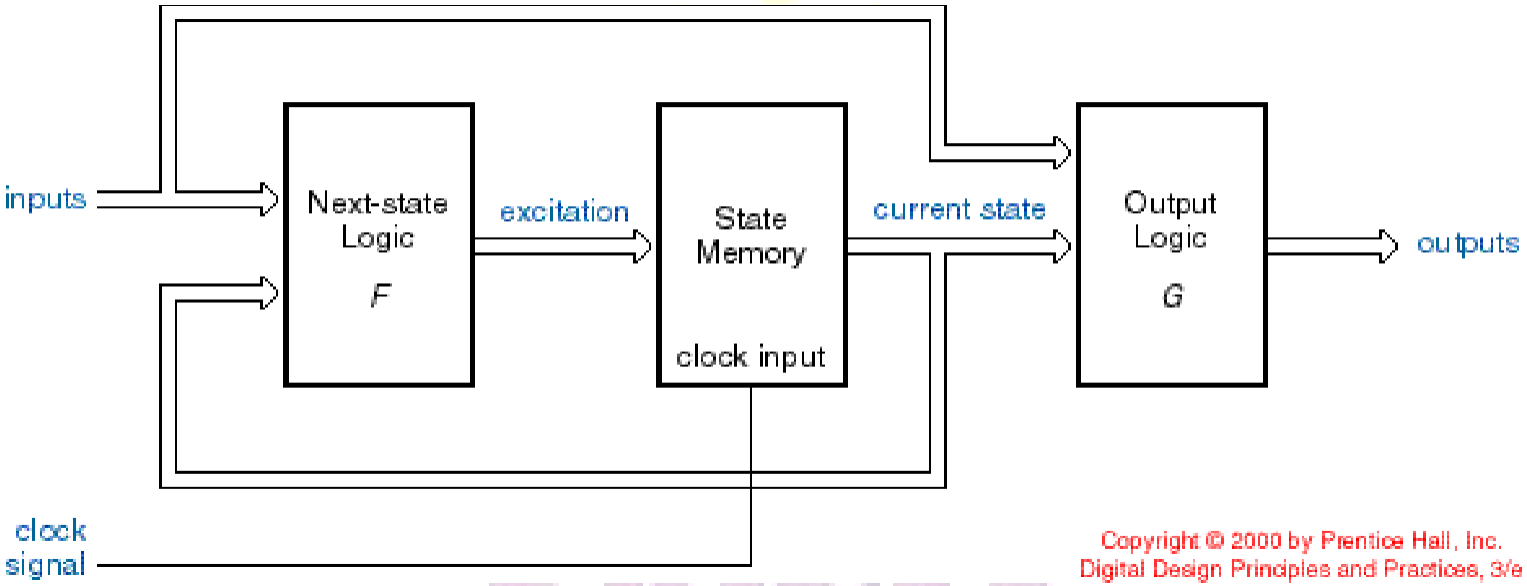
Synchronous Sequential Circuit Analysis

Synchronous Sequential Circuit

- State Memory – A set of *n* edge-triggered flip-flops that store the current state of the machine
  - All flip-flops are triggered from the same master clock signal
  - All change state together
- Combinational circuit
  - Next state logic
  - Output logic – Mealy and Moore



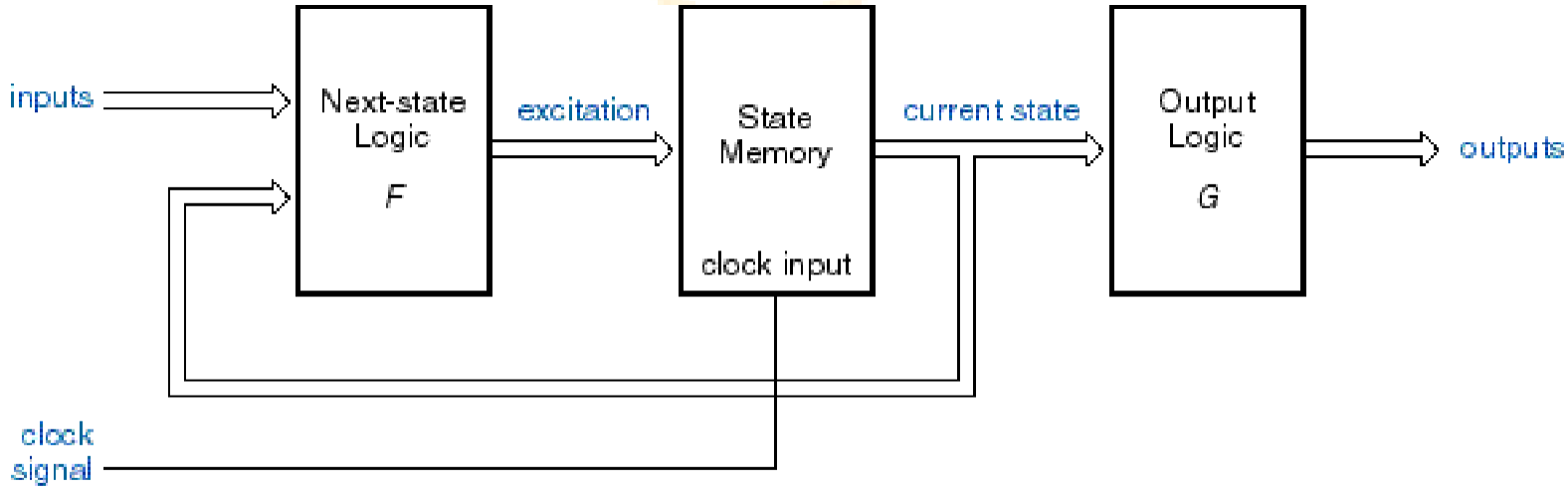
Me



next state = F (current state, inputs)

outputs = G (current state, inputs)

Mc



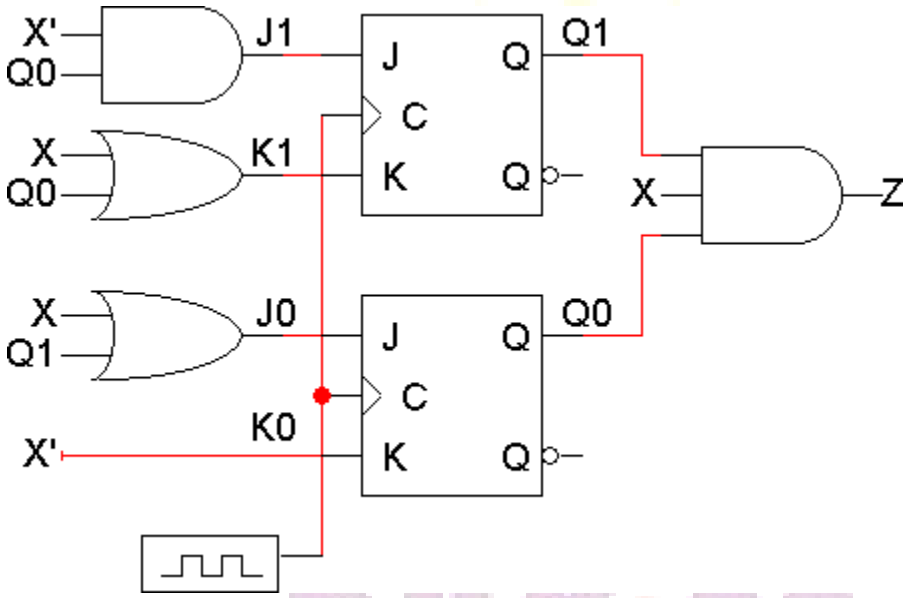
next state = F (current state, inputs)

outputs = G (current state)

An

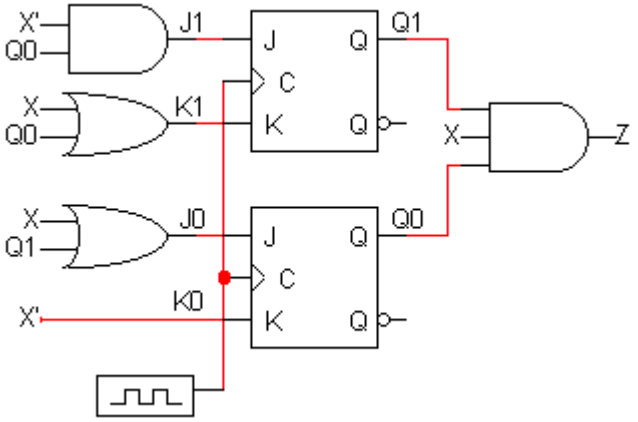
- Characterize as Mealy or Moore machine
- Determine next state equations, i.e., find the function F
  - next state = F (current state, inputs)
- Determine output equations
  - Meally: outputs = F (current state, inputs), or
  - Moore: outputs = F (current state)
- Express as machine behavior
  - State table, or
  - State diagram
- Formulate English description of machine behavior

An



- A sequential circuit with two JK flip-flops
- State or memory: Q1Q0
- One input: X; One output: Z

State table of example circuit



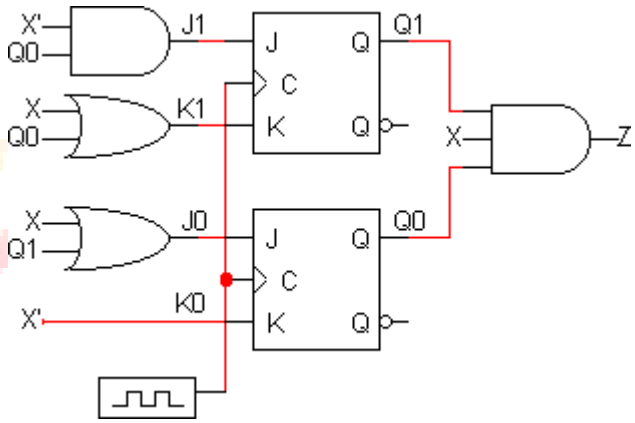
Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Output Equations

- From the diagram, you can see that

$$Z = Q_1 Q_0 X$$

Mealy model circuit !!!



Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
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

Next State Equations – Q(t+1)

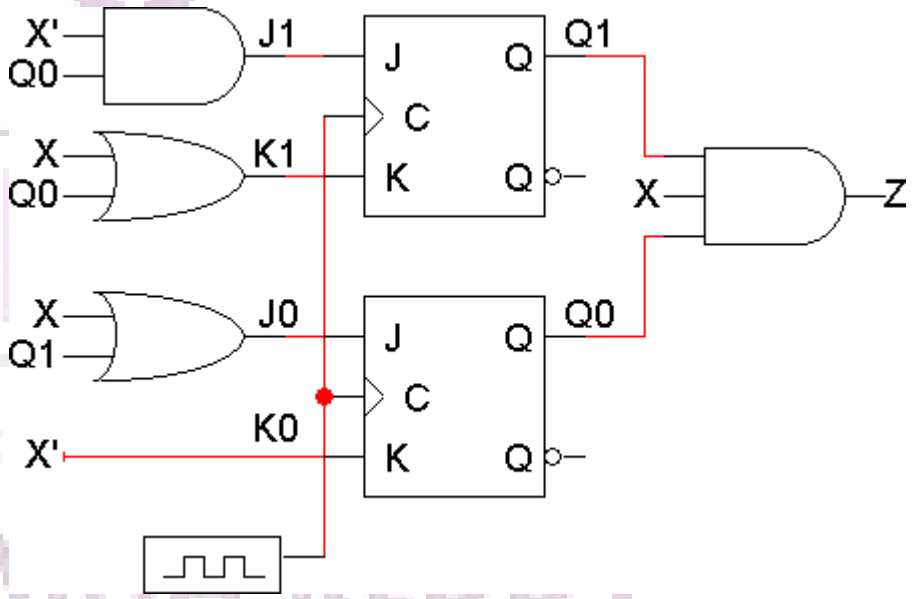
- Find the flip-flop input equations/excitation equations
- Substitute excitation equations in the flip-flop's characteristic equation

$J_1 = X' Q_0$

$K_1 = X + Q_0$

$J_0 = X + Q_1$

$K_0 = X'$



Next State Equations – Q(t+1)

- Excitation equations:
  - $J_1 = X' Q_0$ and $K_1 = X + Q_0$
  - $J_0 = X + Q_1$ and $K_0 = X'$
- Characteristic equation of the JK flip-flop:
  - $Q(t+1) = K'Q(t) + JQ'(t)$
- Next state equations:
  - $Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$

    $= (X + Q_0(t))' Q_1(t) + X' Q_0 (t) Q_1'(t)$

    $= X' (Q_0(t)' Q_1(t) + Q_0(t) Q_1(t)')$

    $= X' (Q_0(t) \oplus Q_1(t))$

  - $Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$

    $= X Q_0(t) + (X + Q_1(t)) Q_0'(t)$

    $= X + Q_0(t)' Q_1(t)$

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
  - $Q_1=0, Q_0=0, X= 0 \Rightarrow Q_1(t+1)= 0$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$
  - $Q_1=0, Q_0=0, X= 0 \Rightarrow Q_0(t+1)= 0$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
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

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
  - $Q_1=0, Q_0=1, X= 1 \Rightarrow Q_1(t+1)= 0$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$
  - $Q_1=0, Q_0=1, X= 1 \Rightarrow Q_0(t+1)= 1$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1			0
0	1	0			0
0	1	1	0	1	0
1	0	0			0
1	0	1			0
1	1	0			0
1	1	1			1

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

State Table & Characteristic Table

- The general JK flip-flop characteristic equation is:

$$Q(t+1) = K'Q(t) + JQ'(t)$$

- We can also determine the next state for each input/current state combination directly from the characteristic table

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

State Table & Characteristic Table

- With these equations, we can make a table showing J_1, K_1, J_0 and K_0

  for the different combinations of present state Q_1Q_0 and input X

$$J_1 = X' Q_0 \qquad J_0 = X + Q_1$$

$$K_1 = X + Q_0 \qquad K_0 = X'$$

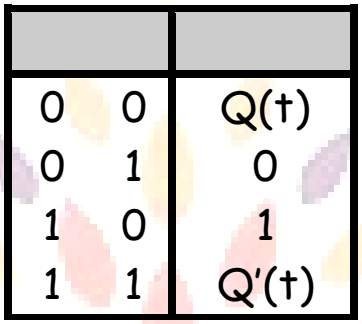
Present State		Inputs	Flip-flop Inputs			
Q_1	Q_0	X	J_1	K_1	J_0	K_0
0	0	0	0	0	0	1
0	0	1	0	1	1	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	1	1	0
1	1	0	1	1	1	1
1	1	1	0	1	1	0

Sta

0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Present State		Inputs	FF Inputs				Next State	
Q_1	Q_0	X	J_1	K_1	J_0	K_0	Q_1	Q_0
0	0	0	0	0	0	1		
0	0	1	0	1	1	0		
0	1	0	1	1	0	1	1	
0	1	1	0	1	1	0		
1	0	0	0	0	1	1		
1	0	1	0	1	1	0		
1	1	0	1	1	1	1		
1	1	1	0	1	1	0		

Sta



0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Present State		Inputs	FF Inputs				Next State	
Q_1	Q_0	X	J_1	K_1	J_0	K_0	Q_1	Q_0
0	0	0	0	0	0	1		
0	0	1	0	1	1	0		
0	1	0	1	1	0	1	1	0
0	1	1	0	1	1	0		
1	0	0	0	0	1	1		
1	0	1	0	1	1	0		
1	1	0	1	1	1	1		
1	1	1	0	1	1	0		

A c

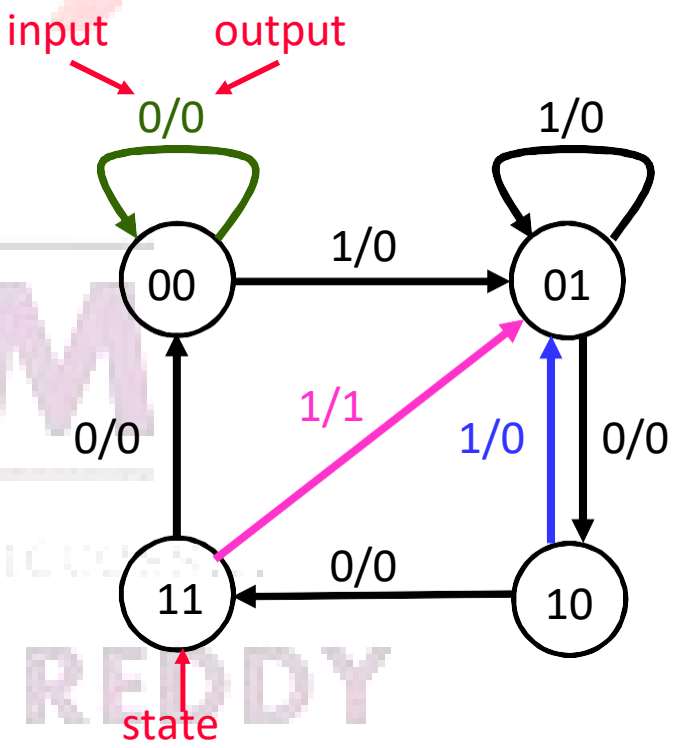
Present State Q_1	Q_0	Inputs X	Next State Q_1	Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Present State Q1	Q0	Next State Input X= 0		Input X= 1		Output Z X= 0	X= 1
0	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	1

Sta

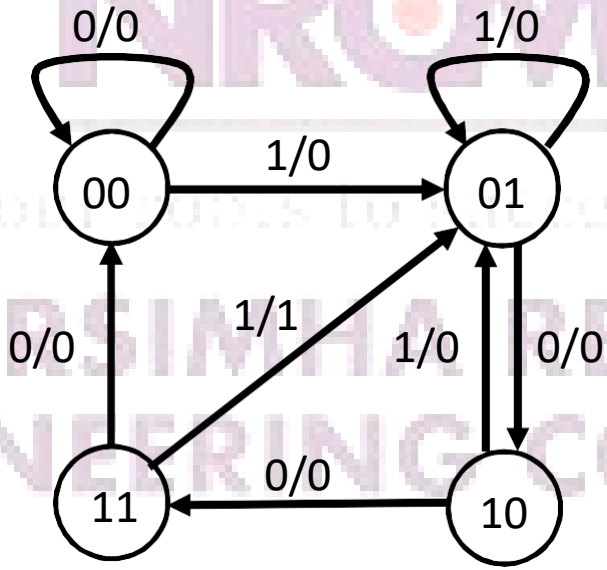
- We can also represent the state table graphically with a state diagram
- A diagram corresponding to our example state table is shown below

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

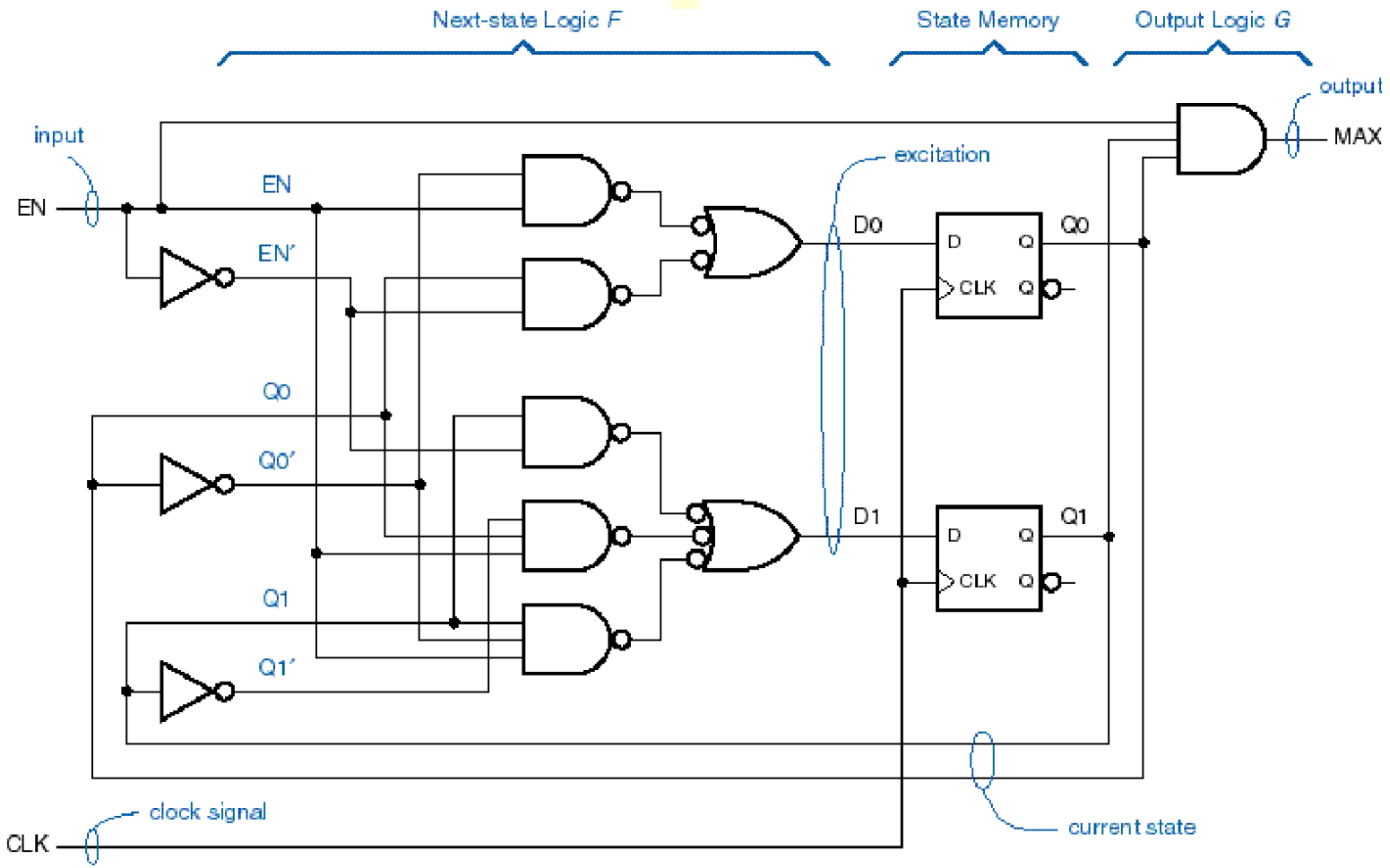


Sizes of state diagrams

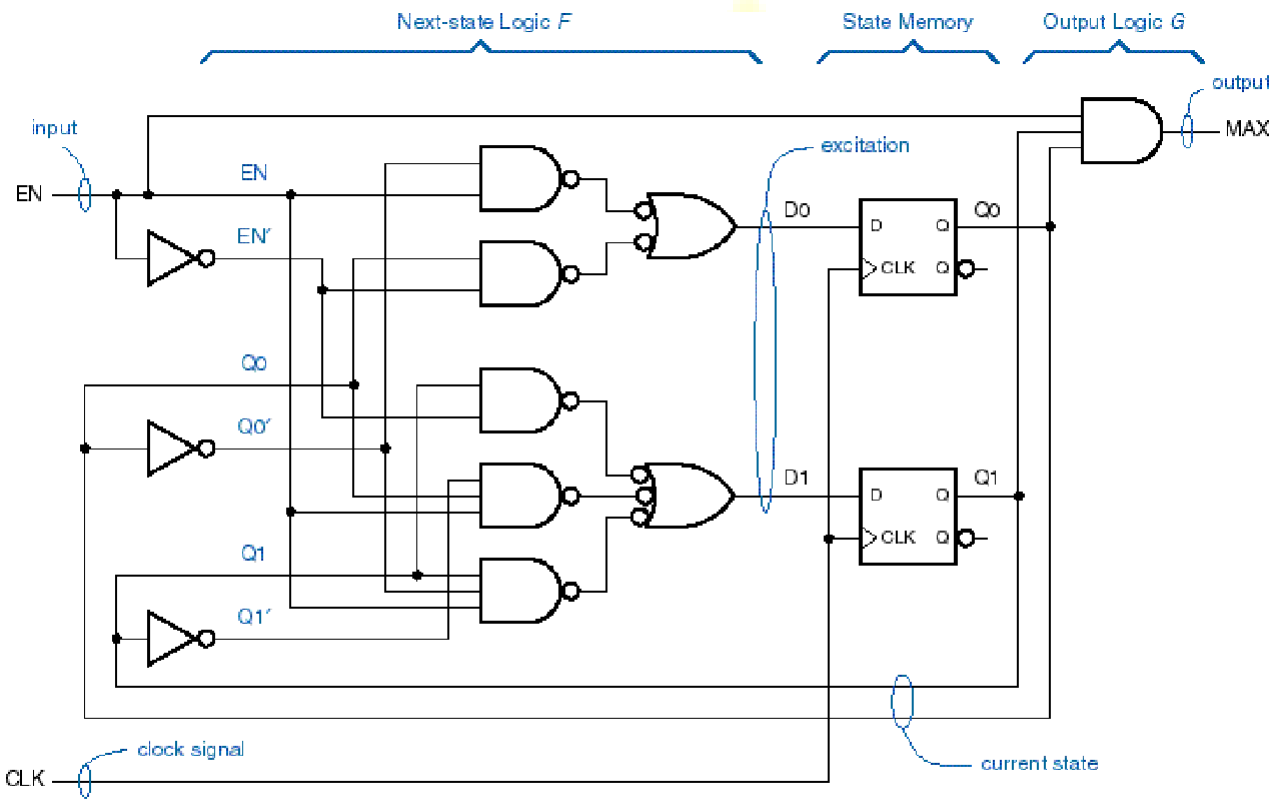
- Always check the size of your state diagrams
  - If there are *n* flip-flops, there should be 2^n nodes in the diagram
  - If there are *m* inputs, then each node will have 2^m outgoing arrows
- In our example,
  - We have two flip-flops, and thus four states or nodes.
  - There is one input, so each node has two outgoing arrows.



An

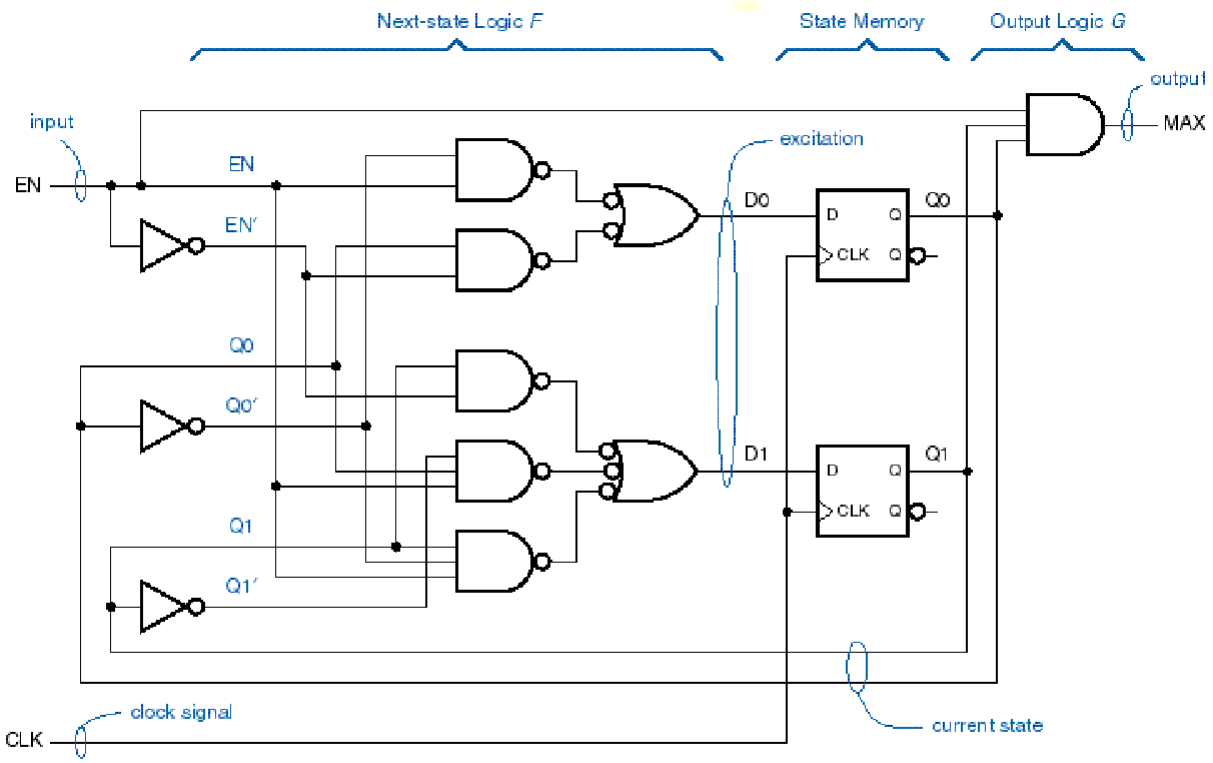


Ex



- $D_0 = EN' Q_0 + EN Q_0'$
- $D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$

Ne



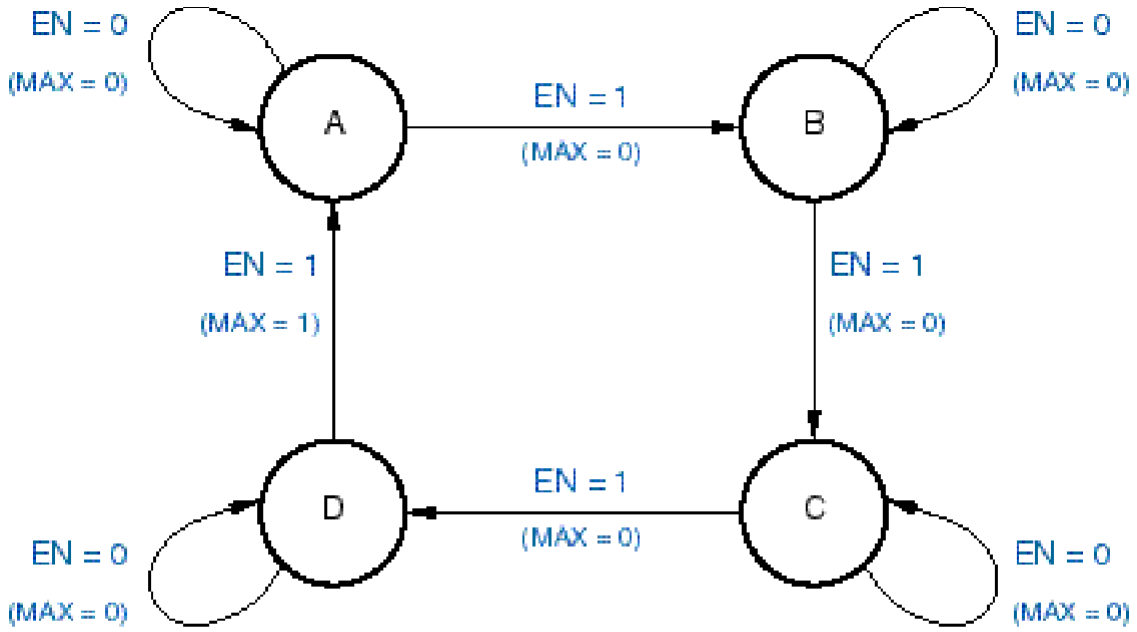
- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = EN Q_1 Q_0$

Mealy State Table

- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = EN Q_1 Q_0$

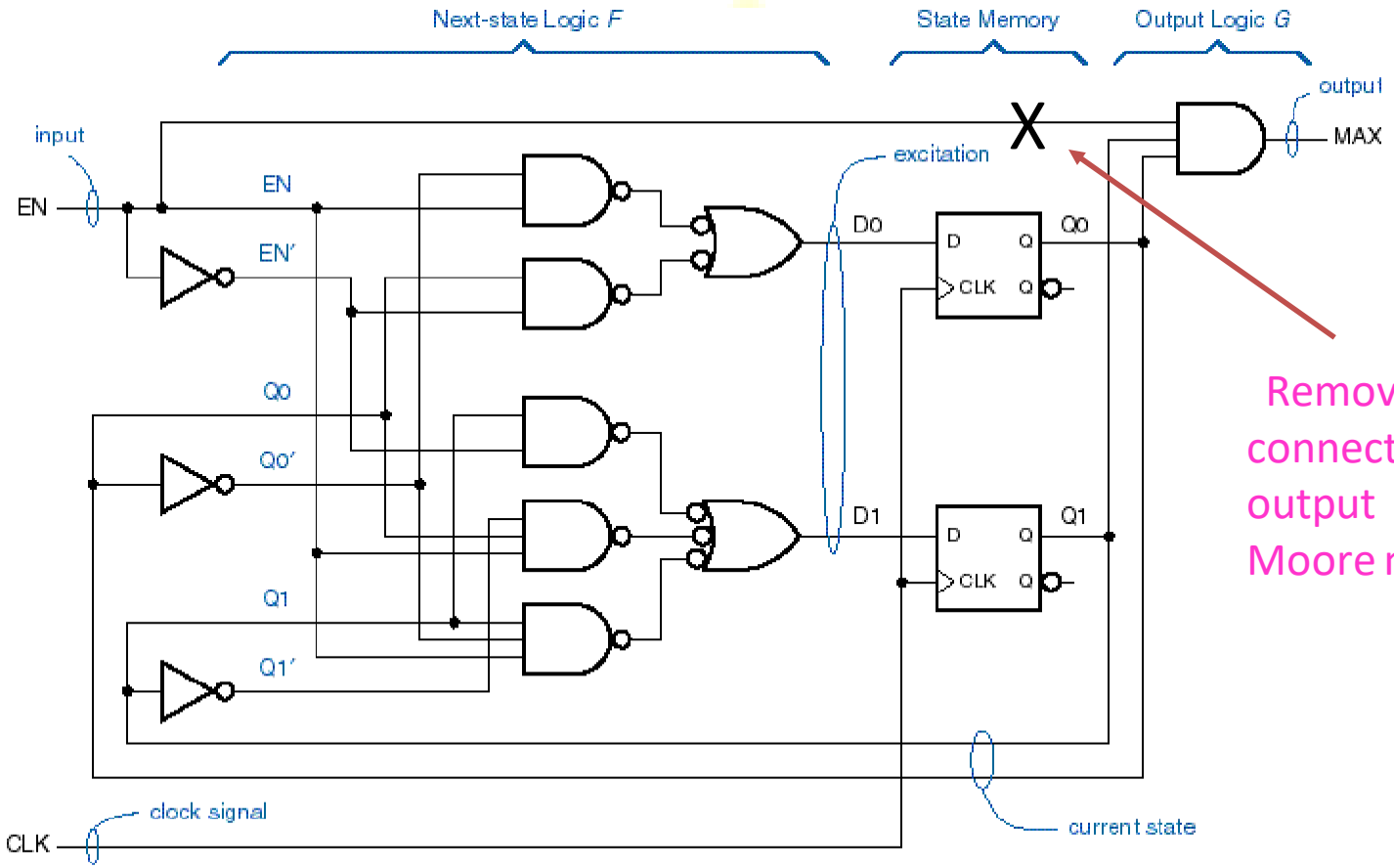
Present State		Next State				Output MAX	
		Input EN= 0		Input EN= 1			
Q1	Q0					X= 0	X= 1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

Me



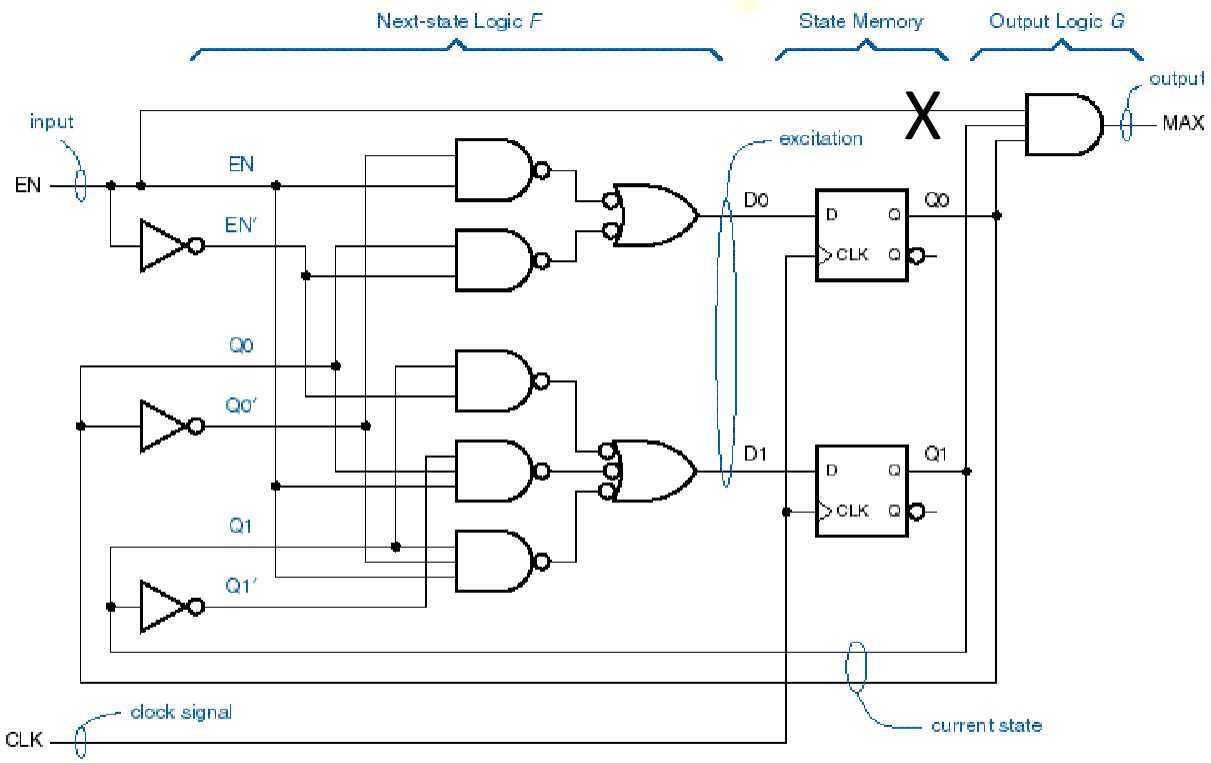
Present State		Next State				Output MAX	
		Input EN= 0		Input EN= 1			
Q1	Q0					X= 0	X= 1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

Mc



Remove input connection to output logic => Moore machine

Ne



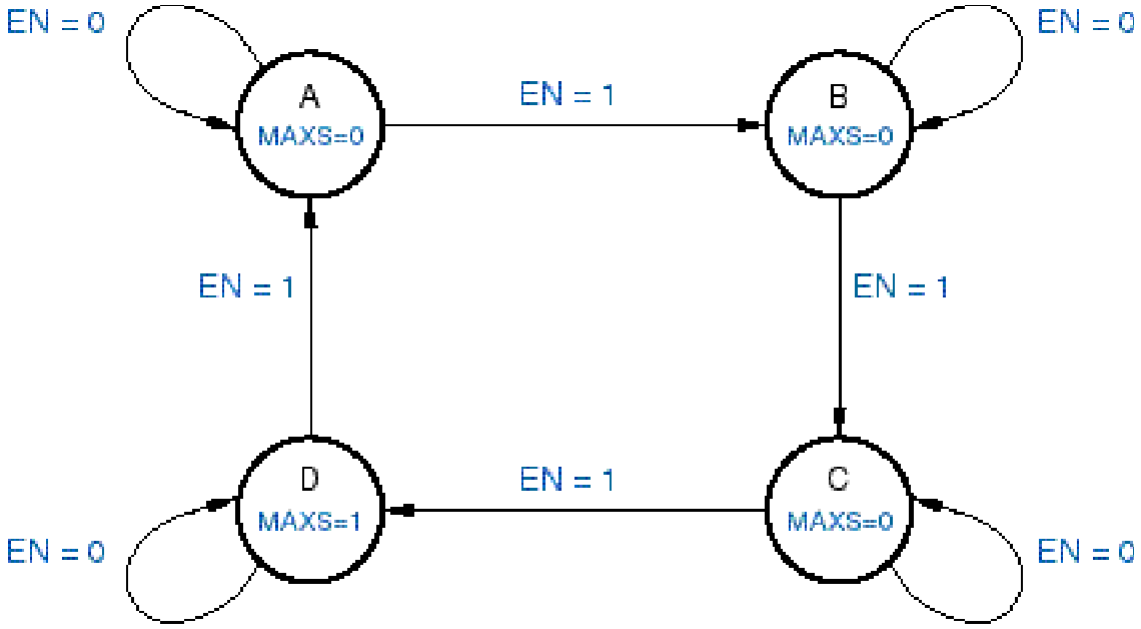
- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = Q_1 Q_0$

Moore State Table

- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = Q_1 Q_0$

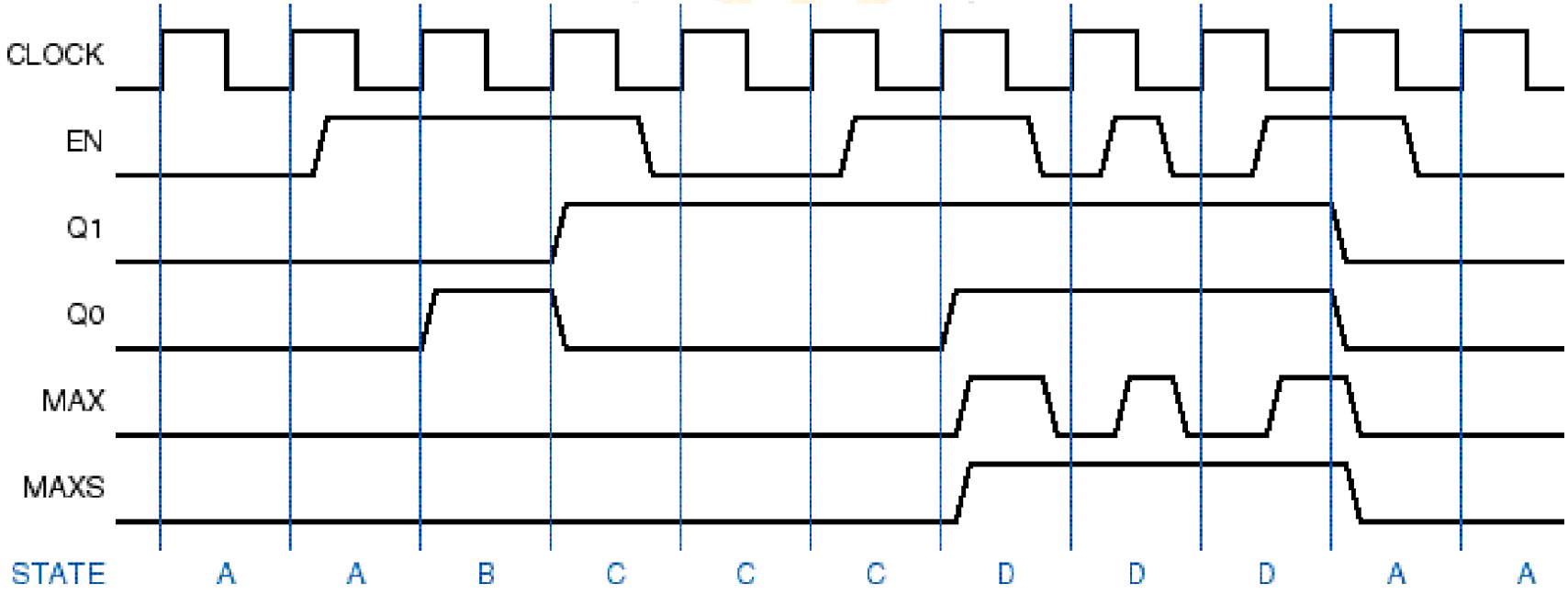
Present State		Next State				Output
		Input EN= 0		Input EN= 1		MAX
Q1	Q0					
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

Mo



Present State Q1	Present State Q0	Next State Input EN= 0		Next State Input EN= 1		Output MAX X= 0	Output MAX X= 1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

Sta



- MAX : Output of the Mealy circuit
- MAXS : Output of the Moore circuit

Shift register

Circuit for simple shift register

Basic applications

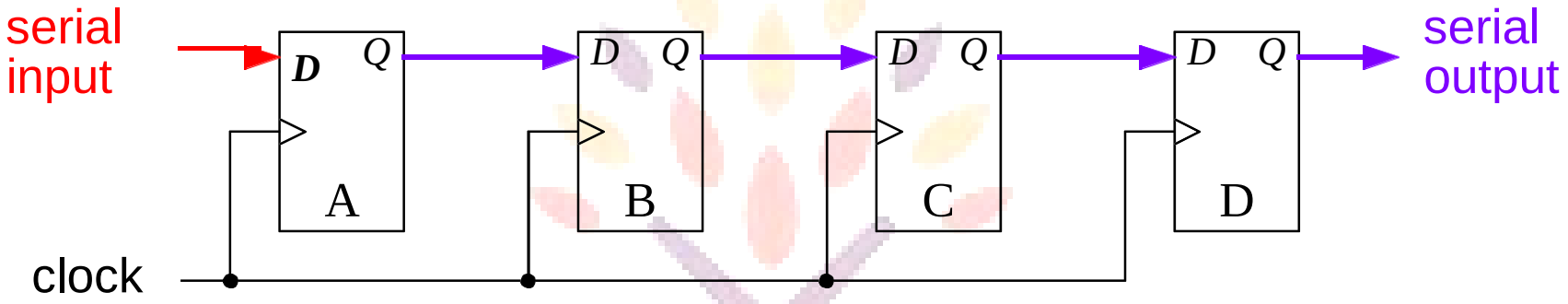
- Ring counters
- Johnson counters
- Pseudo-random binary sequences and encryption
- Ready-made shift registers are available as integrated circuits, such as the ’165

Conversion of data from serial to parallel and *vice versa*

- Large-scale devices such as ‘universal asynchronous receiver transmitters’ (UARTs) are based on shift registers
- Same functions available in microcontrollers (‘shift’ and ‘rotate’ instructions)

Basic shift register

A basic shift register is simply a chain of *D* flip-flops with a common clock.



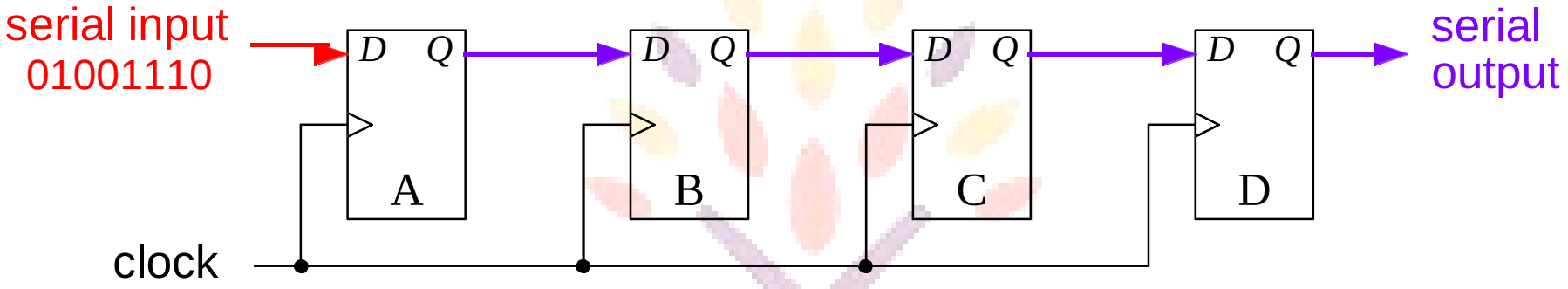
Each flip-flop transfers its *D* input to its *Q* output at a clock transition.

- The effect is to transfer data along the register, one flip-flop per clock cycle.

This type of register is called a serial input-serial output (SISO).

Basic shift register

A basic shift register is simply a chain of *D* flip-flops with a common clock.



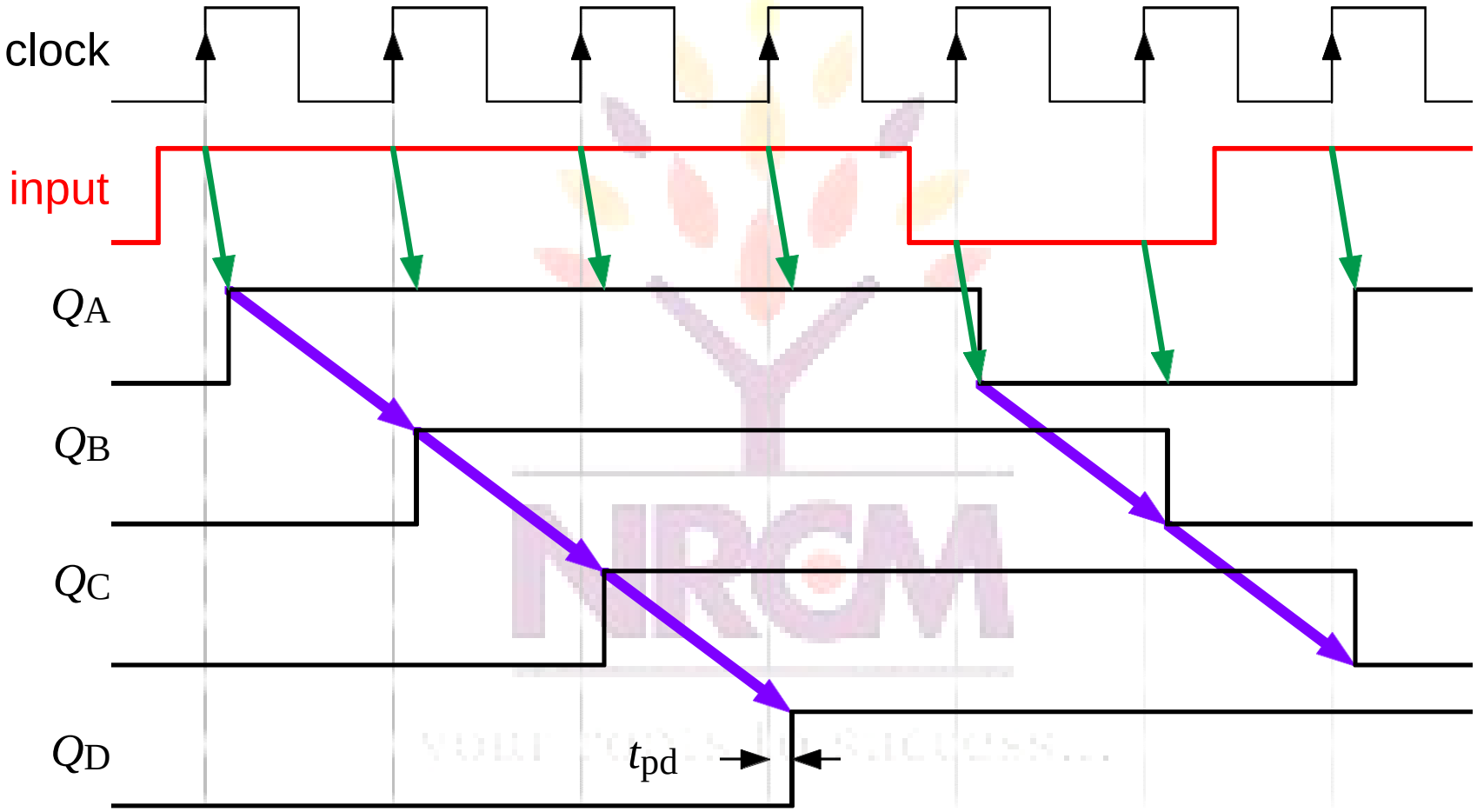
The table shows the contents of the register after successive clock transitions. The assumption is that the register is initially clear.

- The number of clock pulses needed to fill the register is equal to the number of flip-flops used to make the register.
  - This is a 4 bit register.

clock pulses

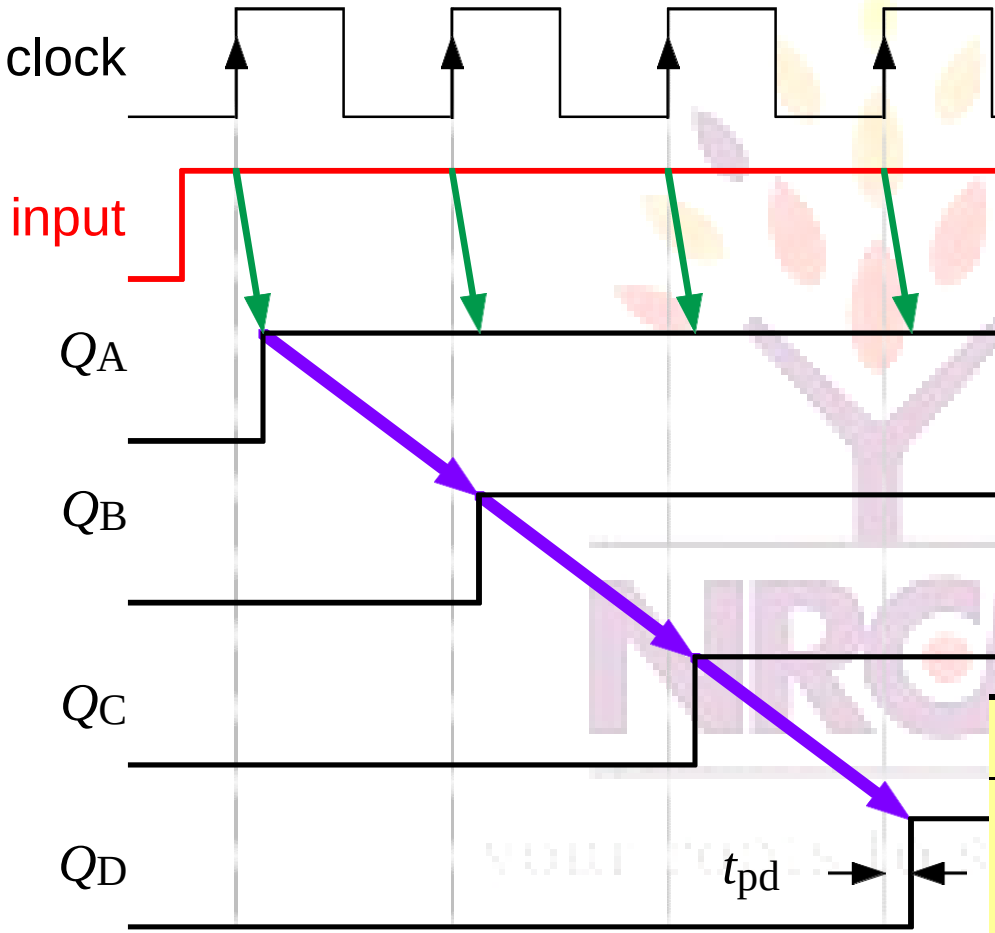
input	Q_A	Q_B	Q_C	Q_D
0	0	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
0	0	1	1	1
0	0	0	1	1
1	1	0	0	1
0	0	1	0	0

Timing for a shift register



The pattern in successive flip-flops moves to the right with each clock cycle to shift the pattern into and out of the register.

Timing for a shift register



input	Q_A	Q_B	Q_C	Q_D
0	0	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
0	0	1	1	1

Applications of a basic shift register

1. **Delay line** — *N* stages delay the signal by *N* clock cycles

2. **Multiplication and division by powers of 2**, because this just requires a shift of the binary number (like multiplication or division by 10 in decimal)

   Example: decimal 3 x 4 = 12 becomes 11 x 100 = 1100 in binary The arithmetic logic unit (ALU) of a computer processor uses a shift register for this purpose.

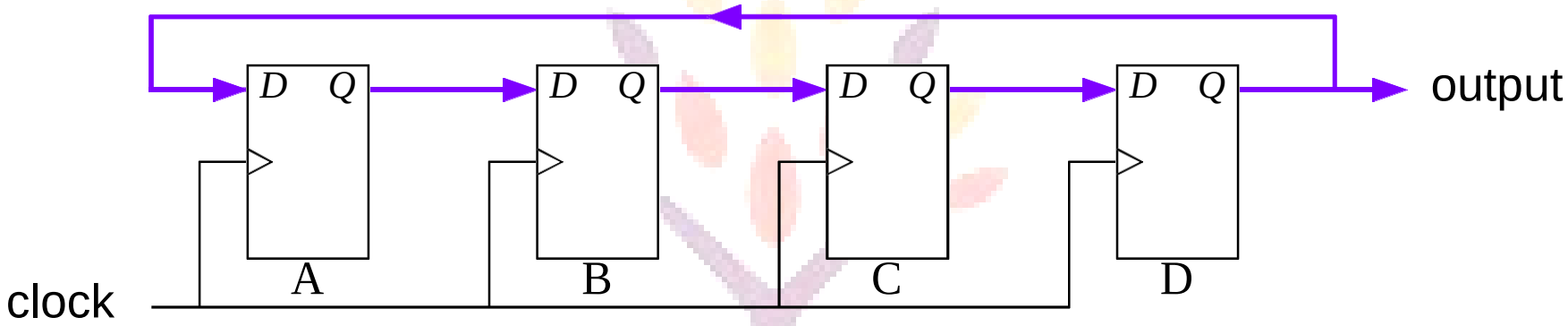
   Warning: the 'sense' of a shift — left or right — is usually based on its effect on binary numbers written in the usual way. For example, 11 → 1100 is called a **left shift**. This is clearer if both numbers are written with 8-bits as 00000011 → 00001100. Similarly, dividing by 2 such as 00010110 → 00001011 is a **right shift**.

   This is the opposite of what we usually draw in a counter circuit, with the least significant bit (LSB) on the left. **Take care!**

   There is a 'rotate' operation where the output from the shift register is fed back to the beginning, usually through the 'carry bit'.

Ring counter

A shift register with its output fed back to its input forms a **ring counter**.



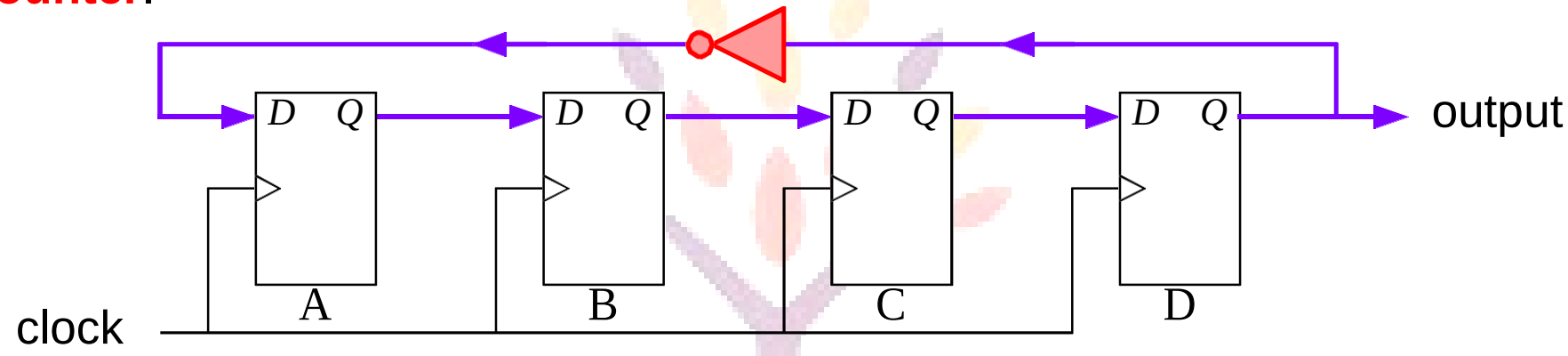
This can be used to generate an arbitrary binary pattern of length N, where N is the number of stages in the ring counter. It must be preloaded with the sequence desired, which then rotates around the counter indefinitely.

One application is to divide down the clock frequency for a slower part of a digital system, while keeping everything synchronous. Modern computers have several 'buses' running at different speeds, where a ring counter is used to create the clocks for the various buses.

It is much harder to multiply a given frequency to obtain a higher frequency signal. A **phase locked loop (PLL)** is often used.

Johnson counter

A ring counter with the **complement** of its output fed back is a **Johnson counter**.



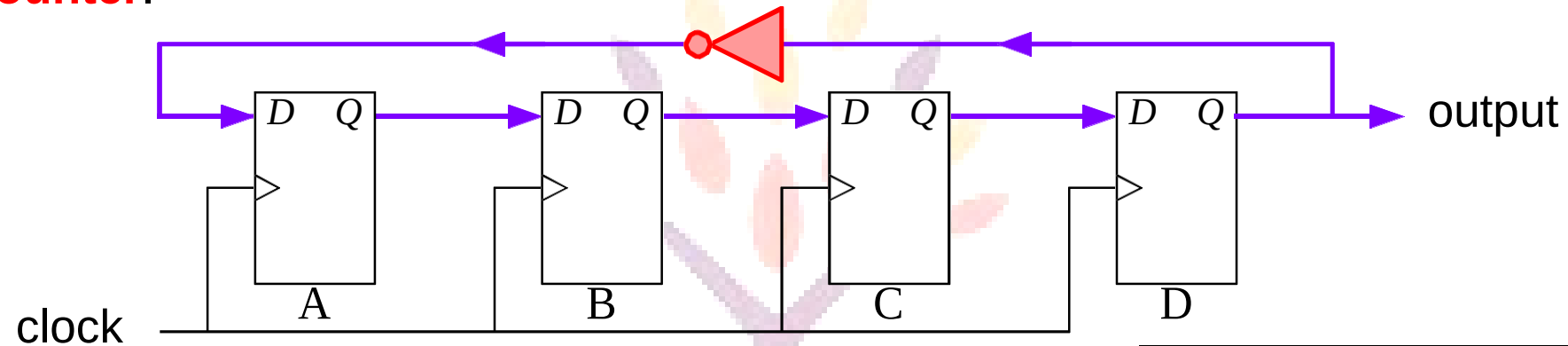
This generates longer sequences than a simple ring counter.

For example, a ring counter with 3 stages produces a cycle of 3 states — a waste as there are $2^3 = 8$ states in all.

A Johnson counter with 3 stages has a cycle of 6 and a separate cycle of 2. It is important to ensure that it follows the correct one!

Johnson counter

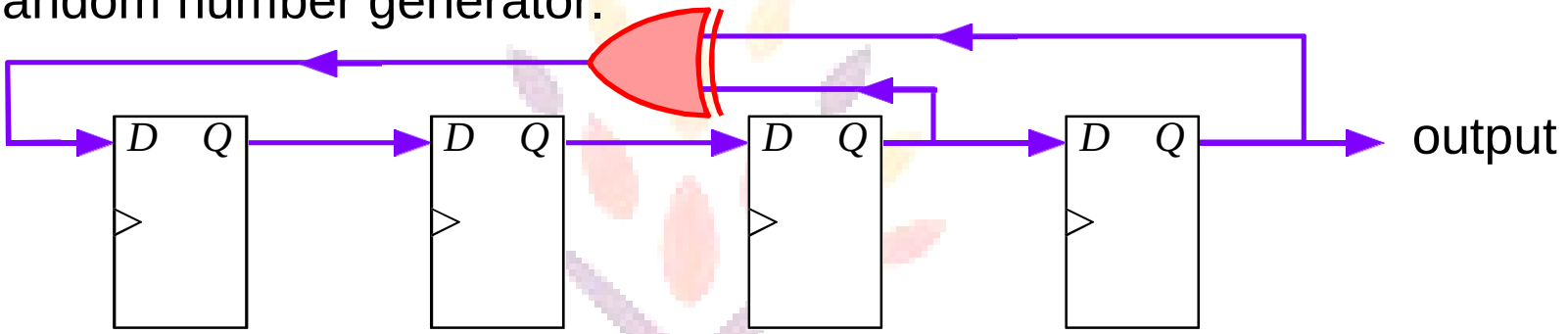
A ring counter with the **complement** of its output fed back is a **Johnson counter**.



Q_A	Q_B	Q_C
0	0	0
1	0	0
1	1	0
1	1	1
0	1	1
0	0	1
1	0	1
0	1	0

Pseudo-random number generator

A ring counter with feedback through an **exclusive-or gate** makes a simple pseudo-random number generator.



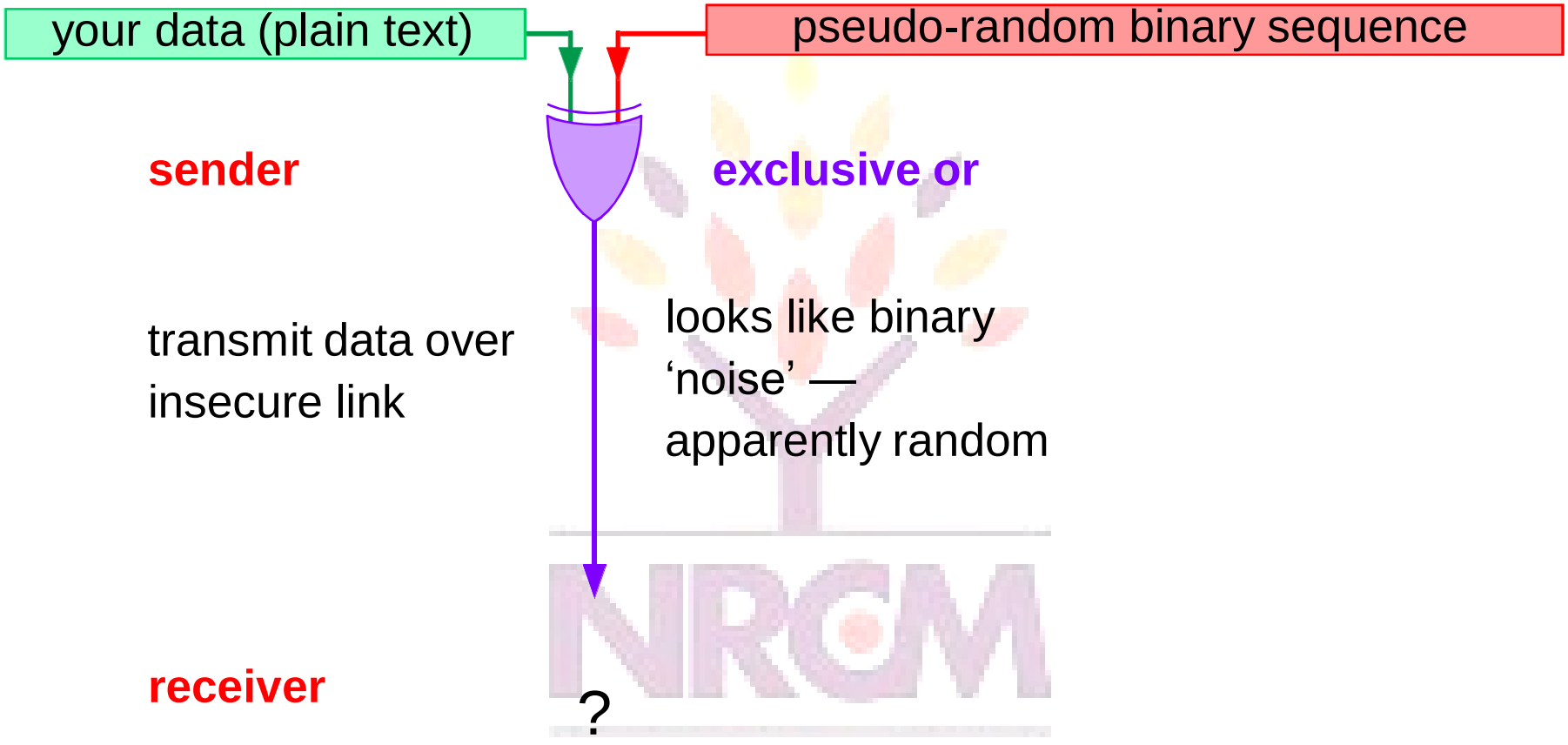
- Pseudo-random sequences of 1s and 0s have many applications, notably in encryption. They appear to be random over 'short' times but the sequence eventually repeats, hence the more accurate term 'pseudo-random'.

- Also, they can be reproduced perfectly if you know both:
  - **the method used to generate the sequence**
  - **the state in the sequence at which to start**

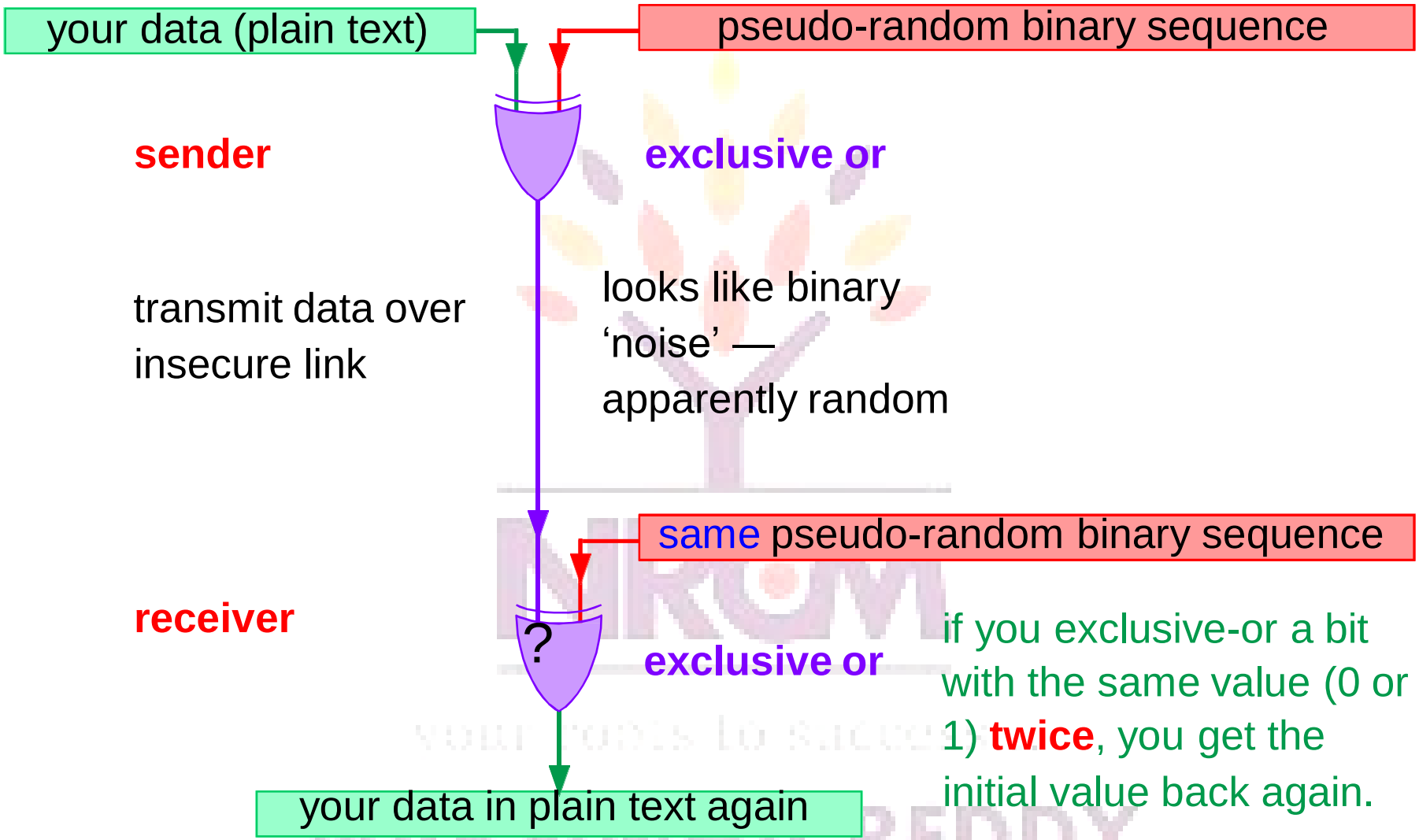
- This is an important feature! — see next sheet.

- The circuit above has a period of $2^4 - 1 = 15$
- (the missing state is 0000 —why?).

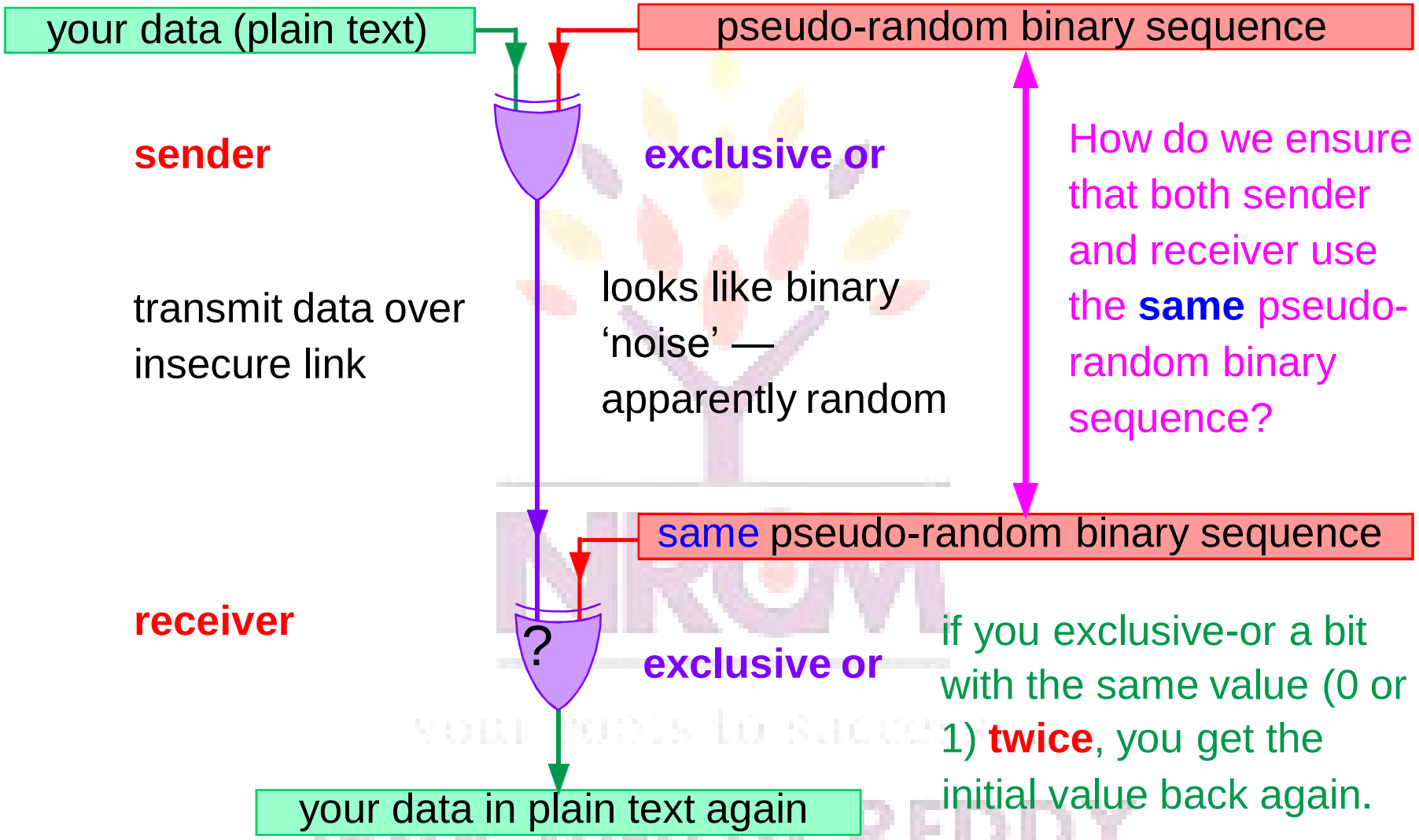
Pseudo-random binary sequences and encryption



Pseudo-random binary sequences and encryption



Pseudo-random binary sequences and encryption



This is the basis of the method used to encrypt data sent over the internet (https) or with a digital mobile phone.

Transmission of data — serial format

Data often has to be transmitted from one computer to another, or from a computer to peripheral equipment (printer, modem, …). This can be done in:

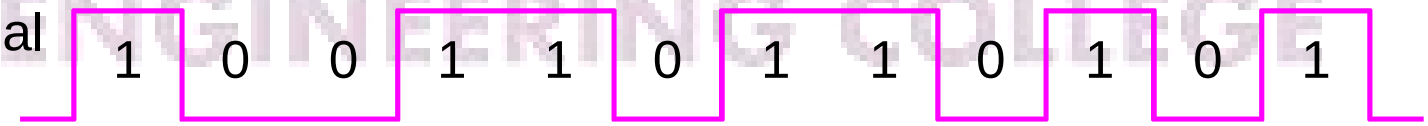
- **serial format**, one bit at a time
- **parallel format**, several bits at a time (e.g. byte at a time, 8 bits)

Serial format is most commonly used because it is simpler. Only a few wires are needed:

- traditional **serial 'COM' ports (RS-232)** need only 3 wires (transmitted data, received data and ground — but more may be used for control)
- **universal serial bus** (USB, common on modern computers) uses 4 wires (two for differential data plus power and ground)

Traditional serial transmission was slow but modern systems use much faster rates (USB version 1 up to 12 Mbits per second, FireWire 1 up to 400 Mbits per second), version 2 of both even faster.

simple serial bit stream



Parallel data

Where higher speed is required, several bits (usually a small number of bytes, each of 8 bits) may be moved at once. More complicated connections are needed — more wires. Common applications include:

- **inside the processor itself**, e.g. our microcontroller handles bytes
- inside a computer system on the **bus** (e.g. PCI) and interfaces to **disk drives** (e.g. e.g. SCSI or IDE)— but these are now mainly serial

Interfaces have changed to serial because it is hard to ensure that all bits on a parallel bus arrive at the same time at the high speed of modern systems.

Parallel data

Where higher speed is required, several bits (usually a small number of bytes, each of 8 bits) may be moved at once. More complicated connections are needed — more wires. Common applications include:

- **inside the processor itself**, e.g. our microcontroller handles bytes
- inside a computer system on the **bus** (e.g. PCI) and interfaces to **disk drives** (e.g. e.g. SCSI or IDE)— but these are now mainly serial

Interfaces have changed to serial because it is hard to ensure that all bits on a parallel bus arrive at the same time at the high speed of modern systems.

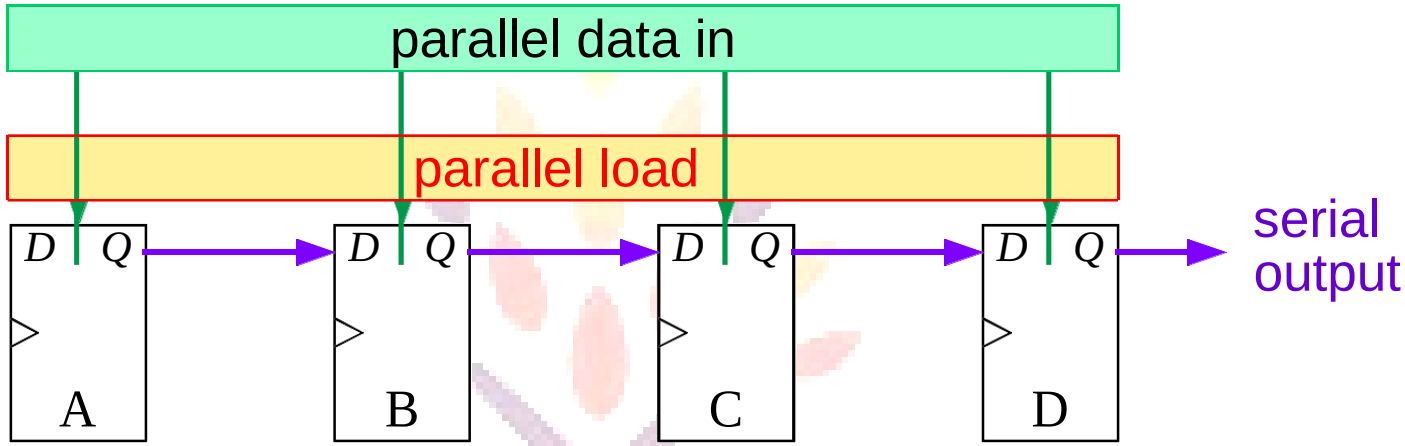
How do you interface a serial device to a computer?

How do we interface an external device that transmits serially with the bus of a computer that transfers one byte (8 bits) at a time?

- **Use a shift register.**

In practice this would almost certainly be buried inside a larger circuit called a **UART** (universal asynchronous receiver transmitter) or something similar.

Use of shift register to serialize data



Extra logic is added to the basic shift register so that all the flip-flops can be loaded in **parallel** (simultaneously), controlled by a shift/load input.

Once the data have been loaded, the clock is enabled and the values are shifted once per clock cycle. This causes the input data to be transferred to the output, one bit at a time — **serial output** (PISO).

The opposite process is used to read in serial data, fill up the shift register, and transfer it in parallel to a bus when the register is full (SIPO).

The register can also be parallel input – parallel output (PIPO).

Shift or rotate instructions can be used for the same process inside a microcontroller (if it doesn't have a UART built in, which many do).

ROM

- The data stored in ROM are always there, whether the power is on or not. A ROM can be removed from the PC, and then replaced, and the data it contains will still be there.
- Data stored in these chips is unchangeable, provides a measure of security against accidental or malicious changes to its contents. Unlike RAM, which can be changed as easily as it is read
  - We will look at five of them to see how they differ in the way they are programmed, erased, and reprogrammed

Mask ROM

- The mask ROM is usually referred to simply as a ROM.
- A regular ROM is constructed from hard-wired logic, encoded in the silicon itself to perform a specific function that cannot be changed.
- They consume very little power and reliable but cannot reprogram or rewrite.
- Several types of user programmable ROMs have been developed to overcome this disadvantage.

Programmable ROM (PROM)

- A mask ROM chip is very expensive and time-consuming to create in small quantities from scratch.
- Mainly, developers created a type of ROM known as programmable read-only memory (PROM).
- This is basically a blank ROM chip that can be written only once using special equipment called a PROM programmer.
- PROM chips have a grid of columns and rows just as ordinary ROMs do.

- The difference is that every intersection of a column and row in a PROM chip has a fuse connecting them.
- Since all the cells have a fuse, the initial (blank) state of a PROM chip is all 1s.
- The user cans selectively burn/blow any of these fuse links to produce the desired stored memory data.
- A charge sent through a column will pass through the fuse in a cell to a grounded row indicating a value of 1.

- To change the value of a cell to 0, you use a PROM programmer to send a specific amount of current to the cell to break the connection between the column and row by burning out the fuse.
- This process is known as burning the PROM.
- Very few bipolar PROMs are still available today.
- TMS27PC256 is a very popular CMOS PROM with a capacity of 32K $\times$ 8.

- An EPROM is a ROM that can be erased and reprogrammed as often as desired. Once programmed.
- The EPROM is a non-volatile memory that will hold its stored data indefinitely.
- A little glass window is provided in the top of the ROM package.
- Ultraviolet light of a specific frequency can be shined through this window for a specified period of time, which will erase all cells at the same time so that an erased EPROM stores all 1s and allow it to be reprogrammed again.

- EPROMs are configured using an EPROM programmer that provides voltage at specified levels depending on the type of EPROM used.

- Obviously this is much more useful than a regular PROM, but it does require the erasing light.

- EPROMs are available in a wide range of capacities and access times. The 27C64 is an example of 8K x 8 CMOS EPROM

Ele **VI)**

- They require dedicated equipment and a labor-intensive process to remove and reinstall them each time a change is necessary.
- The next type of ROM is the EEPROM, which can be erased under software control.
- This is the most flexible type of ROM, and is now commonly used for holding BIOS programs

Ele **VI)**

- In EEPROMs the chip does not have to be removed to be rewritten, the entire chip need not be fully erased to change a specific portion of it, and changing the contents does not require additional dedicated equipment.

- Instead of using UV light, you can return the electrons in the cells of an EEPROM to normal with the localized application of an electric field to each cell.

Ele **VI)**

- This erases the targeted cells of the EEPROM, which can then be rewritten.
- EEPROMs are changed 1 byte at a time, which makes them versatile but slow.
- The Intel 2864 is an example of EEPROM with 8K × 8 array with 13 address inputs and eight data I/O pins

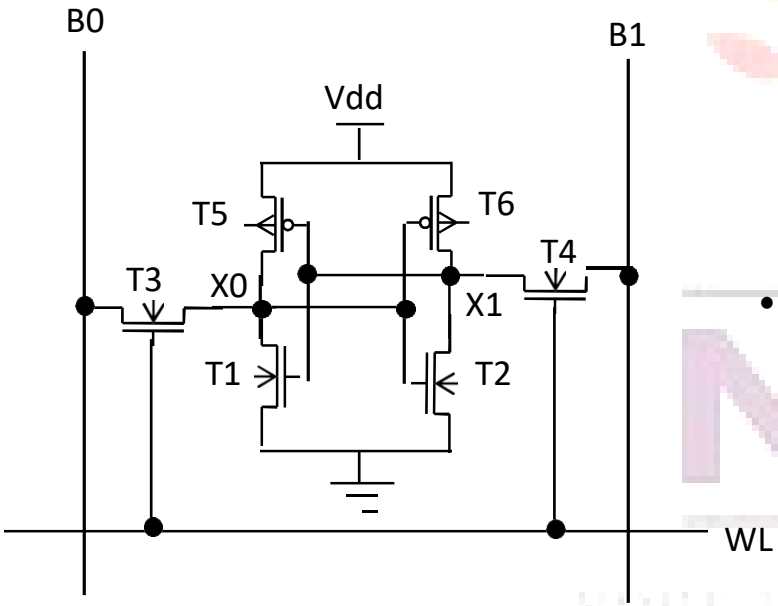
Flash Memory

- Flash memories are so called because of their rapid erase and write times.
- EEPROM chips speed is too slow to use in many products that required quick changes to the data stored on the chip.
- So a new type of EEPROM called Flash memory that uses in-circuit wiring to erase by applying an electrical field to the entire chip or to predetermined sections of the chip called blocks.

Flash Memory

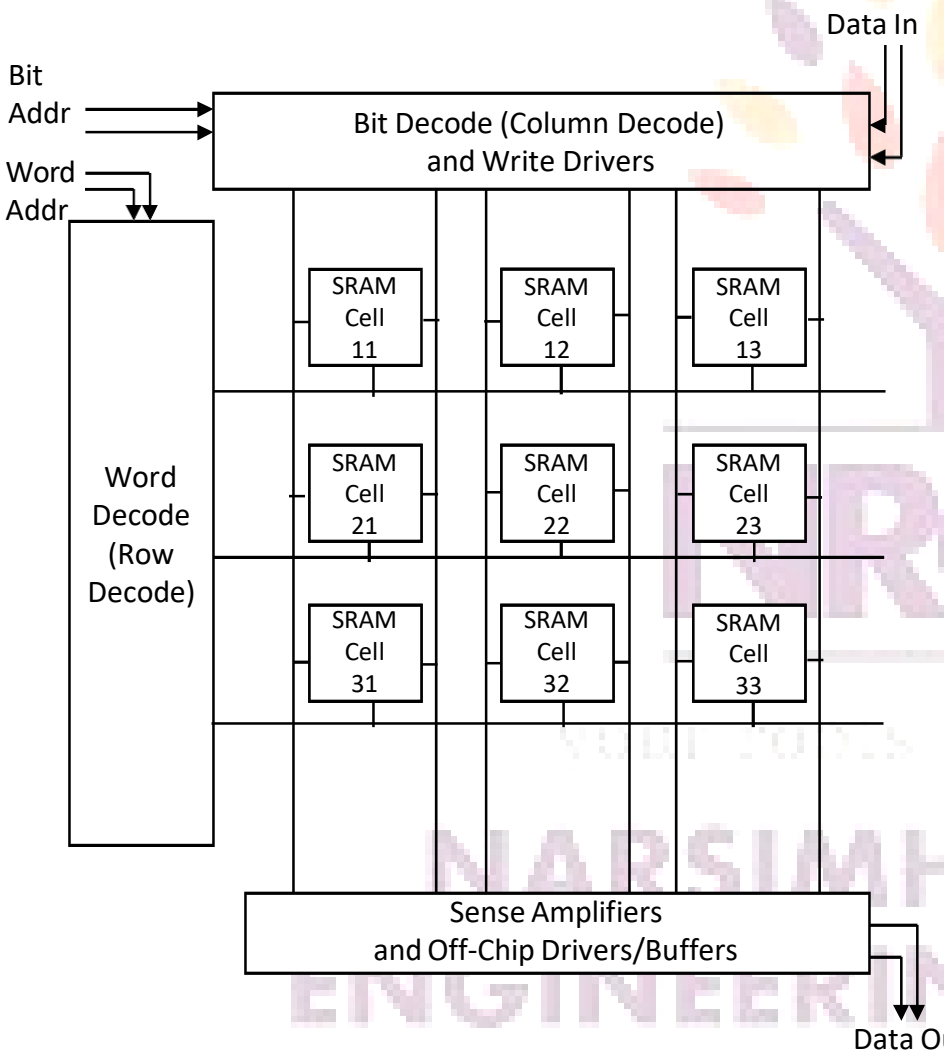
- Flash memory works much faster than traditional EEPROMs because it writes data in chunks, usually 512 bytes in size, instead of 1 byte at a time.
- The 28F256A CMOS IC is an example of flash memory chip, which has a capacity of 32K $\times$ 8.

The SRAM Memory Cell



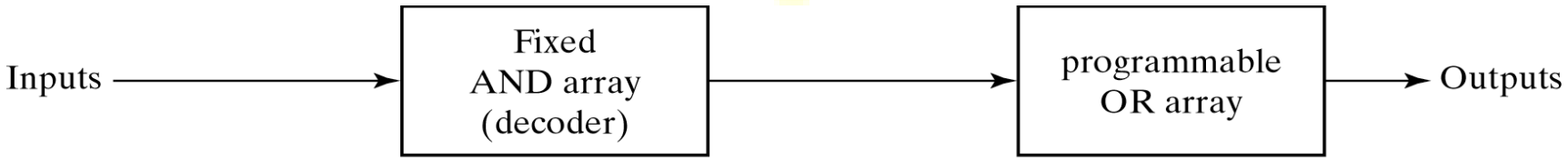
- Circuit Schematic:
  - 4 NFETs and 2 PFETs: T1 & T2 called active devices; T3 & T4 called the I/O devices; T5 & T6 sometimes called loads.
  - The cell is comprised of two cross-coupled inverters (positive feedback).
  - 2 vertical lines (bit lines B0 & B1) are used for sensing state of cell and writing data in the cell
  - 1 horizontal line (word line WL) is used to select a row of cells for writing or reading and to prevent the unselected rows of cells from being disturbed.
- Circuit Operation:
  - The cell has two stable states: "0" and "1"
    - "0" State = Node X0 high and Node X1 low; T2 & T5 are ON, T1 & T6 are OFF.
    - "1" State = Node X1 high and Node X0 low; T1 & T6 are ON; T2 & T5 are OFF.
    - No dc current flows in either state.
  - Write: raise WL to Vdd; pull one bit line high & pull the other bit line low
  - Read: raise WL to Vdd; precharge bit lines to ½ Vdd

SRAM Memory Array Organization

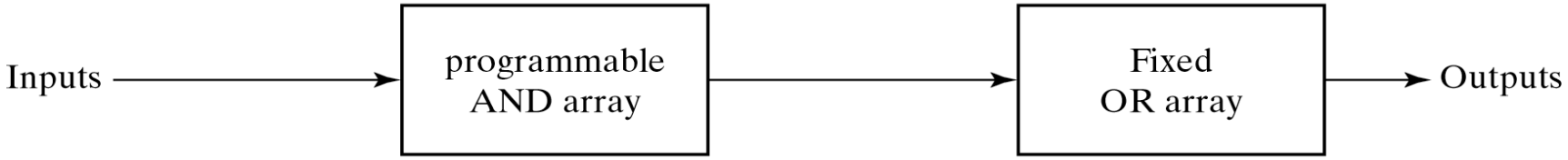


- READ Operation:
  - Word Decode circuitry selects one of n word lines and drives high to Vdd (say WL2); other word lines held at gnd.
  - Bit Lines all precharged to half Vdd
  - Selected cell's I/O devices turned ON and apply a ΔV to bit line pair
  - Sense amp triggers on bit line ΔV and stores read data "0" or "1"
- WRITE Operation:
  - Selected WL is driven high to Vdd by word decode circuitry turning ON I/O devices in selected cells
  - Selected bit column has one BL pulled high to Vdd and the other pulled low to gnd, thus writing the selected cell.
  - Unselected bit columns merely perform a READ operation.

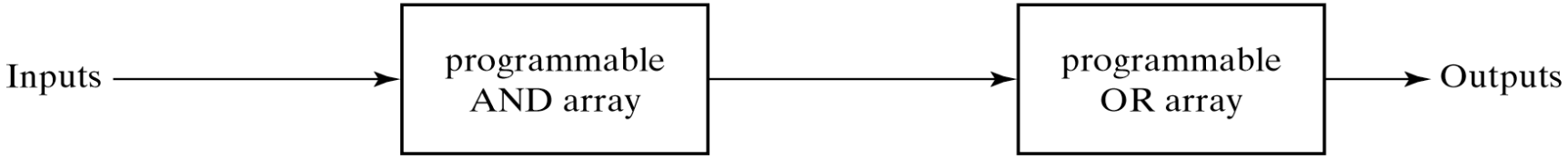
- A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.
- PROM: fixed AND array constructed as a decoder and programmable OR array.
- PAL: programmable AND array and fixed OR array.
- PLA: both the AND and OR arrays can be programmed.



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL)



(c) Programmable logic array (PLA)

Fig. 7-13 Basic Configuration of Three PLDs

Programmable Logic Array

- Fig.7-14, the decoder in PROM is replaced by an array of AND gates that can be programmed to generate any product term of the input variables.
- The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.
- The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$). The output doesn't change and connect to 0 (since $x \oplus 0 = x$).

PLA

F1 = AB’+AC+A’BC’

F2 = (AC+BC)’

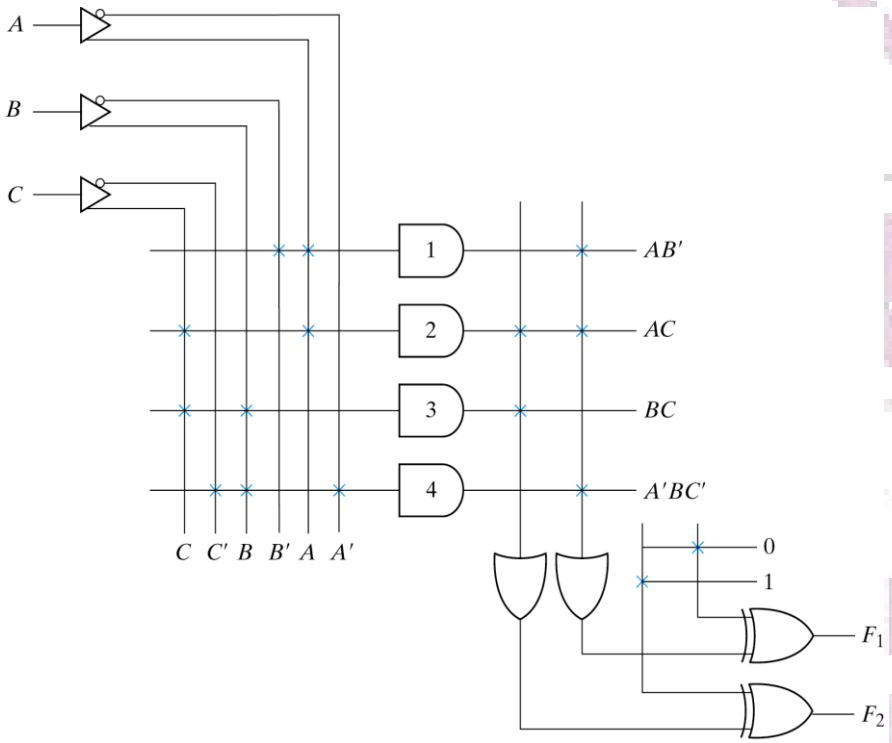


Fig. 7-14 PLA with 3 Inputs, 4 Product Terms, and 2 Outputs

Table 7-5
PLA Programming Table

		Inputs			Outputs (T)	(C)
Product Term		A	B	C	F_1	F_2
AB′	1	1	0	–	1	–
AC	2	1	–	1	1	1
BC	3	–	1	1	–	1
A′BC′	4	0	1	0	1	–

1. First: lists the product terms numerically
2. Second: specifies the required paths between inputs and AND gates
3. Third: specifies the paths between the AND and OR gates
4. For each output variable, we may have a T(ture) or C(complement) for programming the XOR gate

- Careful investigation must be undertaken in order to reduce the number of distinct product terms, PLA has a finite number of AND gates.
- Both the true and complement of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

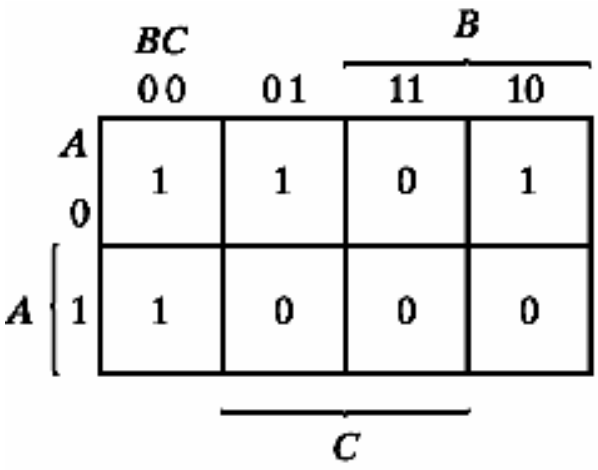
Example

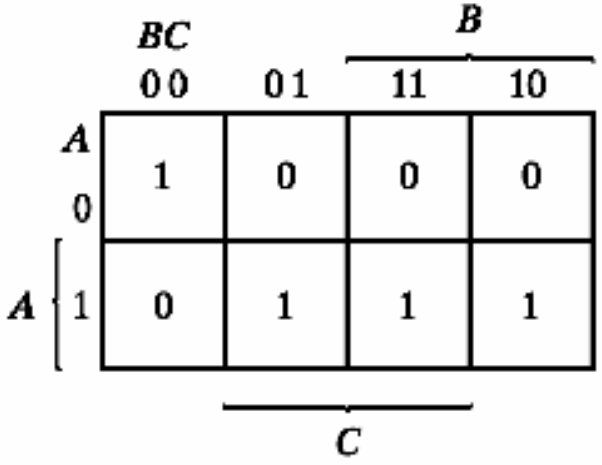
Implement the following two Boolean functions with a PLA:

$F_1(A, B, C) = \sum(0, 1, 2, 4)$

$F_2(A, B, C) = \sum(0, 5, 6, 7)$

The two functions are simplified in the maps of Fig.7-15





1 elements — $F_1 = A'B' + A'C' + B'C'$

0 elements — $F_1 = (AB + AC + BC)'$

$F_2 = AB + AC + A'B'C'$

$F_2 = (A'C + A'B + AB'C')'$

PLA table by simplifying the function

- Both the true and complement of the functions are simplified in sum of products.
- We can find the same terms from the group terms of the functions of F_1, F_1',F_2 and F_2' which will make the minimum terms.

F1 = (AB + AC + BC)'

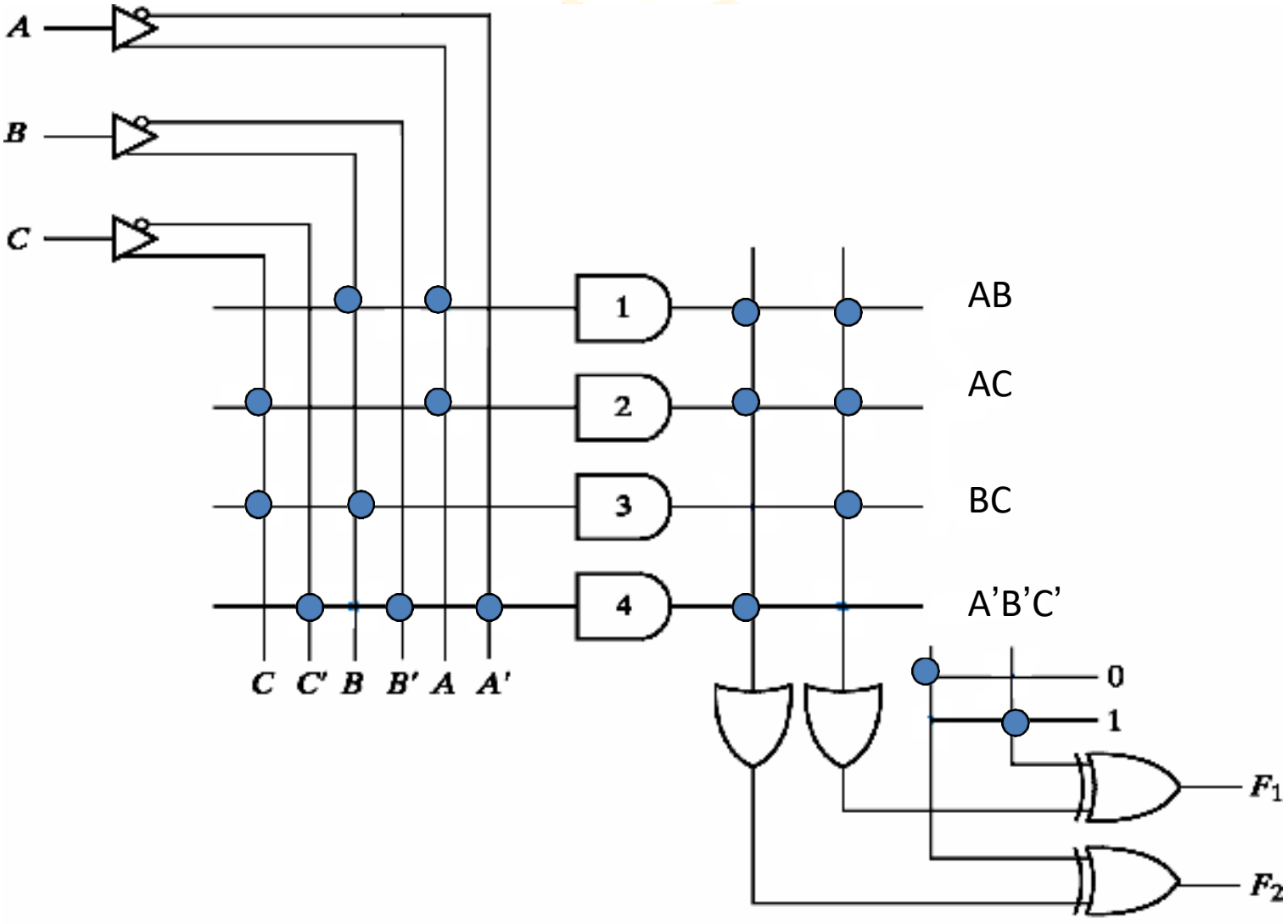
F2 = AB + AC + A'B'C'

PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$A'B'C'$	4	0	0	0	–	1

Fig. 7-15 Solution to Example 7-2

PLA implementation



Programmable Array Logic

- The PAL is a programmable logic device with a fixed OR array and a programmable AND array.

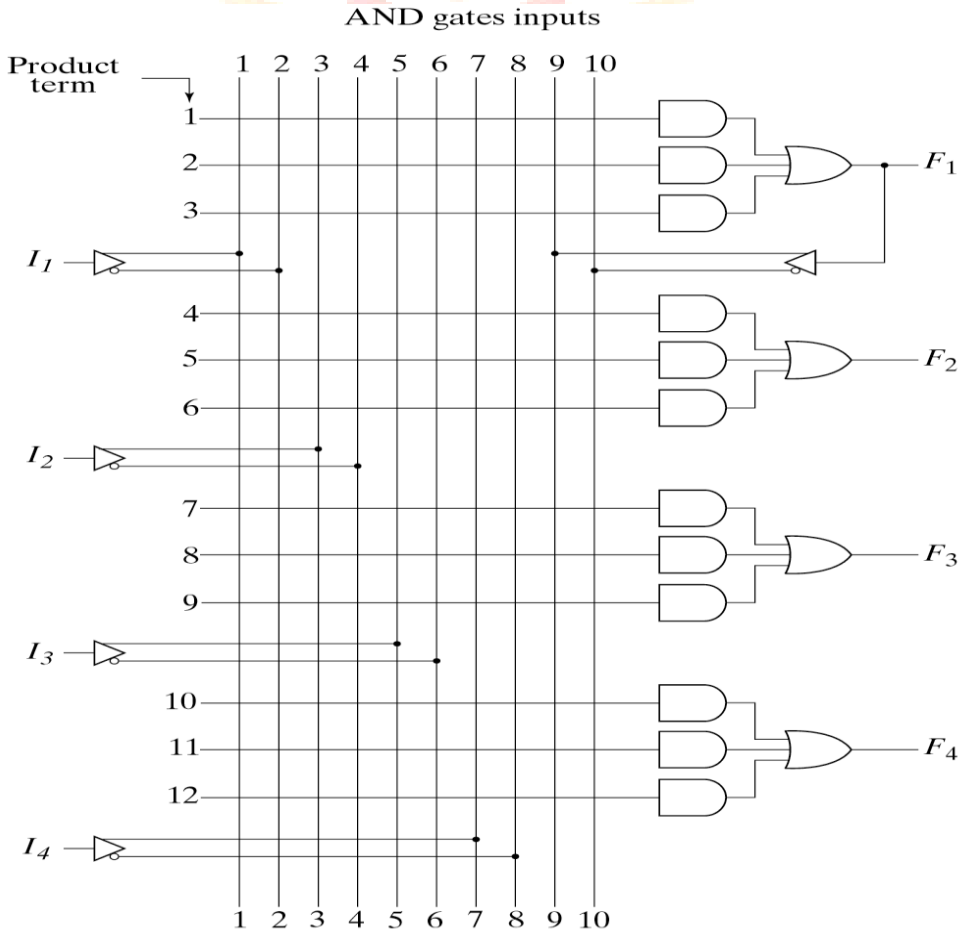


Fig. 7-16 PAL with Four Inputs, Four Outputs, and Three-Wide AND-OR Structure

PAL

- When designing with a PAL, the Boolean functions must be simplified to fit into each section.
- Unlike the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself without regard to common product terms.
- The output terminals are sometimes driven by three-state buffers or inverters.

Example

w(A, B, C, D) = ∑(2, 12, 13)
x(A, B, C, D) = ∑(7, 8, 9, 10, 11, 12, 13, 14, 15)
y(A, B, C, D) = ∑(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)
z(A, B, C, D) = ∑(1, 2, 8, 12, 13)

Simplifying the four functions as following Boolean functions:

w = ABC’ + A’B’CD’
x = A + BCD
w = A’B + CD + B’D’
w = ABC’ + A’B’CD’ + AC’D’ + A’B’C’D = w + AC’D’ + A’B’C’D

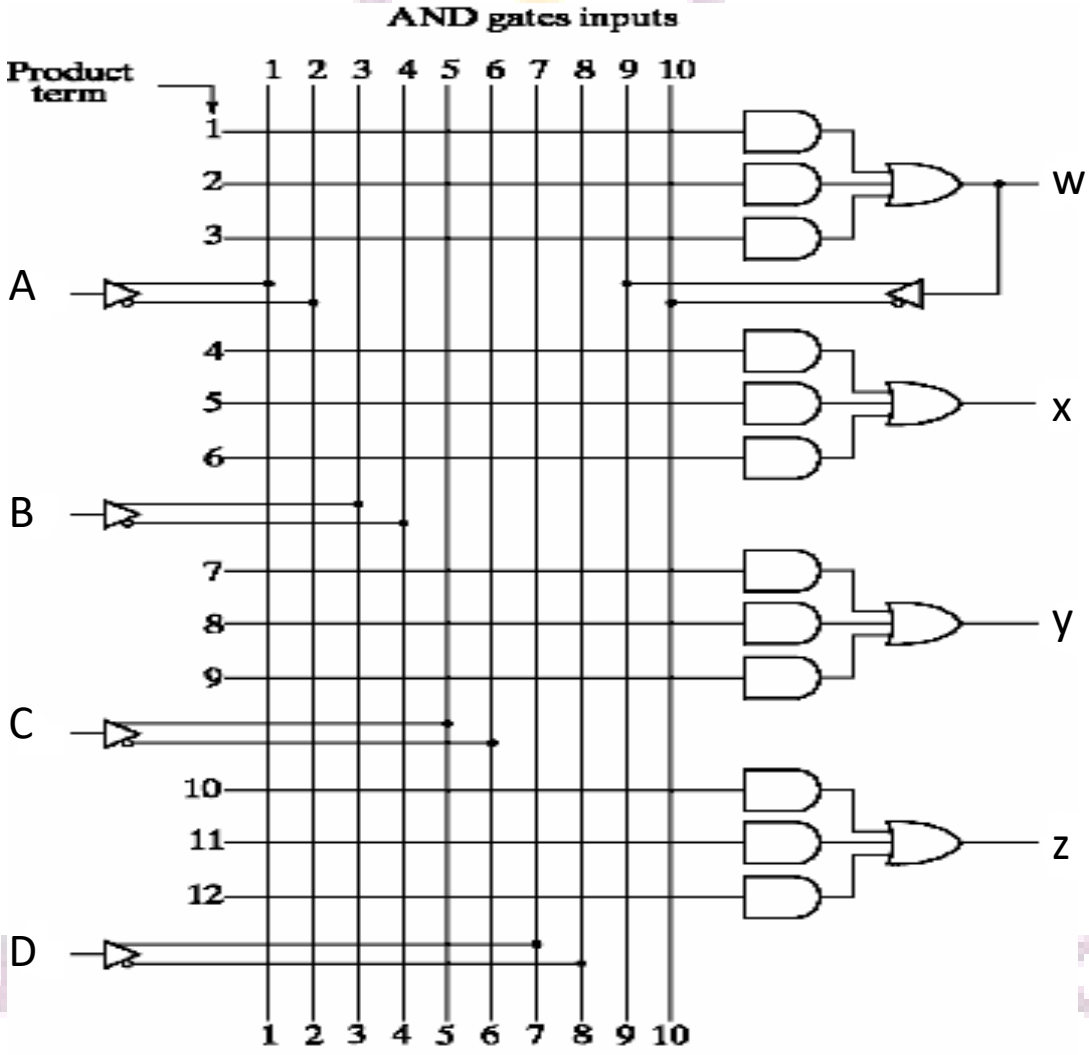
PAL Table

- z has four product terms, and we can replace by w with two product terms, this will reduce the number of terms for z from four to three.

Table 7-6
PAL Programming Table

	AND Inputs					
Product Term	*A*	*B*	*C*	*D*	*W*	**Outputs**
1	1	1	0	–	–	$w = ABC'$
2	0	0	1	0	–	$+ A'B'CD'$
3	–	–	–	–	–	
4	1	–	–	–	–	$x = A$
5	–	1	1	1	–	$+ BCD$
6	–	–	–	–	–	
7	0	1	–	–	–	$y = A'B$
8	–	–	1	1	–	$+ CD$
9	–	0	–	0	–	$+ B'D'$
10	–	–	–	–	1	$z = w$
11	1	–	0	0	–	$+ AC'D'$
12	0	0	0	1	–	$+ A'B'C'D$

PAL implementation



Fuse map for example

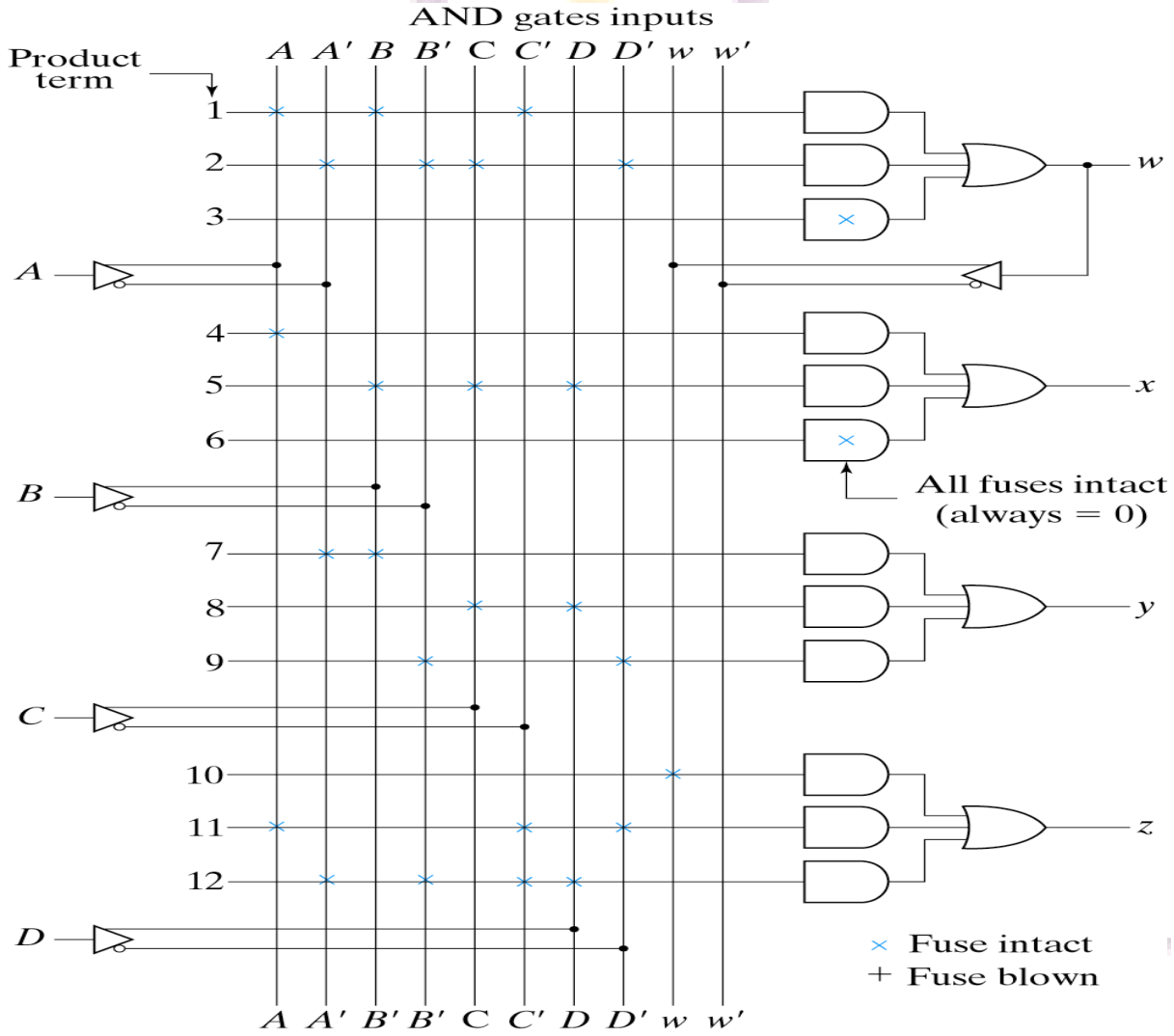


Fig. 7-17 Fuse Map for PAL as Specified in Table 7-6

S

- Sequential programmable devices include both gates and flip-flops.
- There are several types of sequential programmable devices, but the internal logic of these devices is too complex to be shown here.
- We will describe three major types without going into their detailed construction.

1. Sequential (or simple) Programmable Logic Device (SPLD)
2. Complex Programmable Logic Device (CPLD)
3. Field Programmable Gate Array (FPGA)

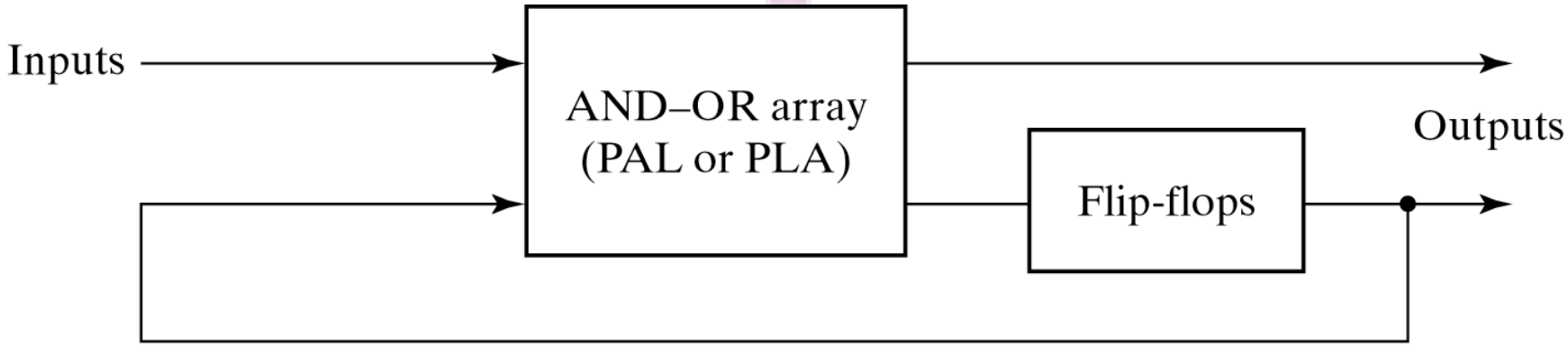


Fig. 7-18 Sequential Programmable Logic Device

- The first programmable device developed to support sequential circuit implementation is the field-programmable logic sequencer(FPLS).
- A typical FPLS is organized around a PLA with several outputs driving flip-flops.
- The flip-flops are flexible in that they can be programmed to operate as either JK or D type.
- The FPLS did not succeed commercially because it has too many programmable connections.

- Each section of an SPLD is called a macrocell.
- A macrocell is a circuit that contains a sum-of-products combinational logic function and an optional flip-flop.
- We will assume an AND-OR sum of products but in practice, it can be any one of the two-level implementation presented in Sec.3-7.

Macrocell

- Fig.7-19 shows the logic of a basic macrocell.
- The AND-OR array is the same as in the combinational PAL shown in Fig.7-16.

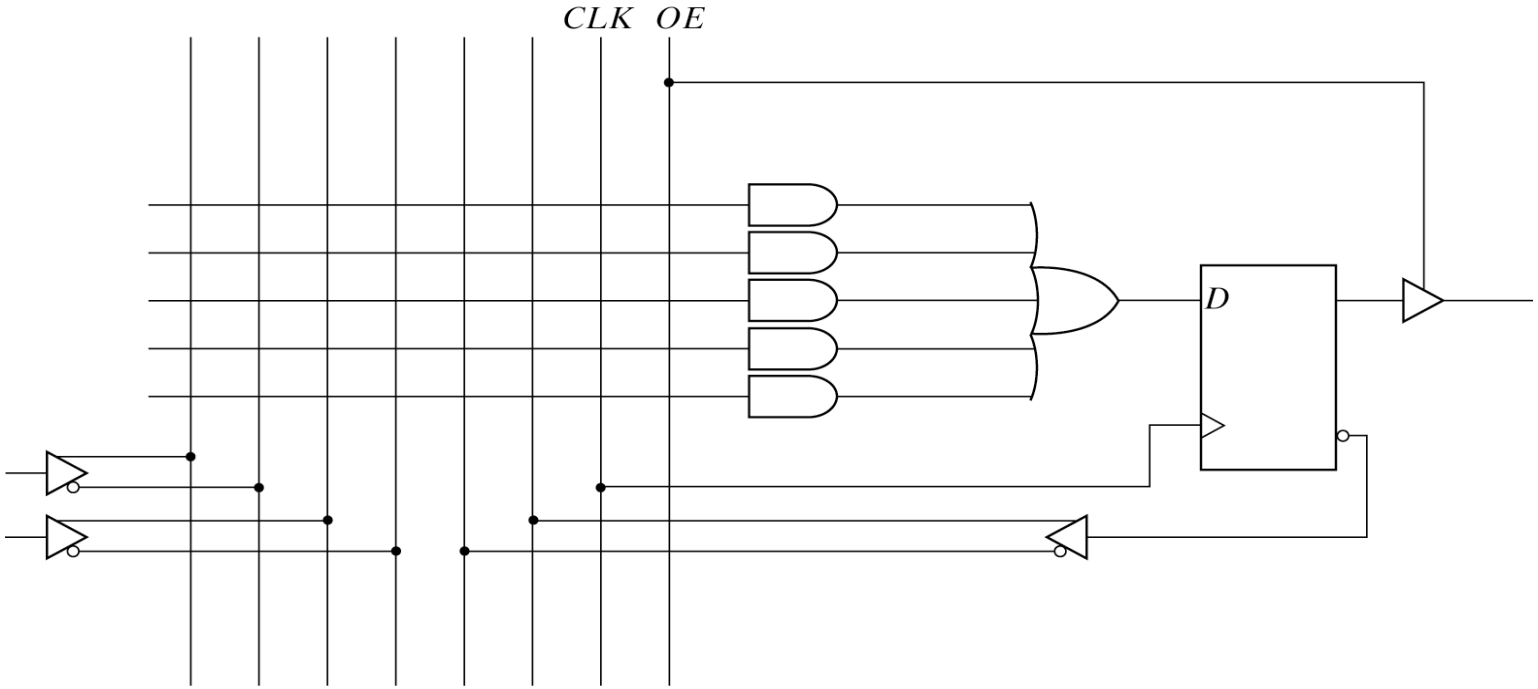


Fig. 7-19 Basic Macrocell Logic

CPLD

- A typical SPLD has from 8 to 10 macrocells within one IC package. All the flip-flops are connected to the common CLK input and all three-state buffers are controlled by the EO input.
- The design of a digital system using PLD often requires the connection of several devices to produce the complete specification. For this type of application, it is more economical to use a complex programmable logic device (CPLD).
- A CPLD is a collection of individual PLDs on a single integrated circuit.

CPLD

- Fig.7-20 shows a general configuration of a CPLD. It consists of multiple PLDs interconnected through a programmable switch matrix. 8 to 16 macrocell per PLD.

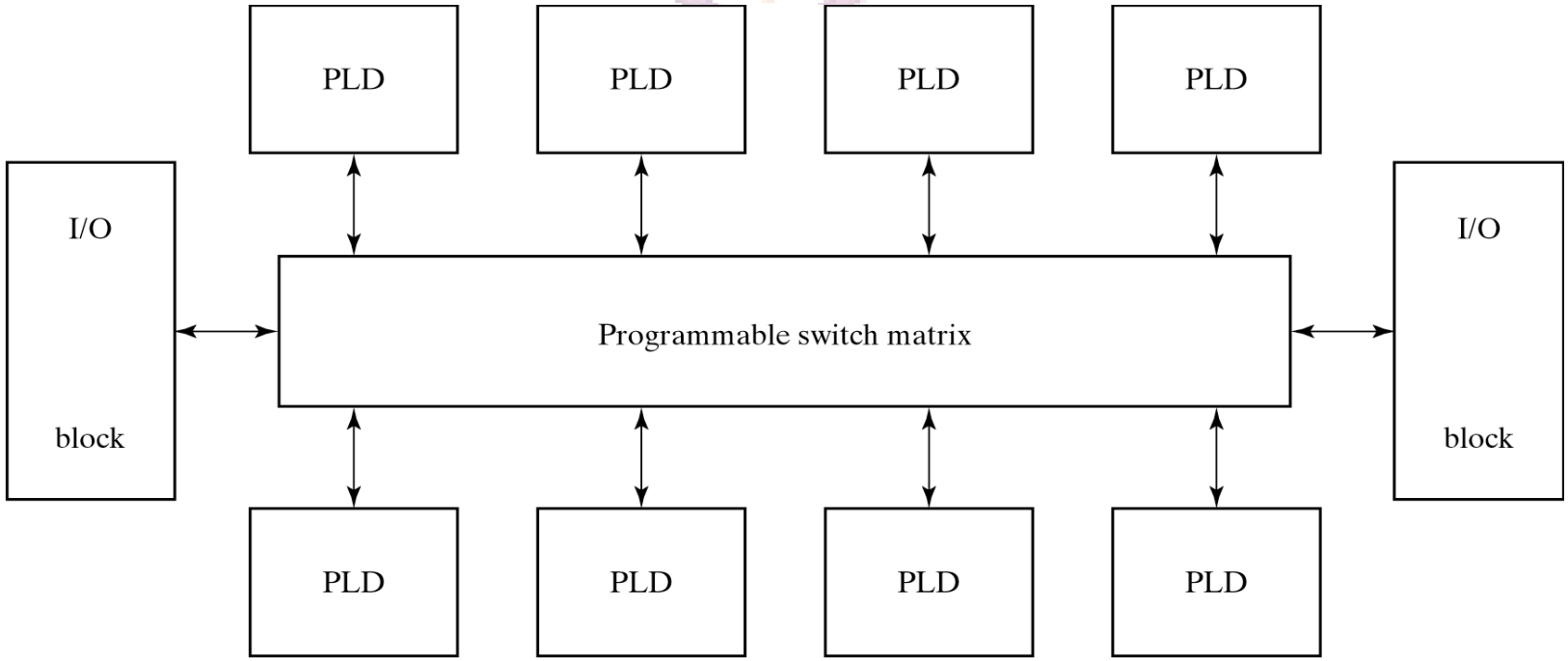


Fig. 7-20 General CPLD Configuration

Gate Array

- The basic component used in VLSI design is the gate array.
- A gate array consists of a pattern of gates fabricated in an area of silicon that is repeated thousands of times until the entire chip is covered with the gates.
- Arrays of one thousand to hundred thousand gates are fabricated within a single IC chip depending on the technology used.

FPGA

- FPGA is a VLSI circuit that can be programmed in the user's location.
- A typical FPGA logic block consists of look-up tables, multiplexers, gates, and flip-flops.
- Look-up table is a truth table stored in a SRAM and provides the combinational circuit functions for the logic block.

Differential of RAM and ROM in FPGA

- The advantage of using RAM instead of ROM to store the truth table is that the table can be programmed by writing into memory.
- The disadvantage is that the memory is volatile and presents the need for the look-up table content to be reloaded in the event that power is disrupted.

Algorithmic State Machines

Introduction

Digital system is specified by the following three components:

- The set of registers in the system
- The operations that are performed on the data stored in the registers.
- The control that supervises the sequences of operations in the system.

Control and Datapath Interaction

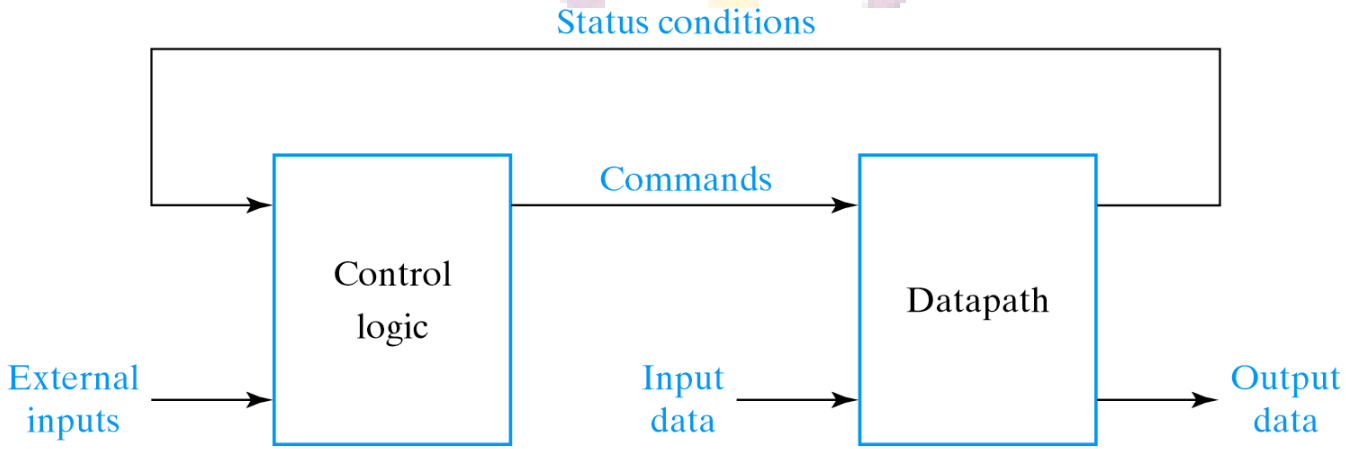


Fig. 8-2 Control and Datapath Interaction

Datapath

- Binary information in digital systems classified as either data or control.
- Data – bits of information manipulated by performing arithmetic and logic operations.
- Hardware components realizing above operations are adders, decoders, multiplexers, counters e.t.c

Control Path

- Command signals used to supervise execution of algorithms by datapath.
- Bi-directional communication with datapath through status conditions used to determine the sequence of control signals.
- Control logic inherently sequential.
- Control logic is usually implemented using FSMs

Algorithm Implementation

- Often we have to implement an algorithm in hardware instead of software
- Algorithm is a well defined procedure consisting of a finite number of steps to the solution of a problem.
- It is often hard to translate the algorithm into an FSM.
- ASMs can serve as stand-alone sequential network model.

Algorithmic State Machine

•Used to graphically describe the operations of an FSM more concisely

•Resembles conventional flowcharts – differs in interpretation.

•Conventional flowchart – sequential way of representing procedural steps and decision paths for algorithm

-No time relations incorporated

•ASM chart – representation of sequence of events together with timing relations between states of sequential controller and events occurring while moving between steps

ASM Chart

•Three basic elements: state box, decision box and conditional box

-State and decision boxes used in conventional flowcharts

-Conditional box characteristic to ASM

•State box

-Used to indicate states in control sequence

•Register operations and output signals used to control generation of next state written

State box

- Represents one state in the ASM.
- May have an optional state output list.
- Single entry.
- Single exit to state or decision boxes.

State Box

State name T3

•Binary code of T3 – 011

•Register operation R <- 0

•START – name of

outputs signal generated

in this stage

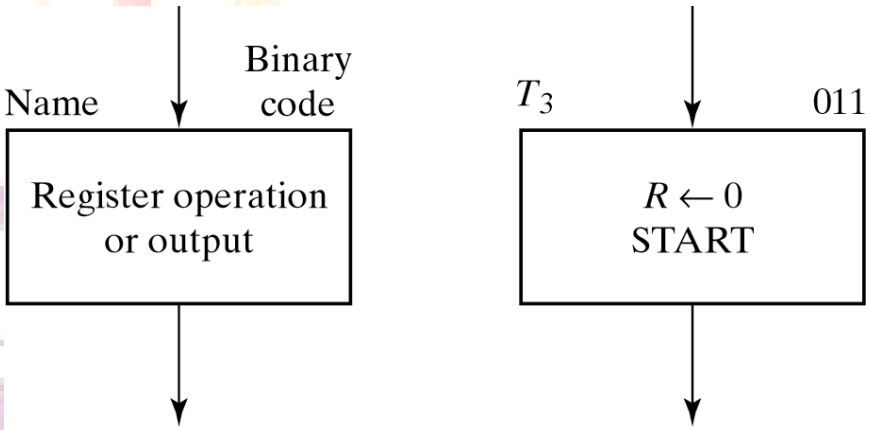


Fig. 8-3 State Box

Decision box

- Provides for next alternatives and conditional outputs.
- Conditional output based on logic value of Boolean expression involving external input variables and status information.
- Single entry.
- Dual exit, denoting if Boolean expression is true or false.
- Exits to decision, state or conditional boxes.

Decision Box

•Input condition subject to test inside diamond shape box

•Two or more outputs represent exit paths dependant on value of condition in decision box

•Two paths for binary based conditions

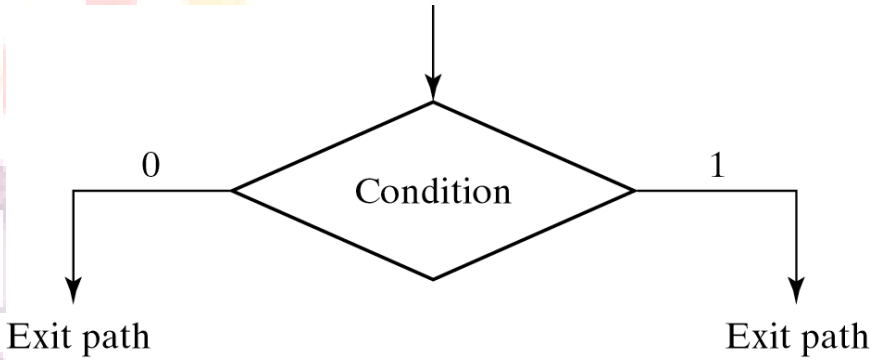


Fig. 8-4 Decision Box

Conditional output box

- **Provides a listing of output variables that are to have a value logic-1, i.e., those output variables being asserted.**
- **Single entry from decision box.**
- **Single exit to decision or state box.**

Conditional Box

- In state T1
  Output signal START
  generated
  Status of input E
  checked
- If E = 1, R <- 0,
otherwise remains
unchanged
- Conditional
operation executed
depending on result
of coming from
decision box

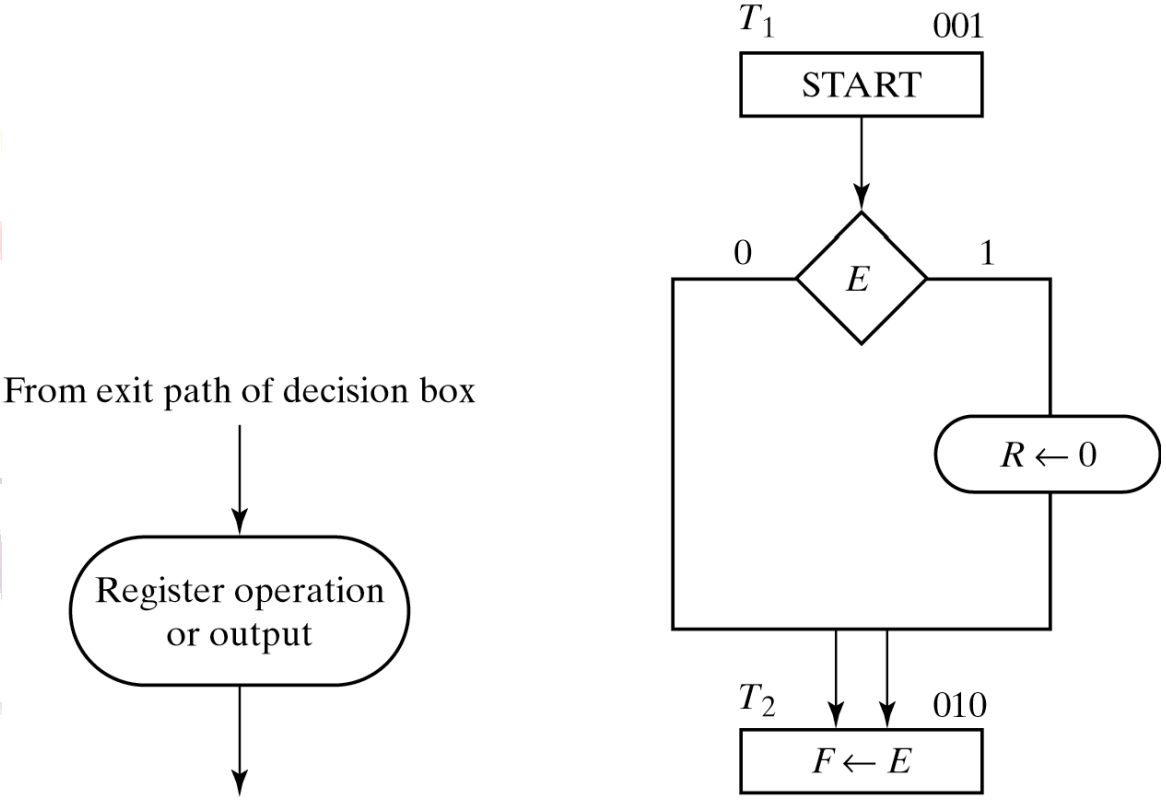


Fig. 8-5 Conditional Box

ASM Block

- Consists of the interconnection of a single state box along with one or more decision and/or conditional boxes.
- It has one entry path which leads directly to its state box, and one or more exit paths.
- Each exit path must lead directly to a state, including the state box in itself.
- A path through an ASM block from its state box to an exit path is called a link path.

Timing Considerations

All sequential elements in datapath and control path controlled by master-clock generator.

Does not necessarily imply single clock in design.

•Multiple clocks can be obtained through division of clock signals from master-clock generator.

•Not only internal signals, but also inputs synchronized with clock.

•Normally, inputs supplied by other devices working with the same master clock.

•Some inputs can arrive asynchronously

Difficult to handle by synchronous designs, require asynchronous glue-logic.

ASM Block

•In conventional flowchart, evaluation of each chart element takes one clock cycle

Step 1: Reg A incremented

Step 2: Condition E evaluated

Step 3: Based on evaluation results, state

T2, T3 or T4 entered

•In ASM the entire block considered as one unit

•All operations within block occurring during single edge transition

The next state evaluated during the same clock

System enters next state T2, T3 or T4 during transition of next clock

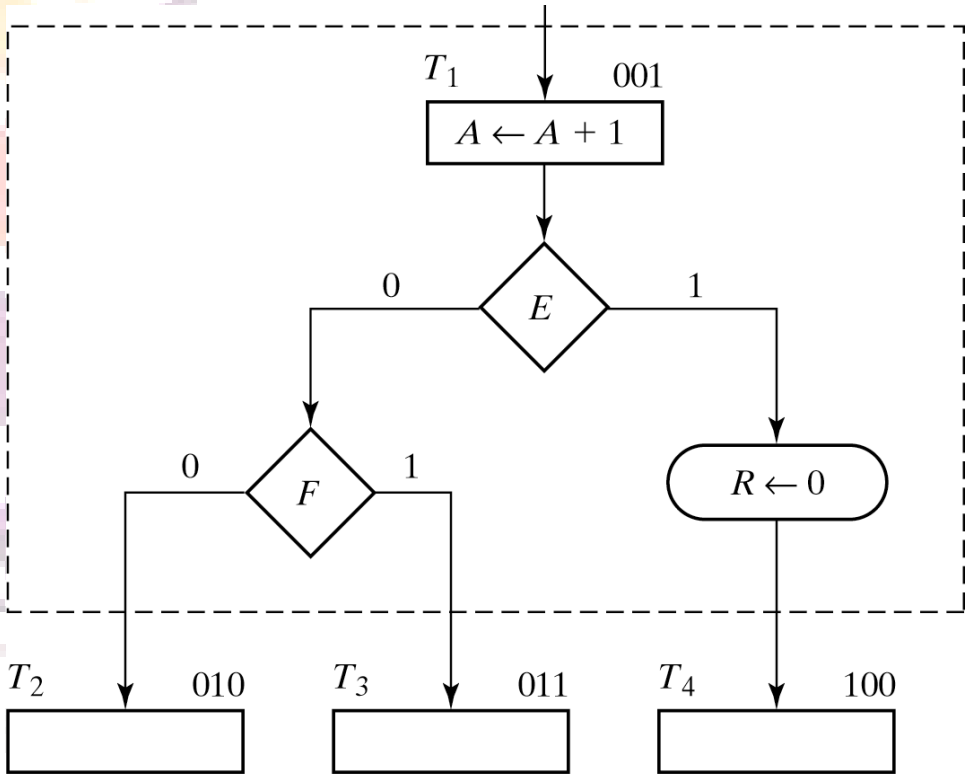
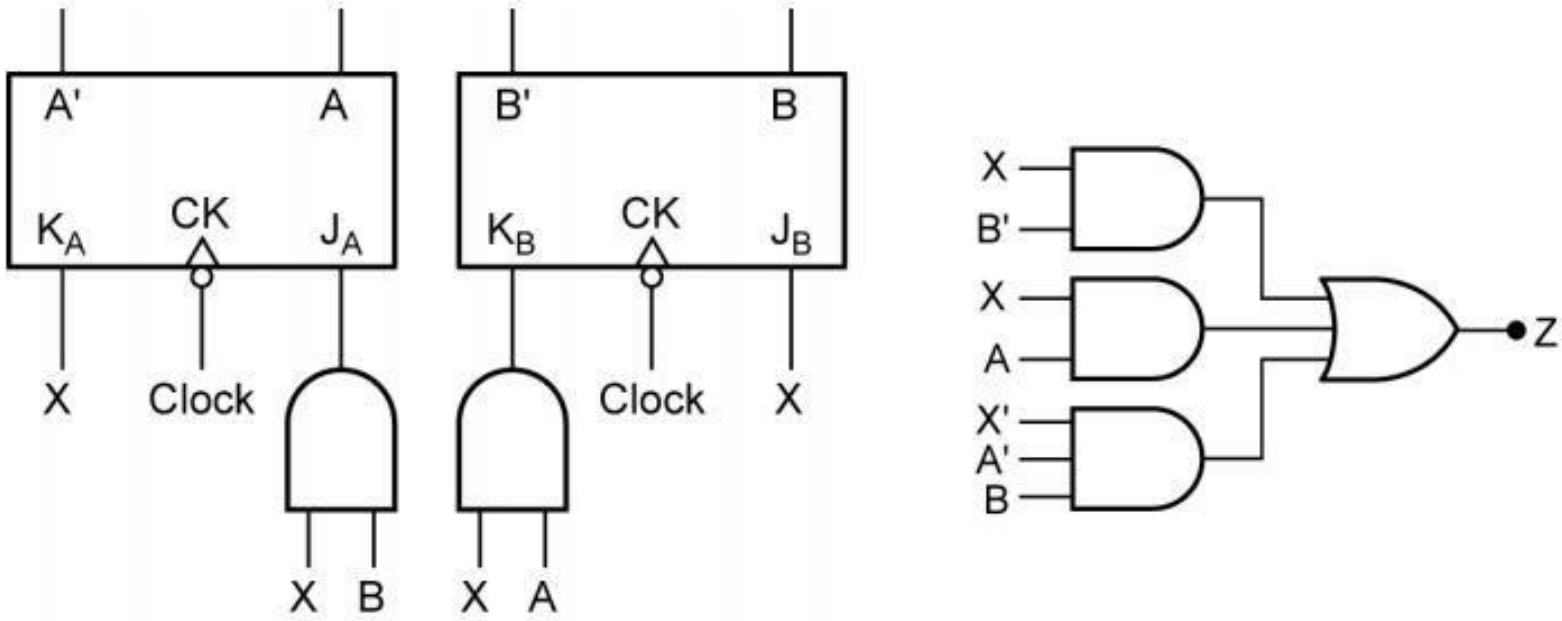


Fig. 8-6 ASM Block

ASM Block

- An ASM block describes the operation of the system during the state time in which it is in the state associated with the block.
- The outputs listed in the state box are asserted.
- The conditions indicated in the decision boxes are evaluated simultaneously to determine which link path is to be followed.
- If a conditional box is found in the selected path then the outputs found in its output list are asserted.
- Boolean expression may be written for each link path. The selected link paths are those that evaluate to logic-1.

Analyze a sequential circuit using JK Flip-Flops



Analysis (JK FF)

The flip-flop input equations are:

$$J_A = X.B \qquad J_B = X$$
$$K_A = X \qquad K_B = X.A$$

The sequential circuit output equation is:

$$Z = X.B' + X.A + X'.A'.B$$

The next-state equations for the flip-flops are:

$$A^+ = J_A.A' + K_A'.A \qquad B^+ = J_B.B' + K_B'.B$$
$$A^+ = X.B.A' + X.A \qquad B^+ = X.B' + X.A.B$$

The corresponding next-state (K-) maps are

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

A^+

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

B^+

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

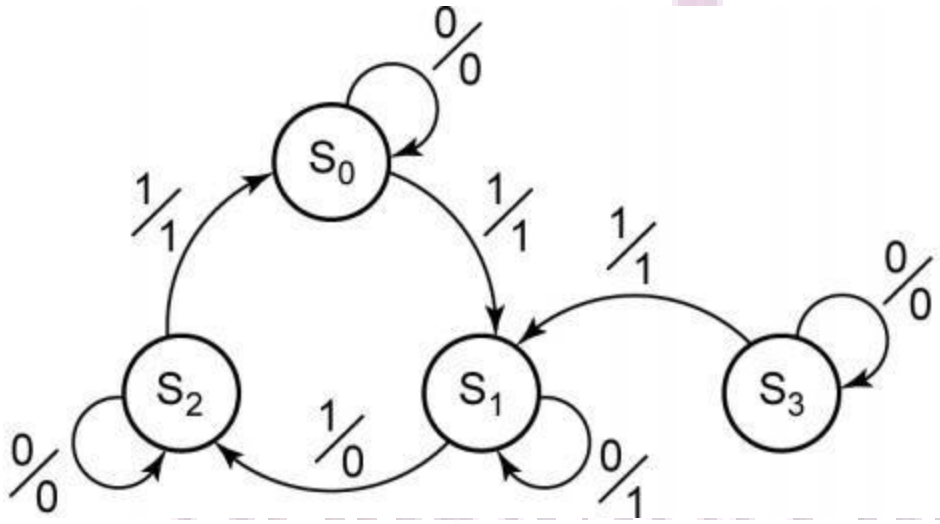
Z

The state table, and transition table, is then:

AB	A^+B^+ $X=0$	A^+B^+ 1	Z $X=0$	Z 1
00	00	01	0	1
01	01	11	1	0
11	11	00	0	1
10	10	01	0	1

Present State	Next State $X=0$	Next State 1	Present Output $X=0$	Present Output 1
S_0	S_0	S_1	0	1
S_1	S_1	S_2	1	0
S_2	S_2	S_0	0	1
S_3	S_3	S_1	0	1

The state diagram can then be drawn from the state table:



MODE OF OPERATIONS

❑Steady-state condition: Current states and next states are the same Difference between Y and y will cause a transition

❑Fundamental mode:

•No simultaneous changes of two or more variables

• The time between two input changes must be longer than the time it takes the circuit to a stable state

•The input signals change one at a time and only when the circuit is in a stable condition Fundamental Mode

❑Pulse Mode:

▪ the inputs and outputs are represented by pulses

▪only one input is allowed to have pulse present at any time

▪ Similar to synchronous sequential circuits except without a clock signal