

# Logic: Expression

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.ee.iitb.ac.in/~viren/>

E-mail: [viren@{cse, ee}.iitb.ac.in](mailto:viren@{cse, ee}.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

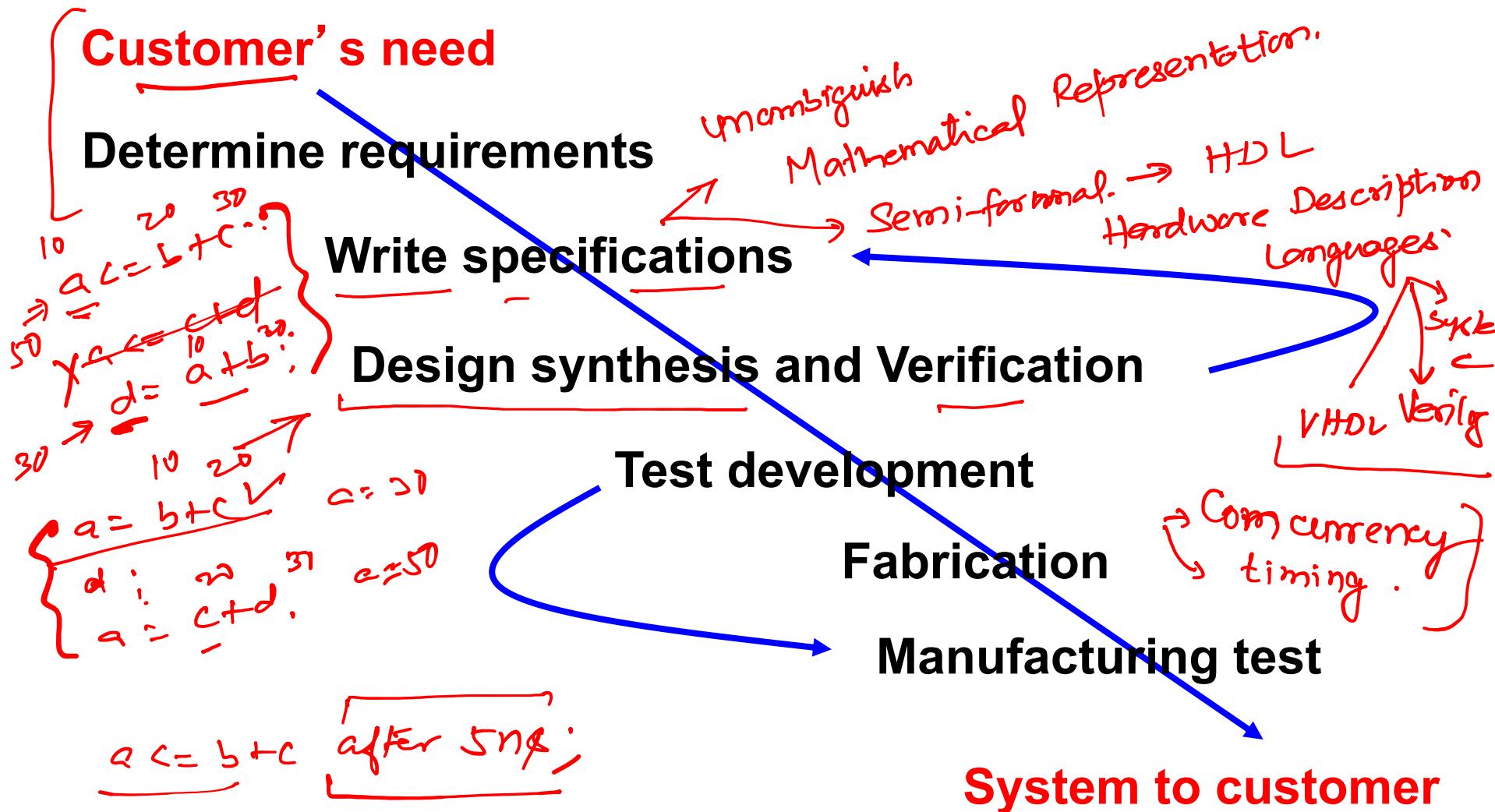
---



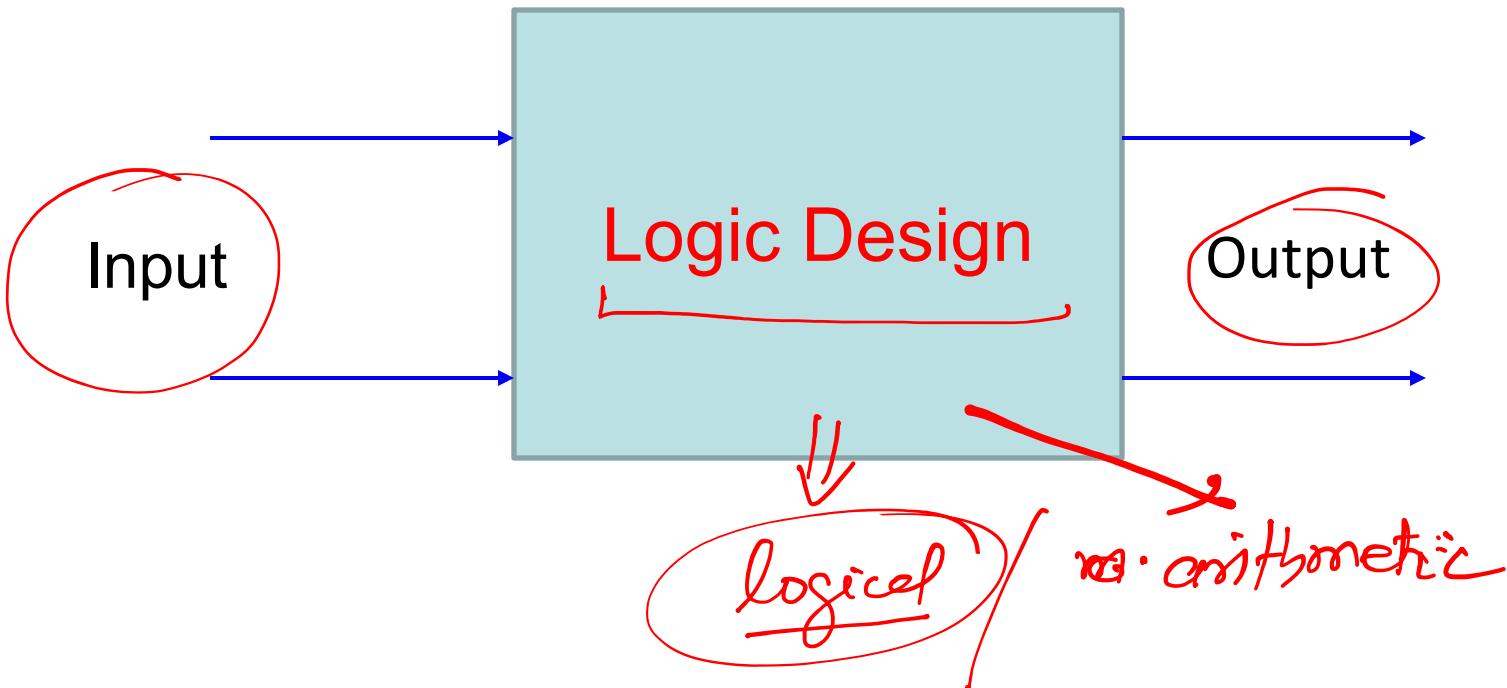
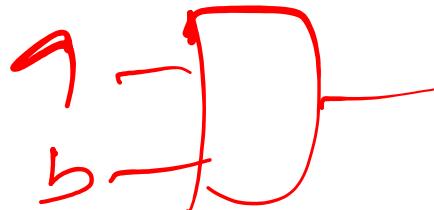
Lecture 2 (06 January 2022)

**CADSL**

# System Realization Process



# Digital System



2 - 5  
3 - 10  
4 - 20

# LOGIC



# What is logic?

---

- Logic is the study of valid reasoning.
- That is, logic tries to establish criteria to decide whether some piece of reasoning is valid or invalid.



# Logic

---

- Crucial for mathematical reasoning
- Used for designing electronic circuitry ✓
- Logic is a system based on propositions.
- A proposition is a statement that is either **true** or **false** (not both).



# Introduction: PL

---

- In Propositional Logic (a.k.a Propositional Calculus or Sentential Logic), the objects are called propositions
- **Definition:** A proposition is a statement that is either true or false, but not both
- We usually denote a proposition by a letter:  $p$ ,  $q$ ,  $r$ ,  $s$ , ...



# Introduction: Proposition

---

- **Definition:** The value of a proposition is called its **truth value**; denoted by
  - $T$  or 1 if it is true or
  - $F$  or 0 if it is false
- Opinions, interrogative, and imperative are not propositions
- **Truth table**

$p$
0
1



# Propositions: Examples

---

- The following are propositions

- Today is Thursday  $M$
- The floor is wet  $W$
- It is raining  $R$

- The following are not propositions

- Python is the best language *Opinion*
- When will be the next class? *Interrogative*
- Do your homework *Imperative*



# Logical Operators

---

- Operators/Connectives are used to create a compound proposition from two or more propositions
  - Negation (denote  $\neg$  or ! Or  $\sim$ )
  - And or logical conjunction (denoted  $\wedge$  or .) – logical AND
  - Or or logical disjunction (denoted  $\vee$  or +) – logical OR
  - XOR or exclusive or (denoted  $\oplus$ )
  - Implication (denoted  $\Rightarrow$  or  $\rightarrow$ )
  - Biconditional (denoted  $\Leftrightarrow$  or  $\leftrightarrow$ )

We define the meaning (semantics) of the logical operators/connectives using **truth tables**



# Logical Operator: Negation

---

- $\neg p$ , the negation of a proposition  $p$ , is also a proposition ✓
- Examples:
  - Today is not Monday.
- Truth table

$p$	$\neg p$
0 ✓	1 ✓
1 ✓	0 ✓



# Logical Operator: Logical And

- The logical operator **And** is true only when both of the propositions are true. It is also called a conjunction ✓
- Examples
  - It is raining and it is cold ✓
  - $(2+3=5)$  and  $(1<2)$   
    T                           T
- Truth table

$p$	$q$	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1



# Logical Operator: Logical Or

- The logical disjunction, or logical Or, is true if one or both of the propositions are true.
- Examples
  - It is raining or it is the second lecture
  - $(2+2=5) \vee (1<2)$
  - You may have cake or ice cream
- Truth table

$p$	$q$	$p \wedge q$	$p \vee q$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



# Logical Operator: Exclusive Or

- The exclusive Or, or XOR, of two propositions is true when exactly one of the propositions is true and the other one is false
- Example
  - The circuit is either ON or OFF but not both
  - Let  $ab < 0$ , then either  $a < 0$  or  $b < 0$  but not both
  - You may have cake or ice cream, but not both
- Truth table

$p$	$q$	$p \wedge q$	$p \vee q$	$p \oplus q$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



# Logical Operator: Implication

- **Definition:** Let  $p$  and  $q$  be two propositions. The implication  $p \rightarrow q$  is the proposition that is false when  $p$  is true and  $q$  is false and true otherwise
  - $p$  is called the hypothesis, antecedent, premise
  - $q$  is called the conclusion, consequence
- **Truth table**

$p$	$q$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1



# Logical Operator: Implication

---

- The implication of  $p \rightarrow q$  can be also read as
  - If  $p$  then  $q$  ✓  $\cancel{p \rightarrow q}$
  - $p$  implies  $q$
  - If  $p$ ,  $q$
  - $p$  only if  $q$
  - $q$  if  $p$
  - $q$  when  $p$
  - $q$  whenever  $p$
  - $q$  follows from  $p$
  - $p$  is a sufficient condition for  $q$  ( $p$  is sufficient for  $q$ )
  - $q$  is a necessary condition for  $p$  ( $q$  is necessary for  $p$ )



# Logical Operator: Implication

---

- Examples

- If you buy you air ticket in advance, it is cheaper.
- If  $x$  is an integer, then  $x^2 \geq 0$ .
- If it rains, the grass gets wet.
- If the sprinklers operate, the grass gets wet.



# Logical Operator: Equivalence

- **Definition:** The equivalence/biconditional  $p \leftrightarrow q$  is the proposition that is true when  $p$  and  $q$  have the same truth values. It is false otherwise.
- Note that it is equivalent to  $(p \rightarrow q) \wedge (q \rightarrow p)$
- **Truth table**

$p$	$q$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1



# Logical Operator: Equivalence

---

- The biconditional  $p \leftrightarrow q$  can be equivalently read as
  - $p$  if **and only** if  $q$
  - $p$  is a **necessary and sufficient** condition for  $q$
  - if  $p$  then  $q$ , and **conversely**
  - $p$  iff  $q$
- Examples
  - $x > 0$  if and only if  $x^2$  is positive
  - You may have pudding iff you eat your meal



# Truth Tables

---

- Truth tables are used to show/define the relationships between the truth values of
  - the individual propositions and
  - the compound propositions based on them

$p$	$q$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1



# Constructing Truth Tables

- Construct the truth table for the following compound proposition

$$((\underline{p \wedge q}) \vee \neg q)$$

✓ LOGICAL

$p$	$q$	$p \wedge q$	$\neg q$	$((p \wedge q) \vee \neg q)$
0	0	0	1	1
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1



# How to Specify Arithmetic Operations?

✗  
✗



# Number System



# Why Binary Arithmetic?

$$\begin{array}{r} 0 \xrightarrow{\quad} 9 \\ \text{---} \\ 3 + 5 \\ \text{---} \\ 0011 + 0101 \end{array}$$



$$= 8 \checkmark$$



$$= \underline{1000}$$

# Number Systems – Representation

- Positive radix, positional number systems
- A number with *radix r* is represented by a string of digits:

$$A_{n-1} A_{n-2} \dots A_1 A_0 . A_{-1} A_{-2} \dots A_{-m+1} A_{-m}$$

in which  $0 \leq A_i < r$  and  $.$  is the *radix point*.

- The string of digits represents the power series:

$$(Number)_r = \left( \sum_{i=0}^{i=n-1} A_i \cdot r^i \right) + \left( \sum_{j=-m}^{j=-1} A_j \cdot r^j \right)$$

(Integer Portion)      +      (Fraction Portion)



279.32  
A<sub>3</sub>A<sub>2</sub>A<sub>1</sub>.A<sub>-1</sub>A<sub>-2</sub>

10

$2 \times 10^2 +$

$7 \times 10^1 *$

$9 \times 10^0 +$

$3 \times 10^{-1} +$

$2 \times 10^{-2}$

# Number Systems – Examples

	General	Decimal	Binary
Radix (Base)	r	10 ✓	2 ✓
Digits	0 => r - 1	0 => 9	0 => 1
Powers of Radix	$r^0$	1	1
	$r^1$	10	2
	$r^2$	100	4
	$r^3$	1000	8
	$r^4$	10,000	16
	$r^5$	100,000	32
	$r^{-1}$	0.1	0.5
	$r^{-2}$	0.01	0.25
	$r^{-3}$	0.001	0.125
	$r^{-4}$	0.0001	0.0625
	$r^{-5}$	0.00001	0.03125



# Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152



# Converting Binary to Decimal

- To convert to decimal, use decimal arithmetic to form S (digit  $\times$  respective power of 2).
- Example: Convert  $11010_2$  to  $N_{10}$ :

Powers of 2: 4 3 2 1 0

1 1 0 1 0

$$1 \times 2^4 = 16$$

$$1 \times 2^3 = 8$$

$$0 \times 2^2 = 0$$

$$1 \times 2^1 = 2$$

$$0 \times 2^0 = 0$$

Sum

$$\rightarrow = 26_{10}$$

$$1 \times 16 + 1 \times 8 + 0 \times 4 + 2 \times 1 + 0 \times 2^0$$



# Converting Decimal to Binary

---

- Method 1
  - Subtract the largest power of 2 that gives a positive remainder and record the power.
  - Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero.
  - Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's.
- Example: Convert  $625_{10}$  to  $N_2$



# Converting Decimal to Binary

- Subtract the largest power of 2 (see slide 14) that gives a positive remainder and record the power.
- Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero.
- Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's.
- Example: Convert  $625_{10}$  to  $N_2$

9 8 7 6 5 4 3 2 1 0  
1 0 0 1 1 1 0 0 0 1

625 - 512	= 113	⇒ 9 .. ✓
113 - 64	= 49 ✓	⇒ 6 - ✓
49 - 32	= 17	⇒ 5
17 - 16	= 1	⇒ 4
1 - 1	= 0	⇒ 0
Placing 1's in the result for the positions recorded and 0's elsewhere:		
9876543210		
1001110001		



# Commonly Occurring Bases

---

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

The six letters (in addition to the 10 integers) in hexadecimal represent:

10, 11, 12, 13, 14, 15



# Binary Arithmetic



# Single Bit Binary Addition

Given two binary digits (X,Y), we get the following sum (S) and carry (C):

0 1

2

10

$$\begin{array}{r} X & 0 & 0 & 1 & 1 \\ + Y & + 0 & + 1 & + 0 & + 1 \\ \hline CS & 0 0 & 0 1 & 0 1 & 1 0 \end{array}$$

Handwritten annotations in red:

- A red checkmark is placed under the first '0' in the CS row.
- A red checkmark is placed under the second '0' in the CS row.
- A red checkmark is placed under the third '0' in the CS row.
- A red checkmark is placed under the fourth '0' in the CS row.
- A red checkmark is placed under the fifth '0' in the CS row.
- A red circle highlights the '1' in the fifth column of the CS row.
- A red arrow points from the circled '1' to the circled '1' in the fifth column of the sum row.
- A red circle highlights the '1' in the fifth column of the sum row.
- A red circle highlights the '2' above the fifth column of the sum row.



# Truth Table: Two Bit Adder

X	Y	Binary Sum (C)(S)
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

$$\begin{array}{l} x = \boxed{\phantom{0}} \\ y = \boxed{\phantom{0}} \end{array} \quad \begin{array}{l} c \\ s \end{array}$$

1 0

CARRY      SUM  $\oplus$



# Truth Tables of Logical Operations

- Truth tables are used to show/define the **relationships** between the truth values of
  - the individual propositions and
  - the compound propositions based on them

1 1  
1 2  
1 3  
3 2 2

$p$	$q$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1



# Single Bit Binary Addition with Carry

---

Given two binary digits (X,Y), a carry in (Z) we get the following sum (S) and carry (C):

Carry in (Z) of 0:

$$\begin{array}{r} z \quad 0 \quad 0 \quad 0 \quad 0 \\ x \quad 0 \quad 0 \quad 1 \quad 1 \\ + y \quad + 0 \quad + 1 \quad + 0 \quad + 1 \\ \hline c s \quad \underline{\underline{0}} \underline{\underline{0}} \quad \underline{\underline{0}} \underline{\underline{1}} \quad \underline{\underline{0}} \underline{\underline{1}} \quad \underline{\underline{1}} \underline{\underline{0}} \end{array}$$

Carry in (Z) of 1:

$$\begin{array}{r} z \quad 1 \quad 1 \quad 1 \quad 1 \\ x \quad 0 \quad 0 \quad 1 \quad 1 \\ + y \quad + 0 \quad + 1 \quad + 0 \quad + 1 \\ \hline c s \quad \underline{\underline{0}} \underline{\underline{1}} \quad \underline{\underline{1}} \underline{\underline{0}} \quad \underline{\underline{1}} \underline{\underline{0}} \quad \underline{\underline{1}} \underline{\underline{1}} \end{array}$$



# Full Adder: Include Carry Input

Z	Y	X	$S = X + Y + Z$	
			Decimal value	Binary value
0	0	0	0	0 0
0	0	1	1	0 1
0	1	0	1	0 1
0	1	1	2	1 0
1	0	0	1	0 1
1	0	1	2	1 0
1	1	0	2	1 0
1	1	1	3	1 1

CARRY      SUM



# Truth Table: Full Adder

Z	Y	X	Binary value (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

CARRY      SUM



# Multiple Bit Binary Addition

---

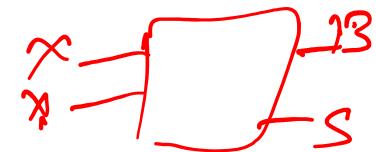
- Extending this to a multiple bit examples:

Carries	<u>00000</u>
Augend	01100
Addend	10001
Sum	<u>11101</u>



# Single Bit Binary Subtraction

- Given two binary digits ( $X, Y$ ), we get the following difference ( $S$ ) and borrow ( $B$ )



X	0	0.	1.	1 .
- Y	- 0	- 1.	- 0	- 1 .
BS	0 0	1 1	0 1	0 0

Red annotations: A red checkmark is under the first '0' in the BS row. A red circle highlights the '1 1' in the BS row. A red arrow points from the '1 1' circle to the word 'Borrow' at the bottom. Red checkmarks are also under the '1' in the BS row and the '0' in the BS row for the last two columns.

*Borrow*



# Truth Table: Two Bit Subtractor

X	Y	Binary Difference (B)(D)
0	0	0 0
0	1	1 1
1	0	0 1
1	1	0 0

# BORROW

# DIFFERENCE



# Single Bit Binary Subtraction with Borrow

- Given two binary digits (X,Y), a borrow in (Z) we get the following difference (D) and borrow (B):
- Borrow in (Z) of 0:

Z	0	0	0	0	0
X	0	0	1	1	1
<u>- Y</u>	<u>-0</u>	<u>-1</u>	<u>-0</u>	<u>-1</u>	
BD	0 0	1 1	0 1	0 0	

- Borrow in (Z) of 1:

Z	1	1	1	1	1
X	0	0	1	1	1
<u>- Y</u>	<u>-0</u>	<u>-1</u>	<u>-0</u>	<u>-1</u>	
BD	1 1	1 0	0 0	1 1	



# Truth Table: Full Subtractor

Z	Y	X	Binary value (C)(D)
0	0	0	0 0
0	0	1	1 1
0	1	0	0 1
0	1	1	0 0
1	0	0	1 1
1	0	1	1 0
1	1	0	0 0
1	1	1	1 1



# Multiple Bit Binary Subtraction

---

- Extending this to a multiple bit example:
- Notes:
  - The 0 is a Borrow-In to the least significant bit.
  - If the Subtrahend > the Minuend, interchange and append a – to the result.

Borrows	<u>00000</u>
Minuend	10110
Subtrahend	10010
Difference	<u>00100</u>



# Multiple Bit Binary Subtraction

---

- Extending this to a multiple bit examples:
- Notes: The 0 is a Borrow-In to the least significant bit. If the Subtrahend > the Minuend, interchange and append a – to the result.

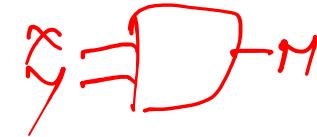
Borrows	<u>00110</u>
Minuend	10110
Subtrahend	10011
Difference	<u>00011</u>



# Binary Multiplication

The binary multiplication table is simple:

$$0 * 0 = 0 \quad | \quad 1 * 0 = 0 \quad | \quad 0 * 1 = 0 \quad | \quad 1 * 1 = 1$$



Extending multiplication to multiple digits:

Multiplicand

1011 ✓

Multiplier

x 101 ✓

Partial Products

1011

0000 -

1011 - -

110111

Product

Unsigned  
numbers



# Signed Magnitude?

---

- Use fixed length binary representation
- Use left-most bit (called *most significant bit* or MSB) for sign:

0 for positive

1 for negative

- Example:

$$+18_{\text{ten}} = \begin{array}{r} 0 \\ 00010010 \end{array}_{\text{two}}$$

MSB.

$$-18_{\text{ten}} = \begin{array}{r} 1 \\ 00010010 \end{array}_{\text{two}}$$

LSB



# Difficulties with Signed Magnitude

---

- Sign and magnitude bits should be differently treated in arithmetic operations.
- Addition and subtraction require different logic circuits.
- Overflow is difficult to detect.
- “Zero” has two representations:
  - + 0<sub>ten</sub> = 00000000<sub>two</sub>
  - 0<sub>ten</sub> = 10000000<sub>two</sub>
- *Signed-integers are not used in modern computers.*



# Thank You



# Logic: Expression

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@{cse, ee}.iitb.ac.in](mailto:viren@{cse, ee}.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 3 (10 January 2022)

**CADSL**

# Signed Number System



# Signed Magnitude?

64 bit

- Use fixed length binary representation
- Use left-most bit (called *most significant bit* or MSB) for sign:

0 for positive

1 for negative

- Example:  $+18_{\text{ten}} = \underline{\text{0}}0010010_{\text{two}}$

$$-18_{\text{ten}} = \underline{\text{1}}0010010_{\text{two}}$$

↑  
0  
1

85 bit  
0 - 255  
- 127 - 0 - 127  
16 bit



# Difficulties with Signed Magnitude

- Sign and magnitude bits should be differently treated in arithmetic operations.
- Addition and subtraction require different logic circuits.
- Overflow is difficult to detect.
- “Zero” has two representations:
  - $+0_{\text{ten}} = 00000000_{\text{two}}$
  - $-0_{\text{ten}} = 10000000_{\text{two}}$
- Signed-integers are not used in modern computers.

A handwritten diagram of binary addition:

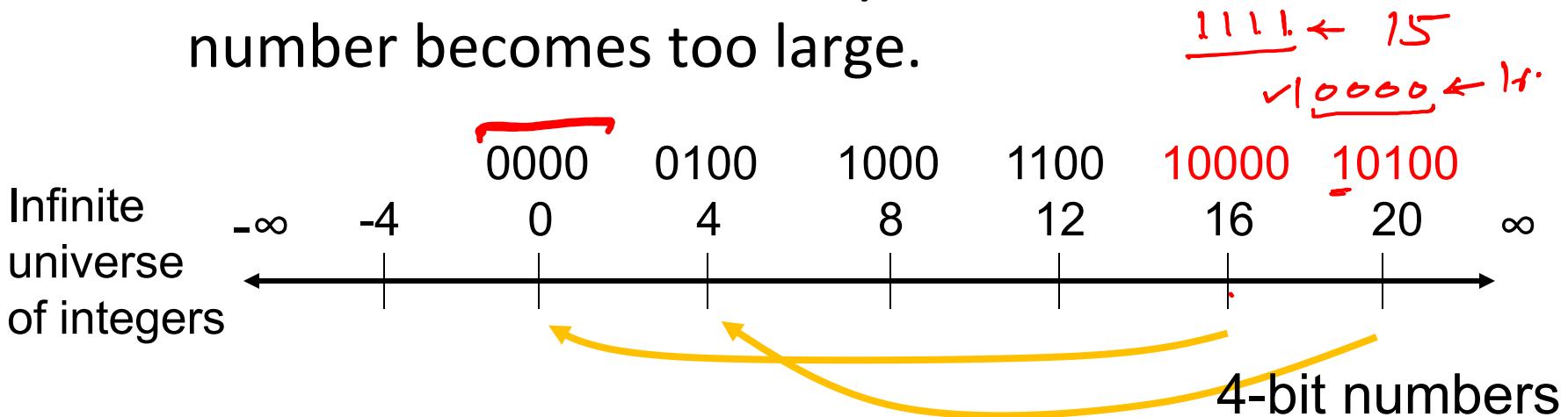
$$\begin{array}{r} 010 \\ + 111 \\ \hline 100 \end{array}$$

The result is circled in red with the word "overflow". A red arrow points from the circled result to the circled "overflow" label.

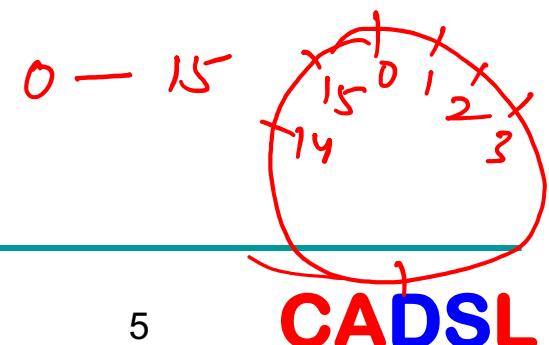


# Problems with Finite Math

- Finite size of representation:
  - Digital circuit cannot be arbitrarily large.
  - Overflow detection – easy to determine when the number becomes too large.



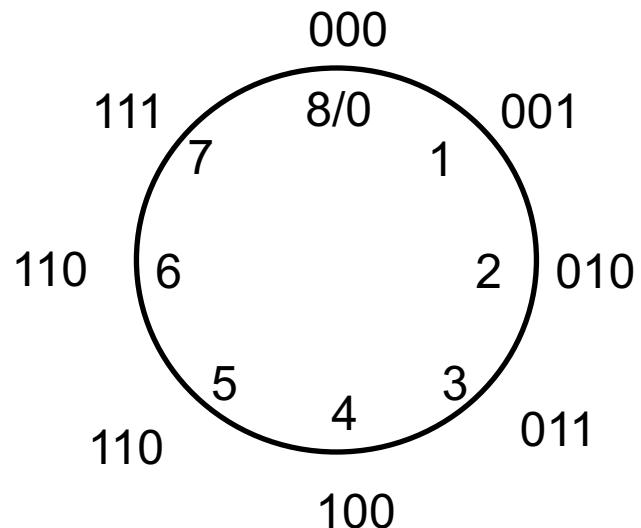
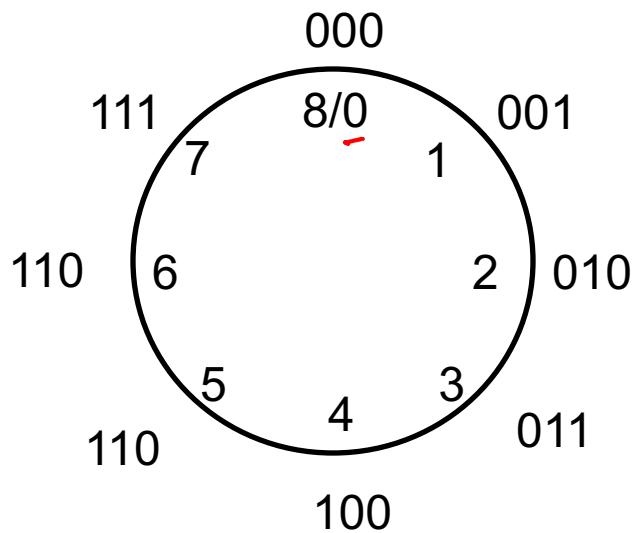
- Represent negative numbers:
  - Unique representation of 0.



# 3-bit Universe

---

Modulo-8  
(3-bit)  
universe

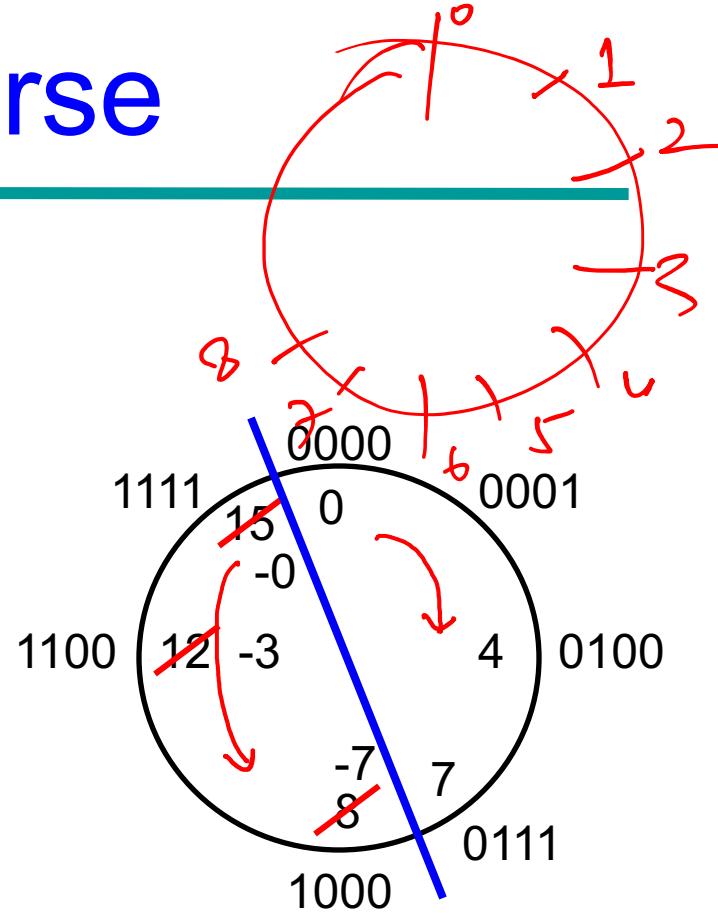
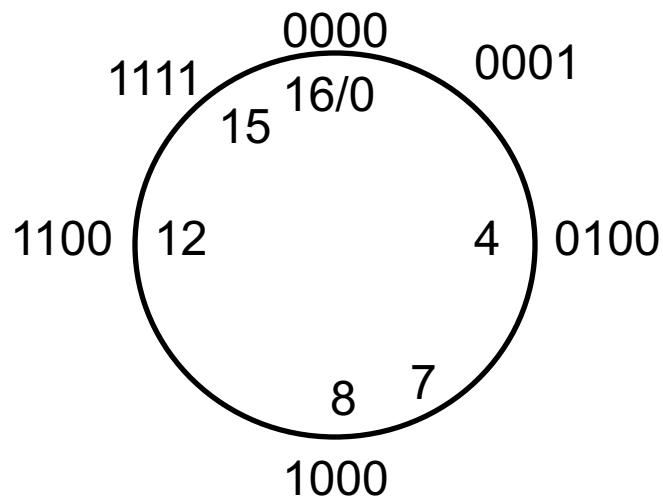


Only 8 integers: 0 through 7, or – 3 through 3



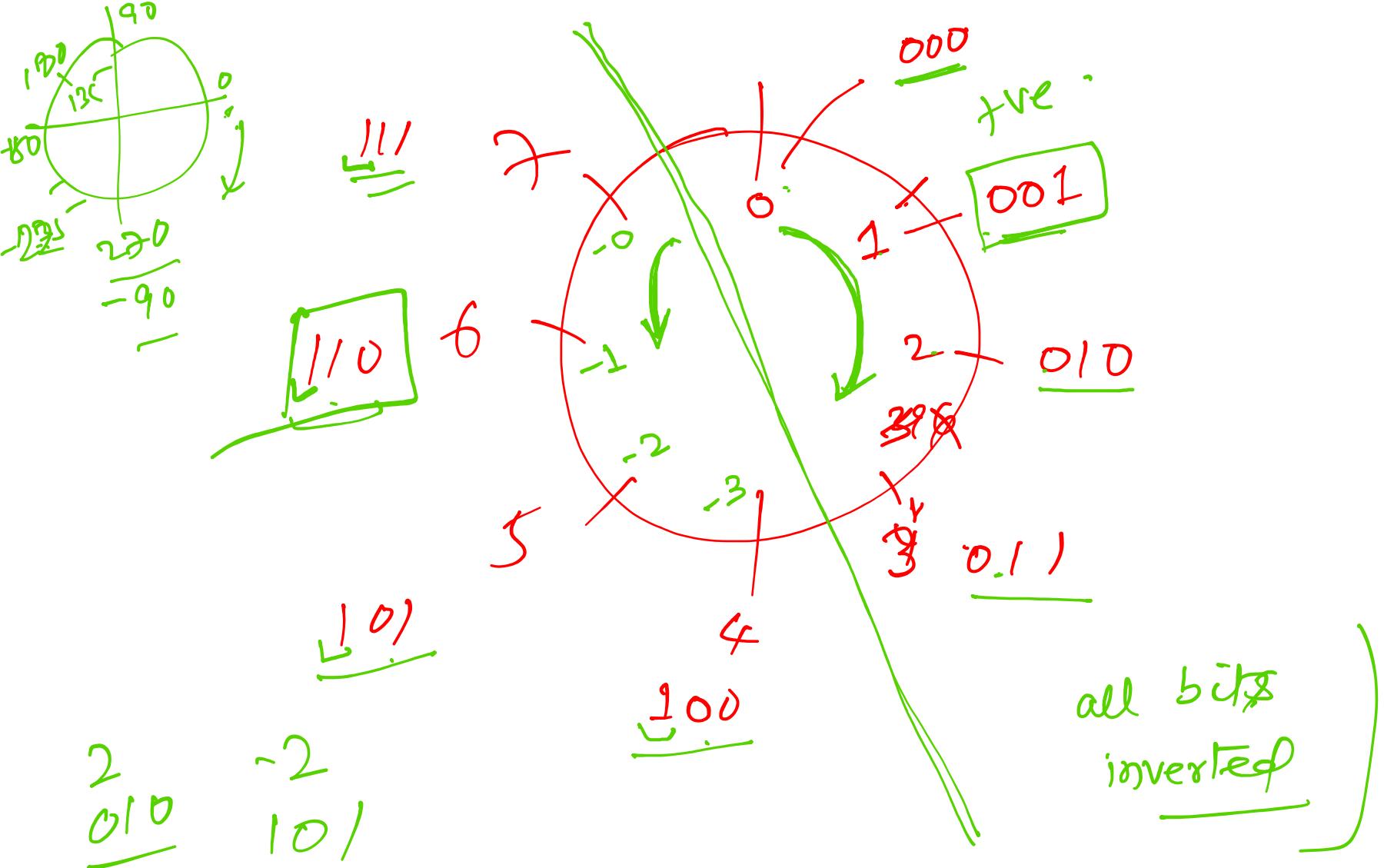
# 4-bit Universe

Modulo-16  
(4-bit)  
universe



Only 16 integers: 0 through 15, or – 7 through 7

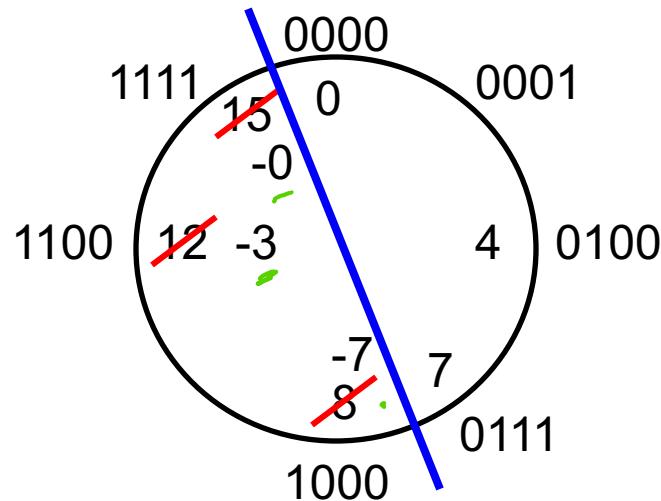




$0 \rightarrow 1$   
 $1 \rightarrow 0$



# One Way to Divide Universe 1's Complement Numbers



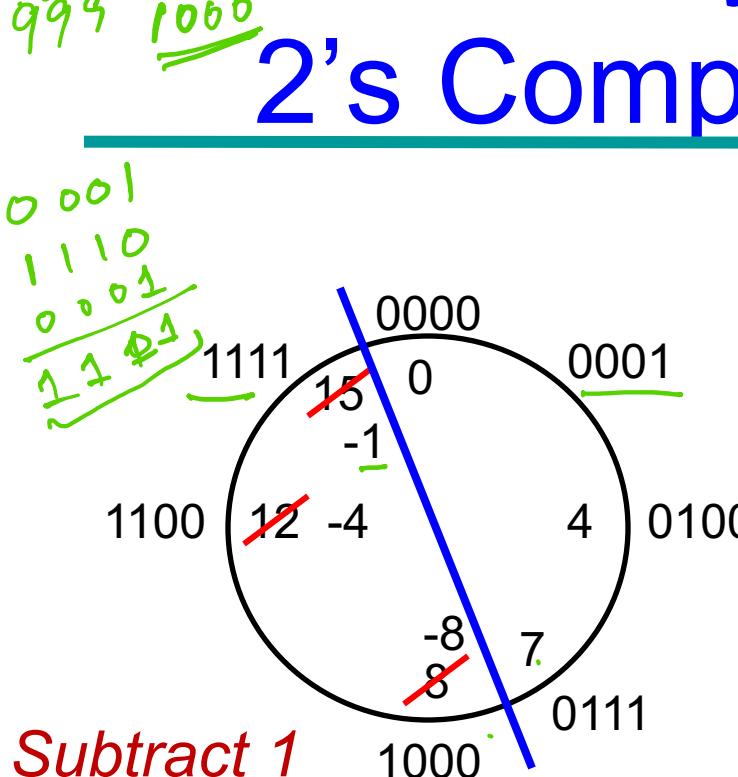
Negation rule: invert bits.

Problem:  $0 \neq -0$

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000



# Another Way to Divide Universe 2's Complement Numbers



Subtract 1  
on this side

Negation rule: invert bits  
and add 1

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	—
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	—	1000



# Integers With Sign – Two Ways

---

- Use fixed-length representation, but no explicit sign bit:
  - 1's complement: To form a negative number, complement each bit in the given number.
  - 2's complement: To form a negative number, start with the given number, subtract one, and then complement each bit, or  
*first complement each bit, and then add 1.*
- 2's complement is the preferred representation.



# 2's-Complement Integers

---

- Why not 1's-complement? *Don't like two zeros.*
- Negation rule:
  - Subtract 1 and then invert bits, or
  - Invert bits and add 1
- Some properties:
  - Only one representation for 0 ✓
  - Exactly as many positive numbers as negative numbers
  - Slight asymmetry – there is one negative number with no positive counterpart



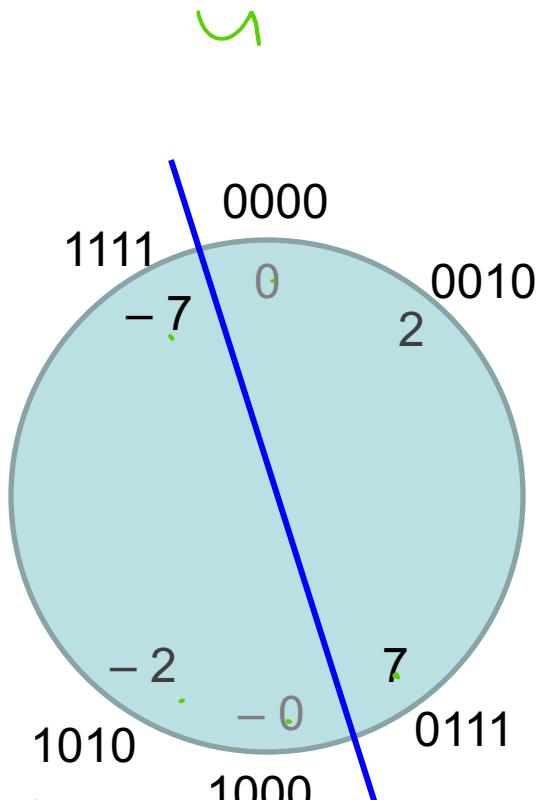
# General Method for Binary Integers with Sign

---

- Select number ( $n$ ) of bits in representation.
- Partition  $2^n$  integers into two sets:
  - 00...0 through 01...1 are  $2^n/2$  positive integers.
  - 10...0 through 11...1 are  $2^n/2$  negative integers.
- Negation rule transforms negative to positive, and vice-versa:
  - ✓ • Signed magnitude: invert MSB (most significant bit)
  - ✓ • 1's complement: Subtract from  $2^n - 1$  or 1...1 (same as “inverting all bits”)
  - ✓ • 2's complement: Subtract from  $2^n$  or 10...0 (same as 1's complement + 1)

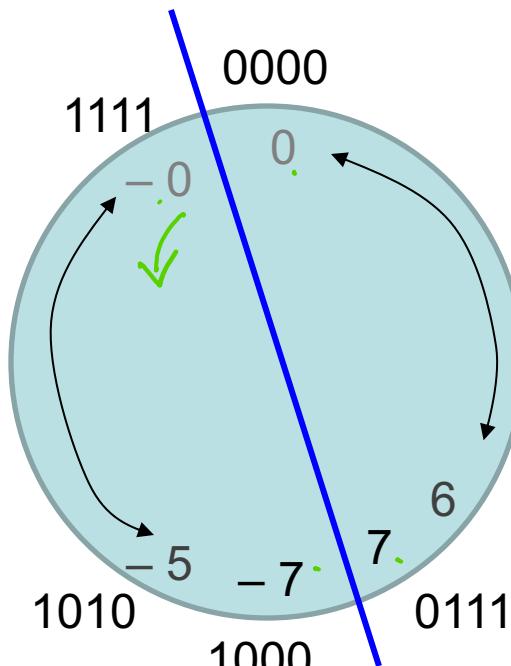


# Three Systems ( $n = 4$ )



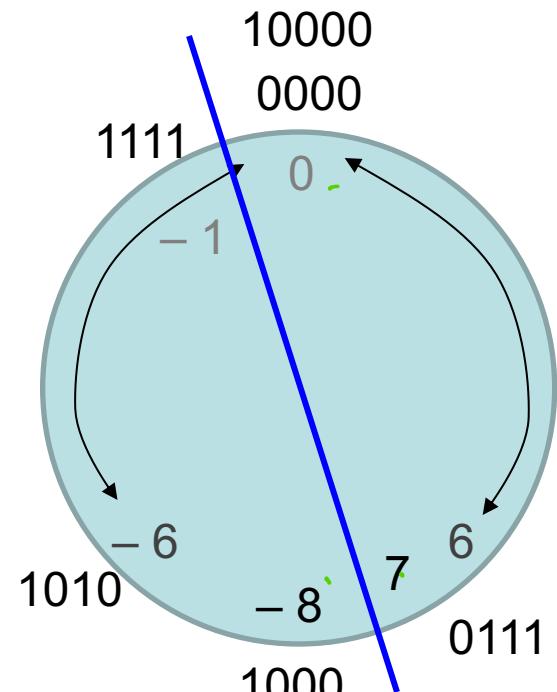
$$1010 = -2$$

Signed magnitude



$$1010 = -5$$

1's complement integers



$$1010 = -6$$

2's complement integers

# Three Representations

---

## Sign-magnitude

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = - 0  
101 = - 1  
110 = - 2  
111 = - 3

## 1's complement

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = - 3  
101 = - 2  
110 = - 1  
111 = - 0

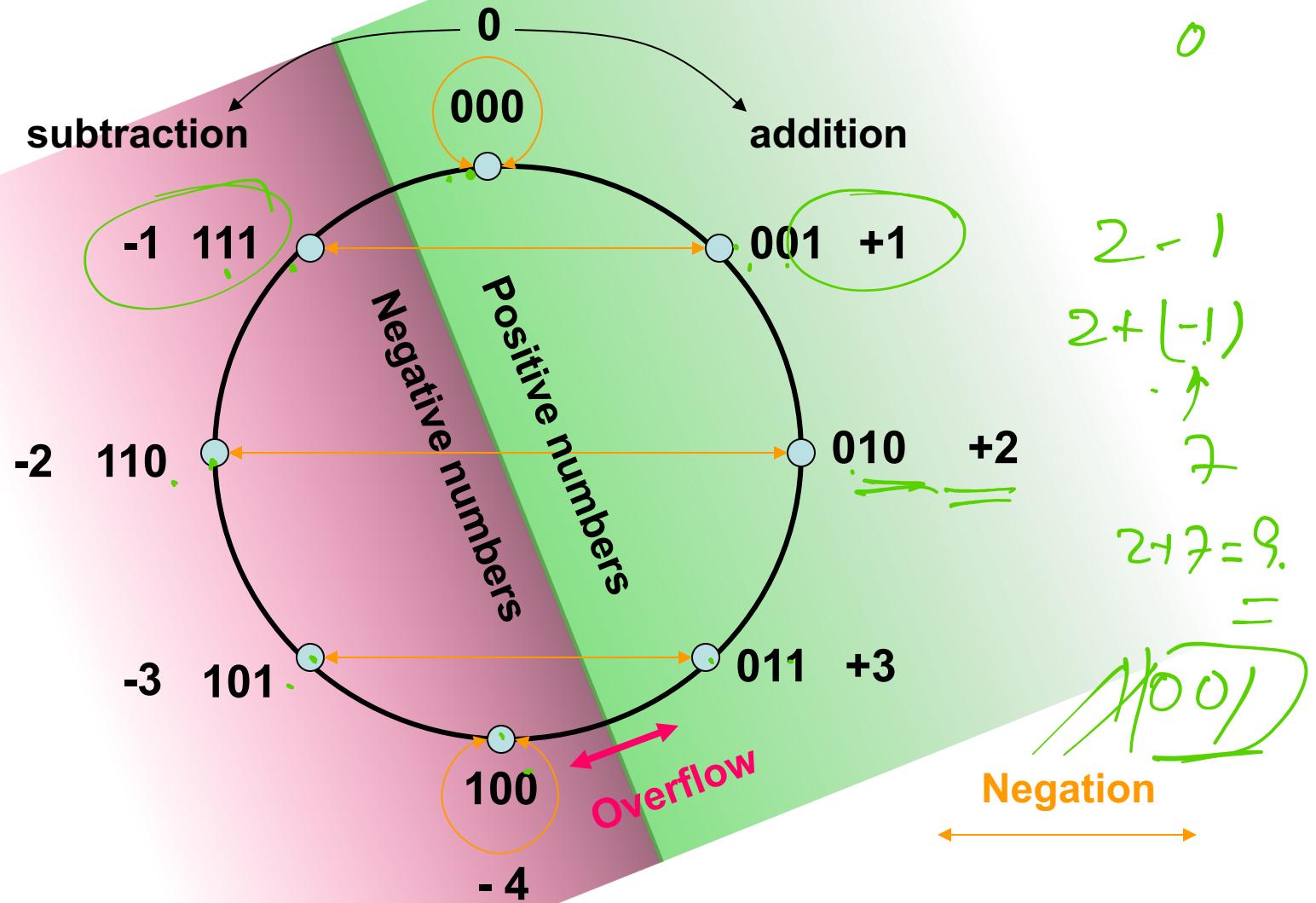
## 2's complement

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = - 4  
101 = - 3  
110 = - 2  
111 = - 1

(Preferred)



# 2's Complement Numbers ( $n = 3$ )



# Summary

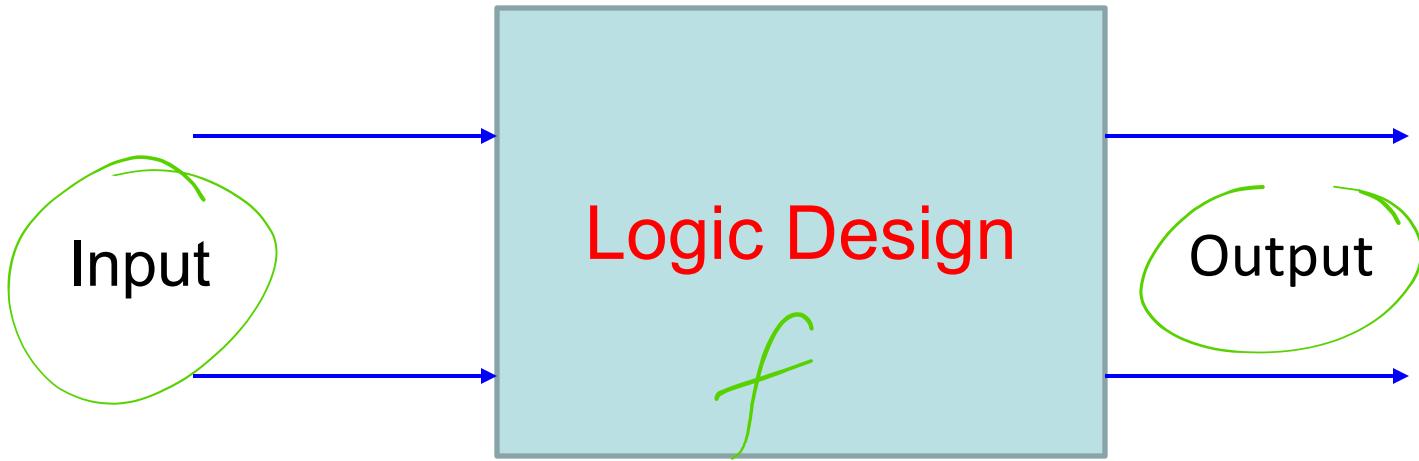
---

- For a given number ( $n$ ) of digits we have a finite set of integers. For example, there are  $10^3 = 1,000$  decimal integers and  $2^3 = 8$  binary integers in 3-digit representations.
- We divide the finite set of integers  $[0, r^n - 1]$ , where radix  $r = 10$  or  $2$ , into two equal parts representing positive and negative numbers.
- Positive and negative numbers of equal magnitudes are complements of each other:  $x + \text{complement}(x) = 0$ .



# Digital System

---



$$\text{output} = f(\underline{\text{input}}) \quad \{0,1\}$$

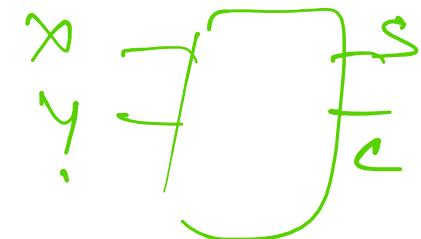
# I/O Behaviour: Automobile Ignition

---

- Engine turns on when
- Ignition key is applied AND
  - either Car is in parking gear OR
  - Brake pedal is on
- AND
  - either Seat belt is fastened OR
  - Car is in parking gear



# Truth Table: Half Adder



X	Y	Binary Sum (C)(S)
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

## CARRY

## SUM



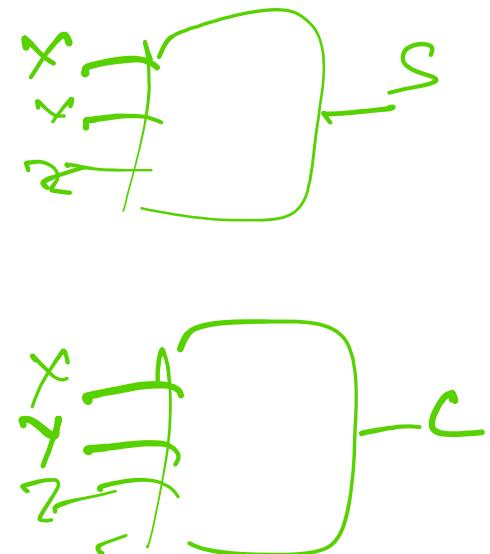
# ✓ Truth Table: Full Adder

Z	Y	X	Binary value (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



# Truth Table: Full Adder

Z	Y	X	Carry C	Sum 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



# Truth Tables of logical functions

---

- Truth tables are used to show/define the relationships between the truth values of
  - the individual propositions and
  - the compound propositions based on them

$p$	$q$	$p \cdot q$	$P+q$	$p \oplus q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1



# Logic Expressions

$$\frac{3+2}{3-2} = 3+(-2)$$

## Truth Table

X Y Z	F
0 0 0	0
0 0 1	1 ✓
0 1 0	0
0 1 1	0
1 0 0	1 ✓.
1 0 1	1 ✓ .
1 1 0	1 ✓ .
1 1 1	1 ✓ .

## Logic Expression

$$F = \overline{\cancel{X}} \cdot \overline{\cancel{Y}} \cdot \cancel{Z} + \cancel{X} \cdot \overline{\cancel{Y}} \cdot \cancel{Z} + \cancel{X} \cdot \overline{\cancel{Y}} \cdot \cancel{Z}$$

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

if

✓

- Logic expressions, truth tables describe the same function!
- Truth tables are unique; expressions are not. This gives flexibility in implementing functions.



# Thank You



# Logic: Implementation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@{cse, ee}.iitb.ac.in](mailto:viren@{cse, ee}.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



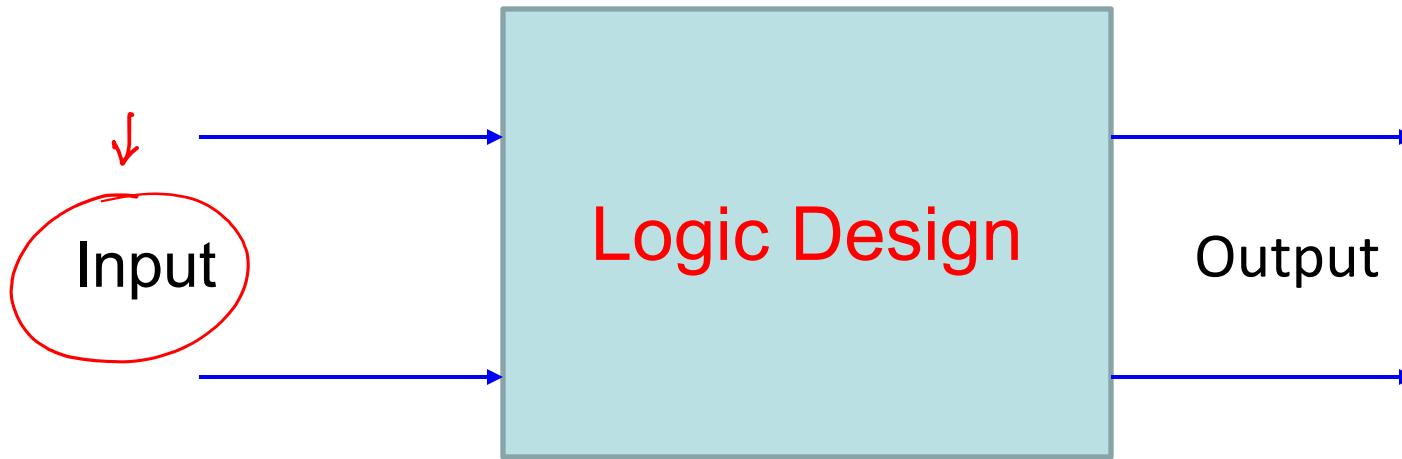
Lecture 4 (11 January 2022)

**CADSL**

# Digital System

---

Truth Table :



$$\text{Output} = f(\text{Input})$$



# canonical form Logic Expressions

Truth Table ✓

$$n=3$$

X Y Z	F
0 0 0	0 1
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

$$2^2 = 8$$

$$\{0,1\}$$

Logic Expression ✓

$$F = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z \\ + X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$$



$$2^{2^n}$$

$$2^n$$

- Logic expressions, truth tables describe the same function!
- Truth tables are unique; expressions are not. This gives flexibility in implementing functions.

Optimization  $\Rightarrow$  minimise cost



# How Many Logic Functions?

- Output column of truth table has length  $2^n$  for  $n$  input variables.
- It can be arranged in  $2^{2^n}$  ways for  $n$  variables.
- Example:  $n = 1$ , single variable.

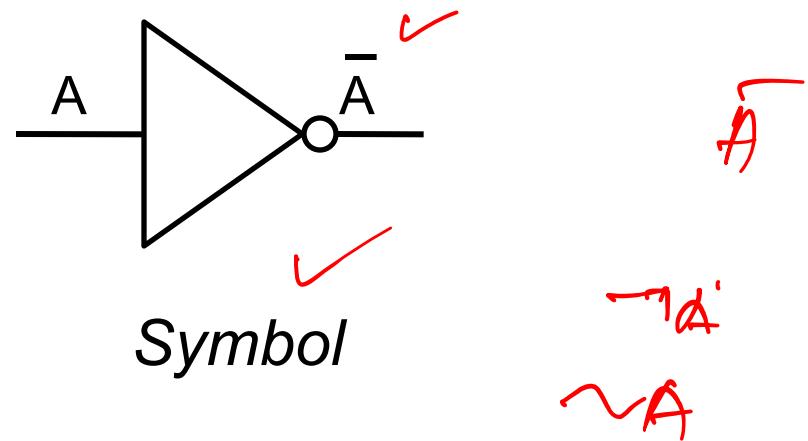
2<sup>2<sup>n</sup></sup>  
2<sup>2<sup>1</sup></sup> = 4

Input	Output functions			
A	F1(A)	F2(A)	F3(A)	F4(A)
0.	0 ]	0 :	1 :	1 .
1.	0 ]	1 :	0 .	1 .



# NOT Gate

Truth Table	
A	$\bar{A}$
0	1
1	0



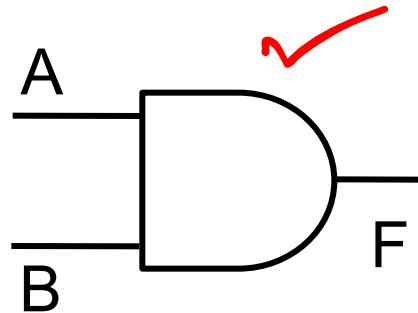
*Logic Function*



# AND Gate

---

*Symbol*



*Logic Function*

**Truth Table**

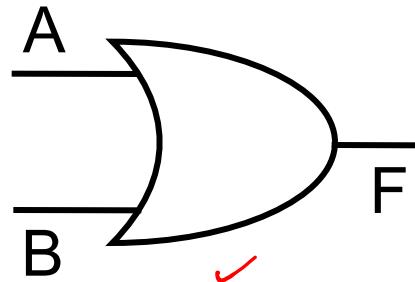
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



# OR Gate

---

*Symbol*



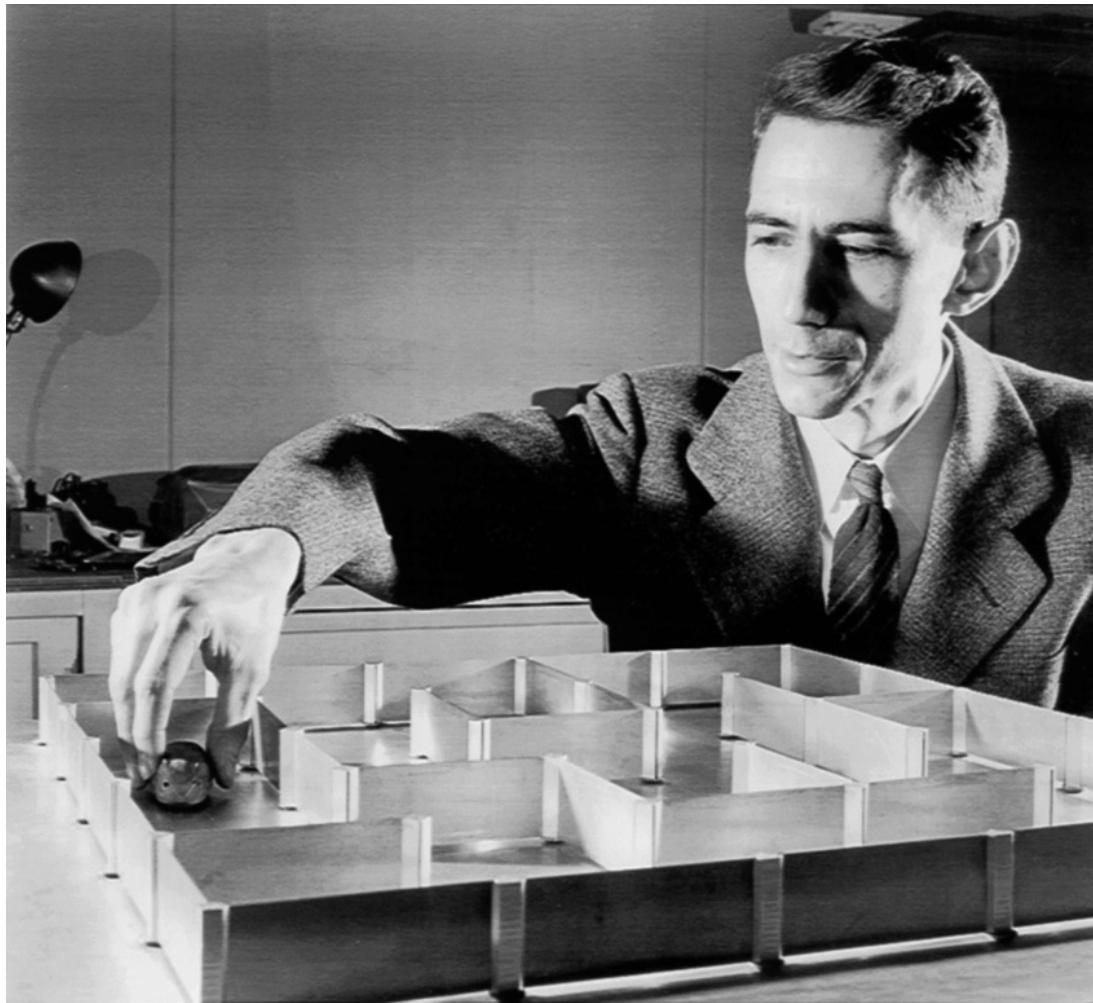
*Logic Function*

**Truth Table**

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



# Claude E. Shannon (1916-2001)

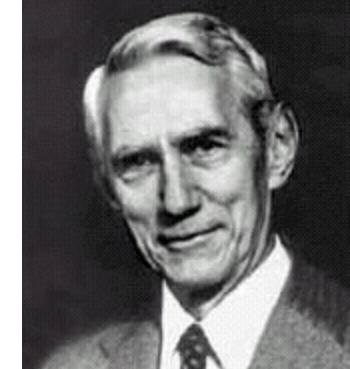


[http://www.kugelbahn.ch/sesam\\_e.htm](http://www.kugelbahn.ch/sesam_e.htm)



# Shannon's Legacy

---



- A *Symbolic Analysis of Relay and Switching Circuits*,  
Master's Thesis, MIT, 1940. Perhaps the most  
influential master's thesis of the 20<sup>th</sup> century.
- *An Algebra for Theoretical Genetics*, PhD Thesis, MIT,  
1940.
- Founded the field of Information Theory.
- C. E. Shannon and W. Weaver, *The Mathematical  
Theory of Communication*, University of Illinois Press,  
1949. A “must read.”



# Logic Function Implementation

- Using Switches

- For inputs:

- logic 1 is switch closed
    - logic 0 is switch open

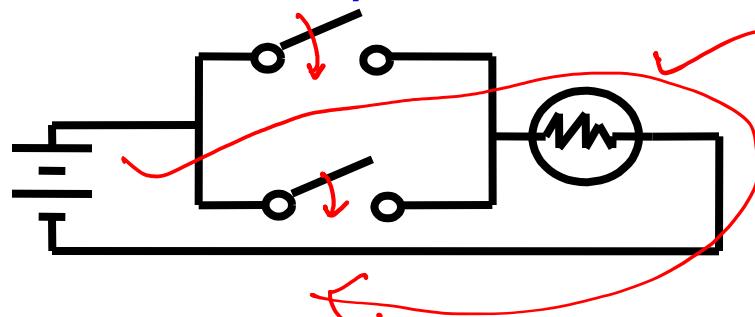
- For outputs:

- logic 1 is light on
    - logic 0 is light off.

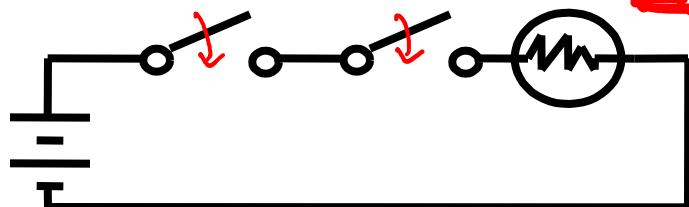
- NOT uses a switch such  
that:

- logic 1 is switch open
    - logic 0 is switch closed

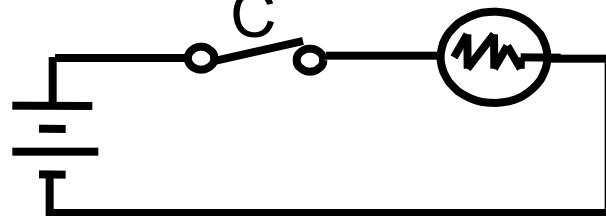
Switches in parallel => OR



Switches in series => AND



Normally-closed switch => NOT



# Switching Devices

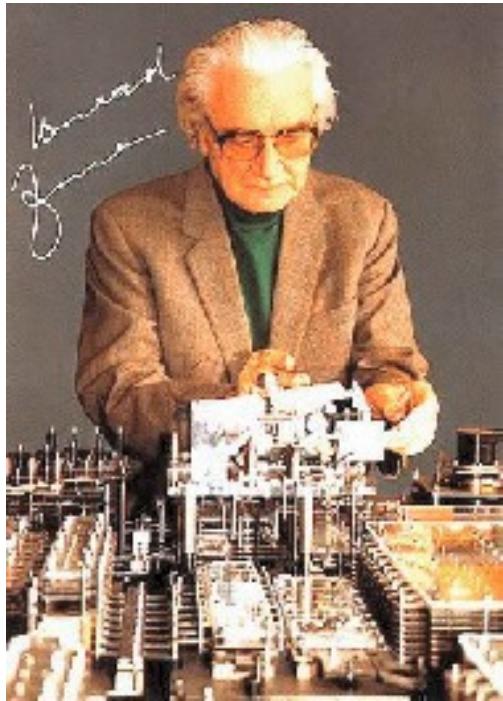
---

- Electromechanical relays (1940s)
- Vacuum tubes (1950s)
- Bipolar transistors (1960 - 1980) ↗ BJT
- Field effect transistors (1980 - ) ✓
- Integrated circuits (1970 - ) ↗

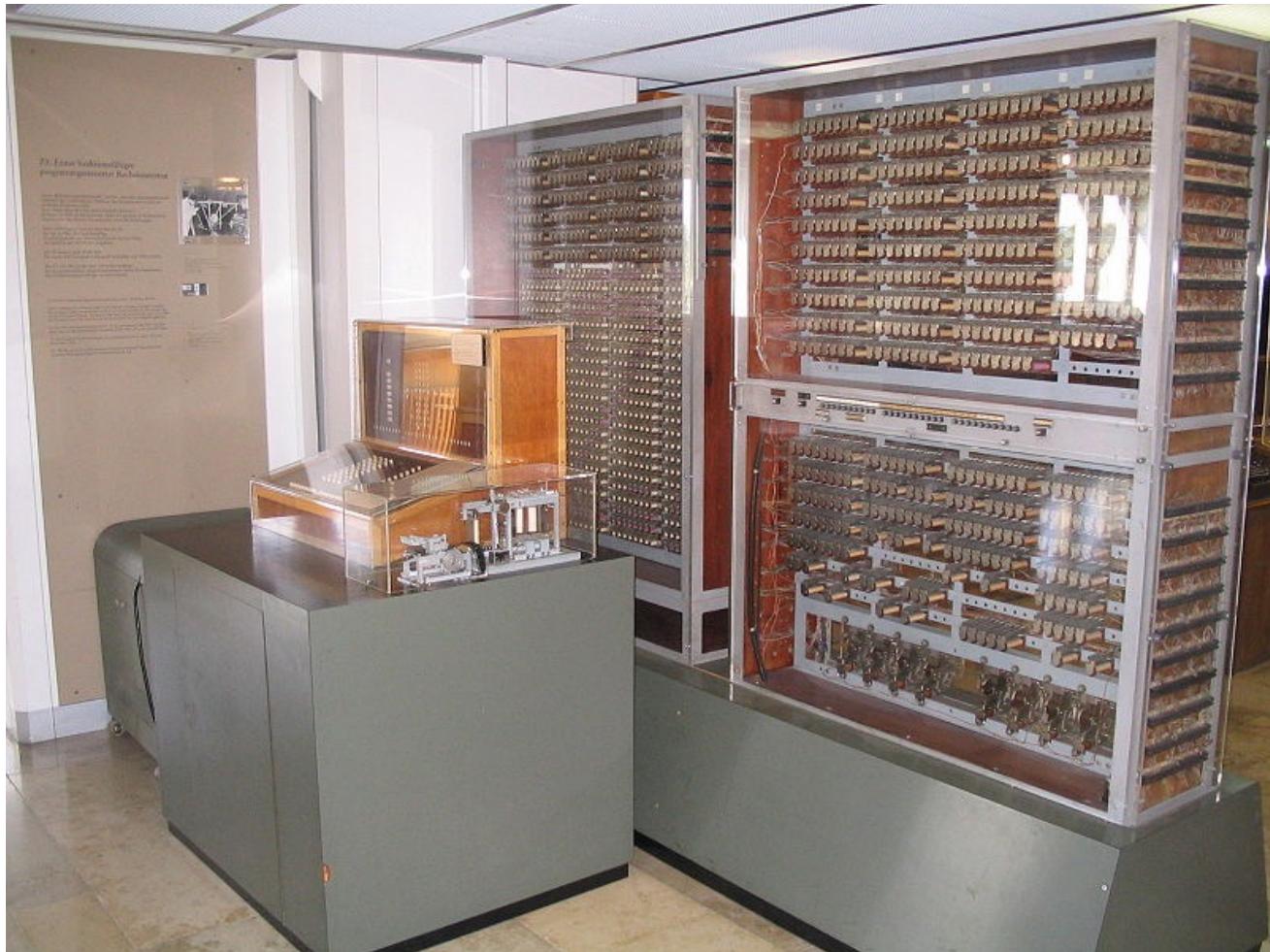


# Relay Computers

## Conrad Zuse (1910-1995)



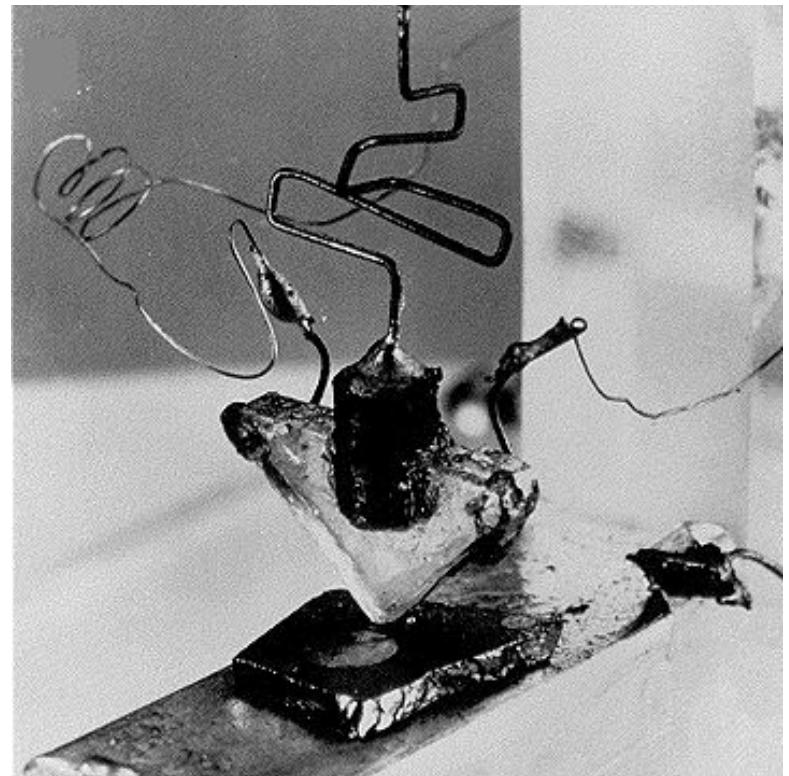
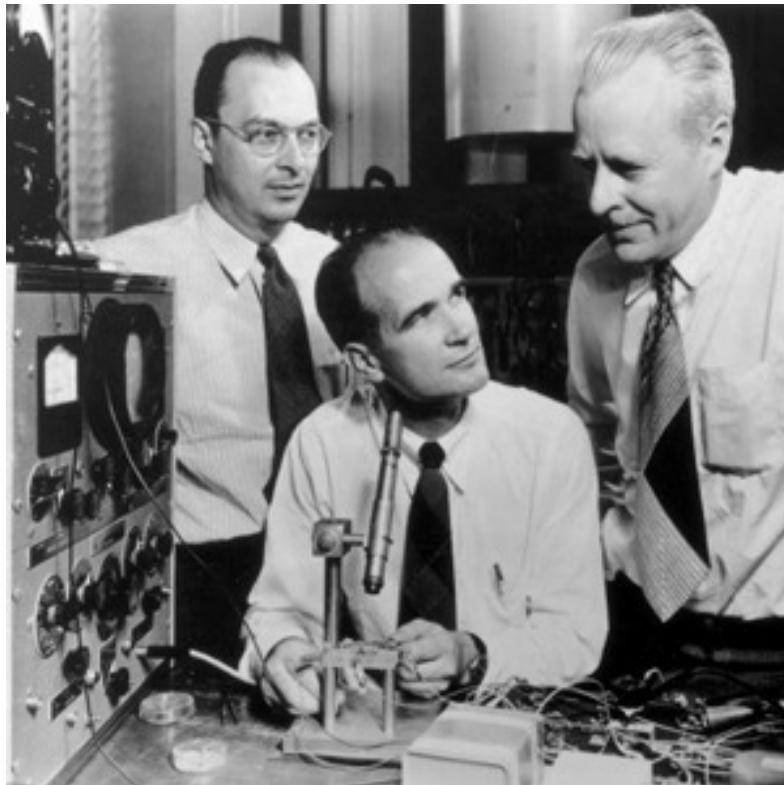
Z1 (1938)



Z3 (1941)

# Transistor, 1948

The thinker, the tinkerer, the visionary and the transistor  
John Bardeen, Walter Brattain, William Shockley  
Nobel Prize, 1956



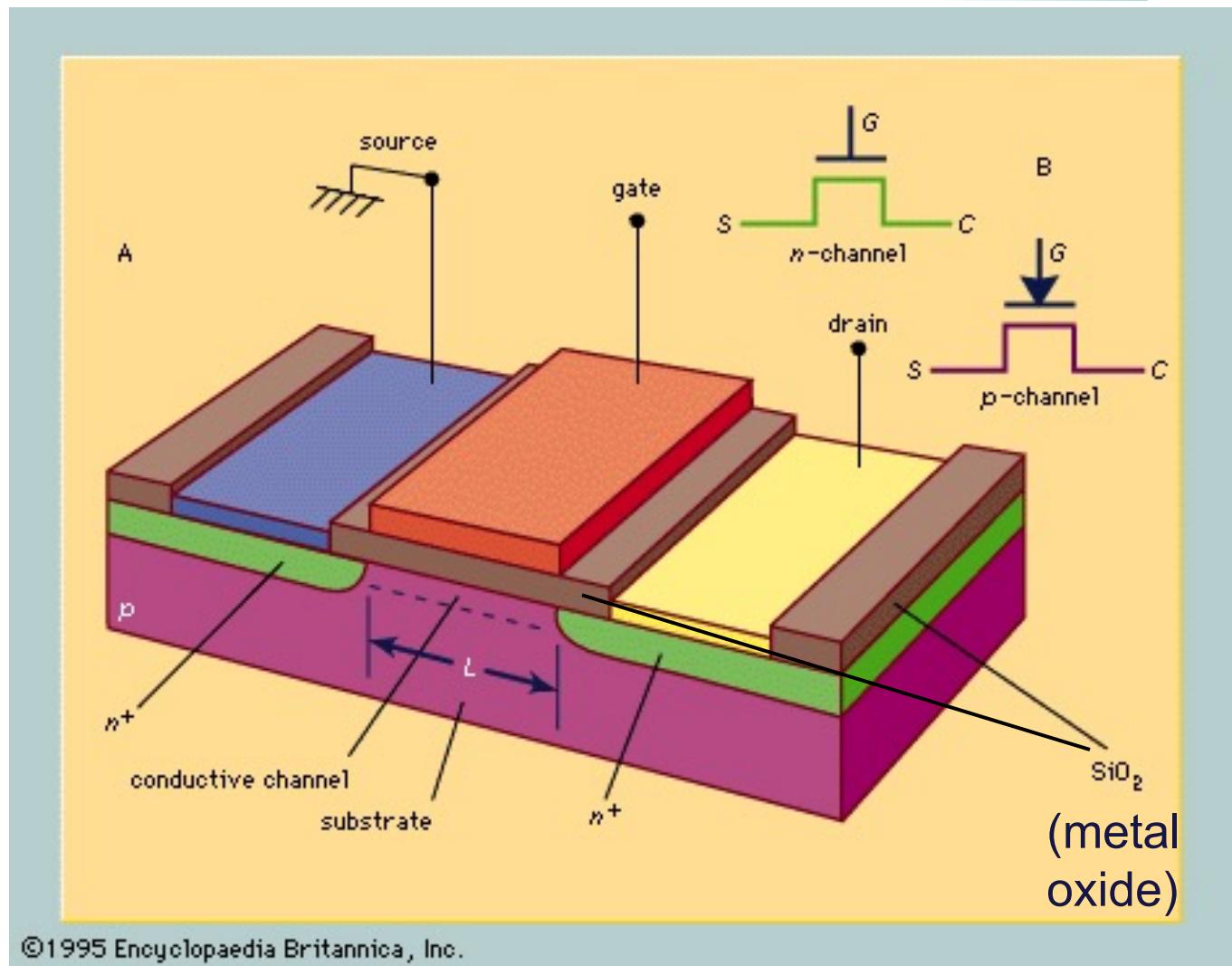
# Bell Laboratories, Murray Hill, New Jersey

---

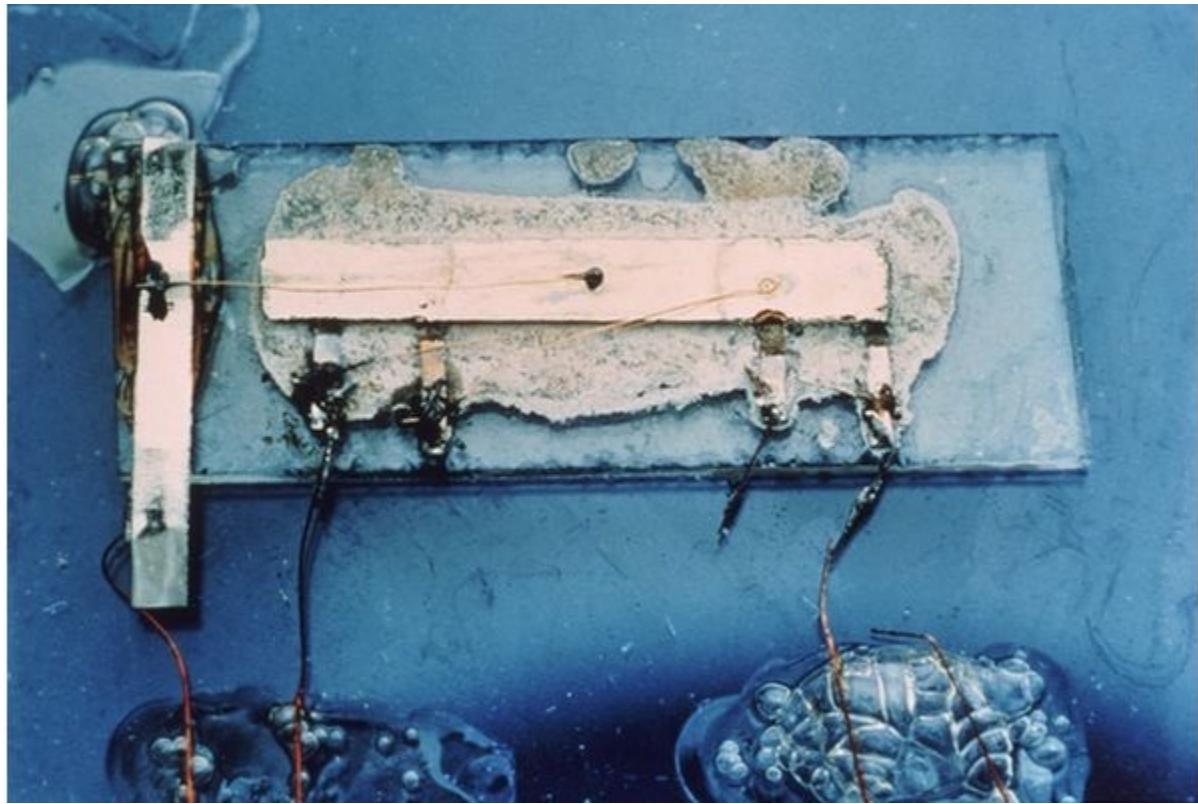
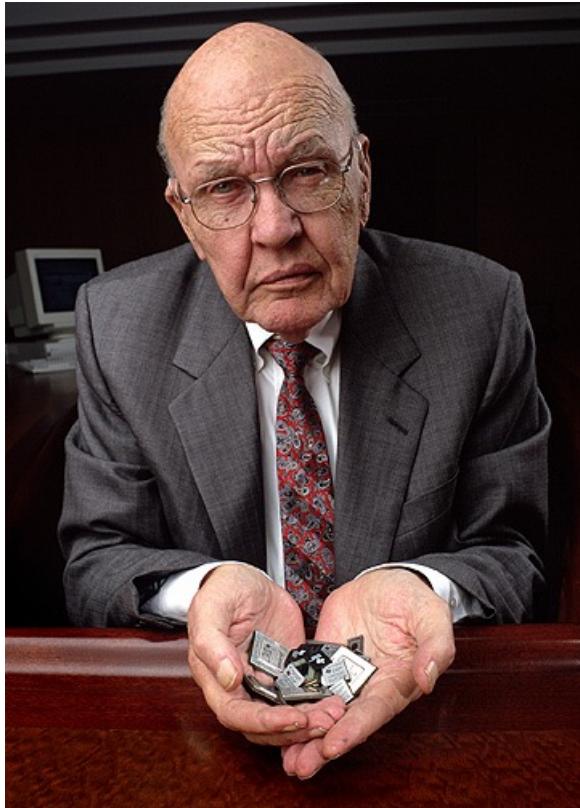


# Field Effect Transistor (FET)

a.k.a.  
metal oxide  
semiconductor  
(MOS) FET.



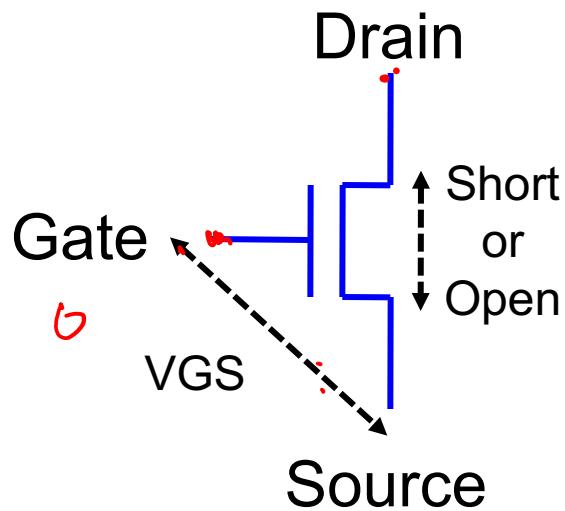
# Integrated Circuit (1958)



Jack Kilby (1923-2005), Nobel Prize, 2000

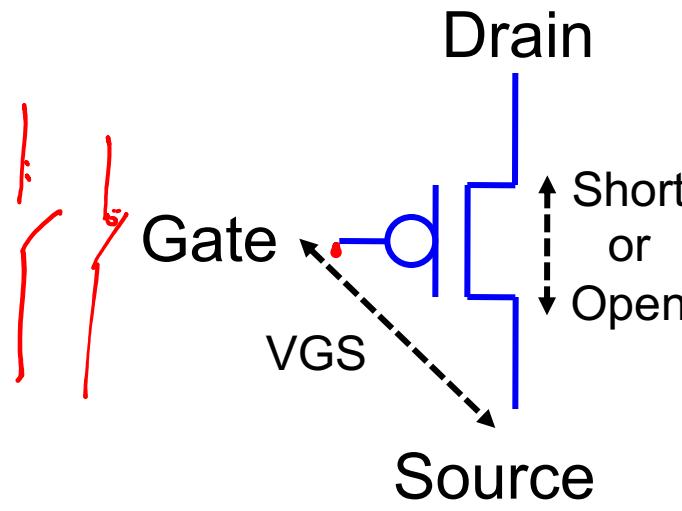


# MOSFET (Metal Oxide Semiconductor Field Effect Transistor)



NMOSFET

$V_{GS} = 0$ , open  
 $V_{GS} = \text{high}$ , short



PMOSFET

$V_{GS} = 0$ , short  
 $V_{GS} = \text{high}$ , open

## Reference:

R. C. Jaeger and T. N. Blalock, *Microelectronic Circuit Design*,  
Third Edition, McGraw Hill.



# NMOSFET NOT Gate (Early Design)

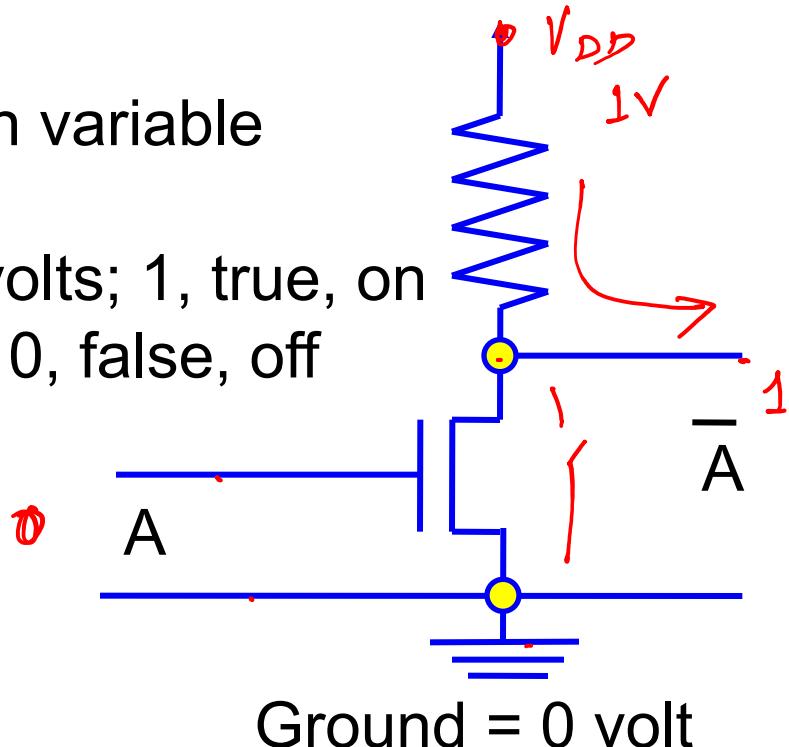
1.8

Power supply  
VDD volts  
w.r.t. ground

A: Boolean variable

A = VDD volts; 1, true, on

A = 0 volt; 0, false, off



Problem: When  $A = 1$ , current leakage causes power dissipation.

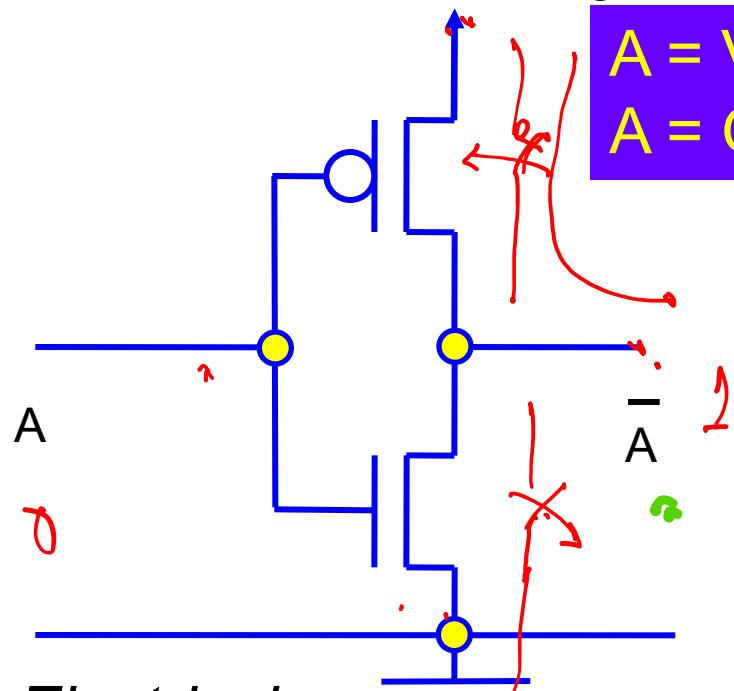
Solution: Complementary MOS design proposed by

F. M. Wanlass and C.-T. Sah,  
“Nanowatt Logic Using Field-  
Effect Metal-Oxide  
Semiconductor Triodes,”  
*International Solid State Circuits  
Conference Digest of Technical  
Papers*, Feb 20, 1963, pp. 32-33.

# CMOS NOT Gate (Modern Design)

Power supply

$V_{DD} = 1$  volt; voltage depends on technology.



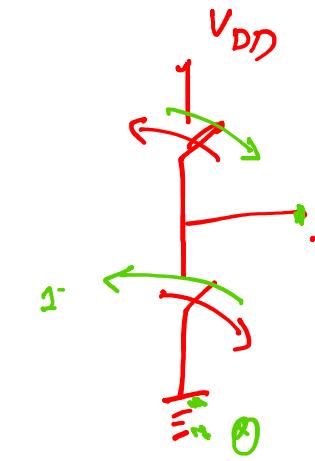
*Electrical  
Circuit*

GND  
Ground

$A = V_{DD} = 1$  volt is state “1”  
 $A = GND = 0$  volt is state “0”

Truth Table	
A	$\bar{A}$
0	1
1	0

*Boolean Function*



*Symbol*

$\overline{A}$



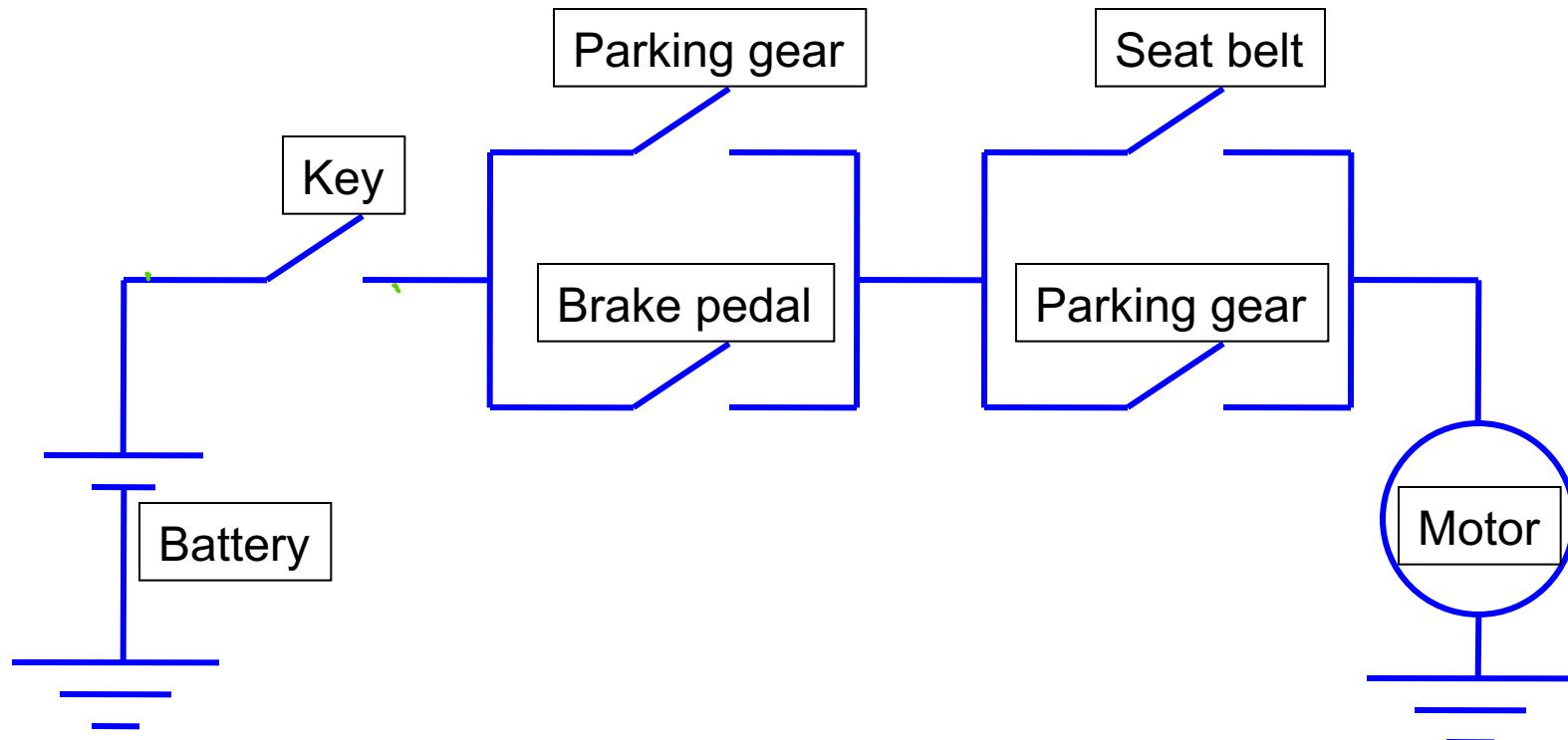
# Example: Automobile Ignition

---

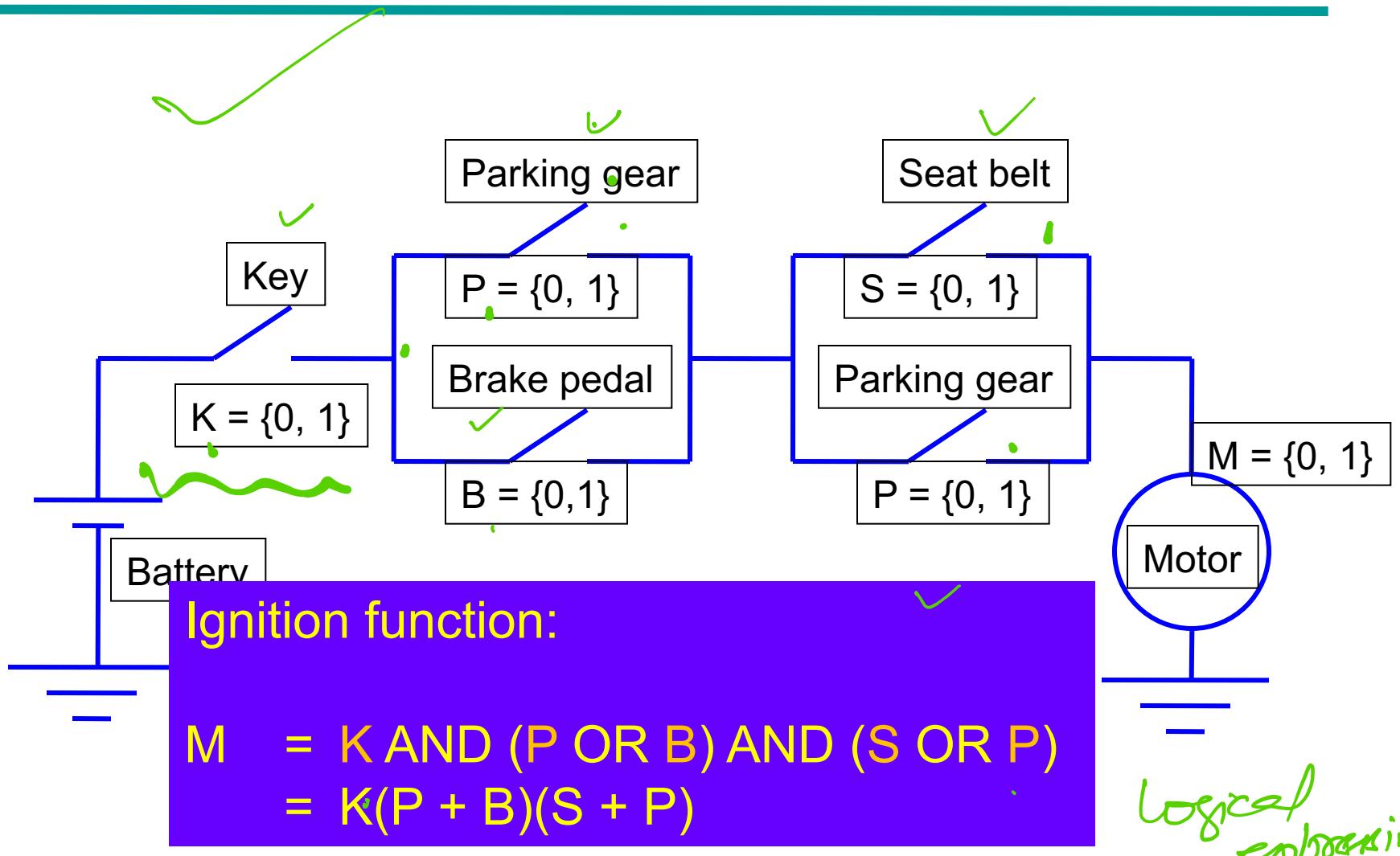
- Engine turns on when
- Ignition key is applied AND
  - Car is in parking gear OR
  - Brake pedal is on
- AND
  - Seat belt is fastened OR
  - Car is in parking gear



# Switching logic

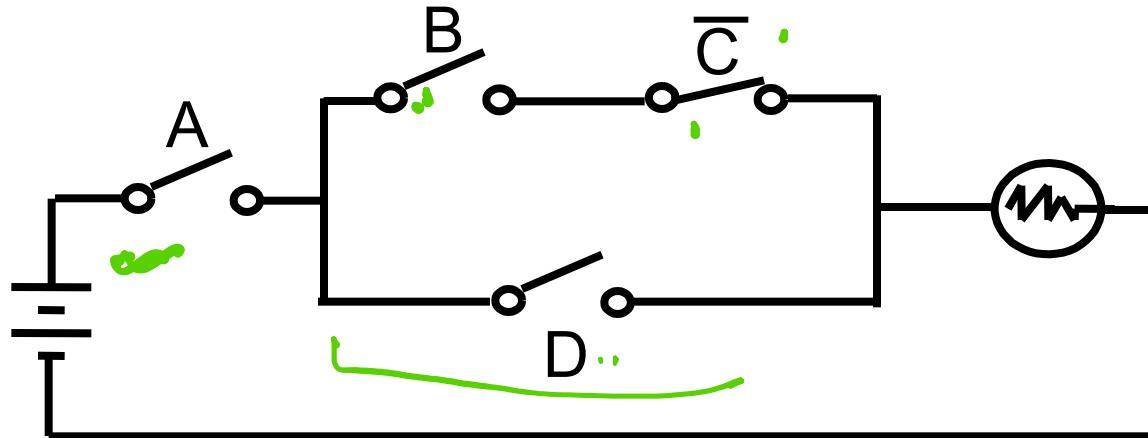


# Define Variables



# Logic Function Implementation

- Example: Logic Using Switches



- Light is on ( $L = 1$ ) for  $L(A, B, C, D) = \underline{A} ((B C') + D) = \underline{\underline{A}} B C' + \underline{\underline{A}} D$  and off ( $L = 0$ ), otherwise.
- Useful model for relay circuits and for CMOS gate circuits, the foundation of current digital logic technology

# Logic Expression

$$\begin{array}{r} 15+5+3 \\ \hline 23 \end{array}$$

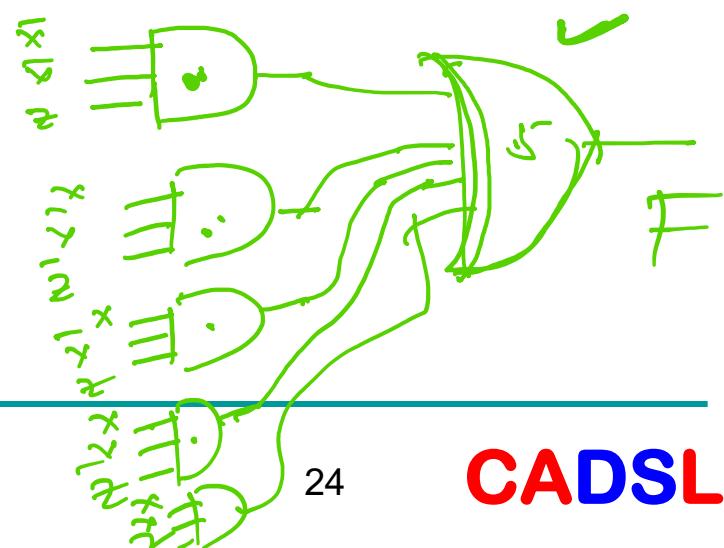
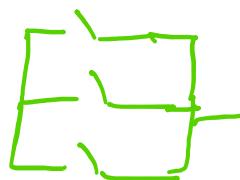
Truth Table

X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

Logic Expression

$$F = \underbrace{\bar{X} \cdot \bar{Y} \cdot Z}_{X \cdot Y \cdot \bar{Z}} + \underbrace{X \cdot \bar{Y} \cdot \bar{Z}}_{X \cdot \bar{Y} \cdot Z} + \underbrace{X \cdot \bar{Y} \cdot Z}_{X \cdot Y \cdot Z}$$

$$X \cdot Y \cdot \bar{Z} + X \cdot Y \cdot Z$$



# Logic Expressions

Truth Table

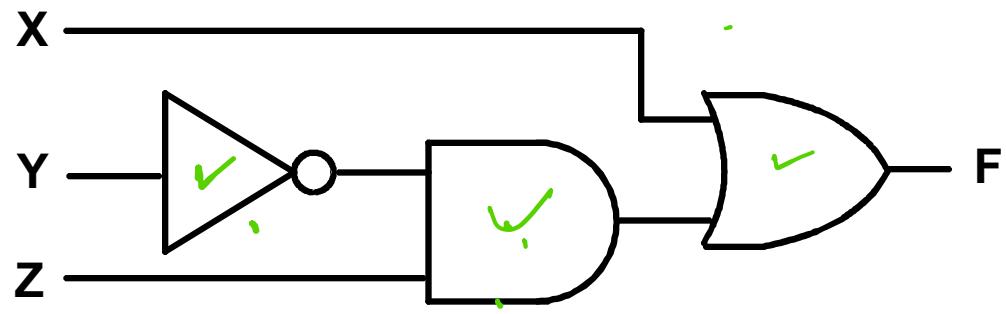
X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

Equation

$$F = X + \bar{Y} \cdot Z$$

SOP

Logic Diagram



$$2+2\times1=5$$



# Digital Logic Design

- Express input output relationship using Truth table
- Generate the logical expression by disjunction (OR) terms (conjunction of variables – AND) where system evaluates to true
- Replace all operators by the logic gates
- Replace logic gates by its transistor level circuit

Switches



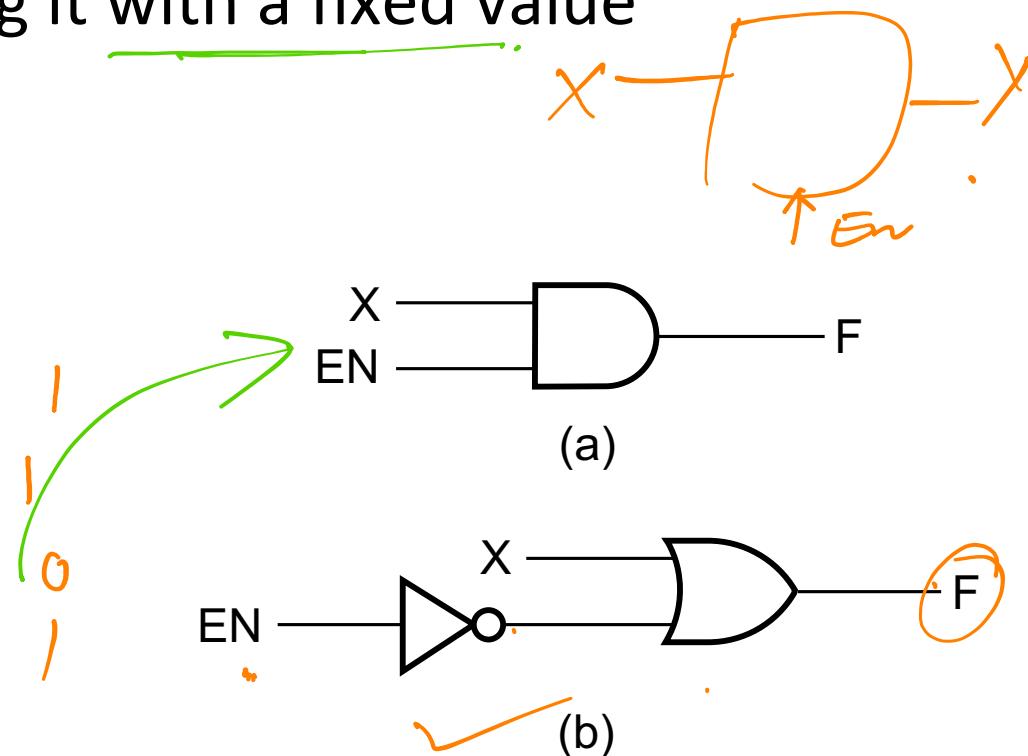
# Common Functions



# Enabling Function

- *Enabling* permits an input signal to pass through to an output
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value
- When disabled, 0 output
- When disabled, 1 output

EN	X	Y
1	0	0
1	1	0
0	0	0
0	1	1



# Decoding Function

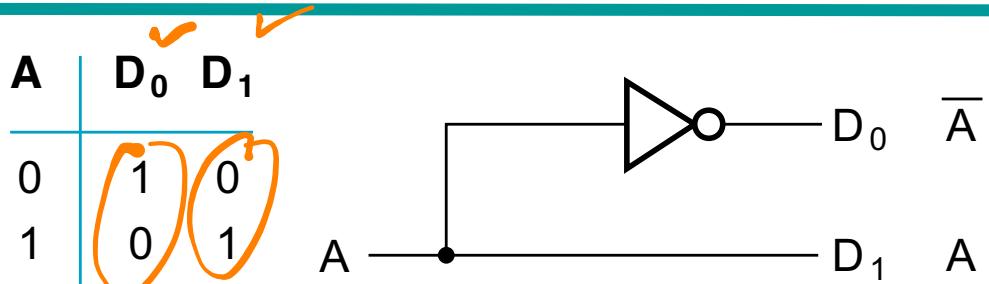
---

- Decoding - the
  - Conversion of  $n$ -bit input to  $m$ -bit output
  - Given  $n \leq m \leq 2^n$
- Circuits that perform decoding are called *decoders*
  - Called  $n$ -to- $m$  line decoders, where  $m \leq 2^n$ , and
  - Generate  $2^n$  (or fewer) 1's in output for the  $n$  input variables

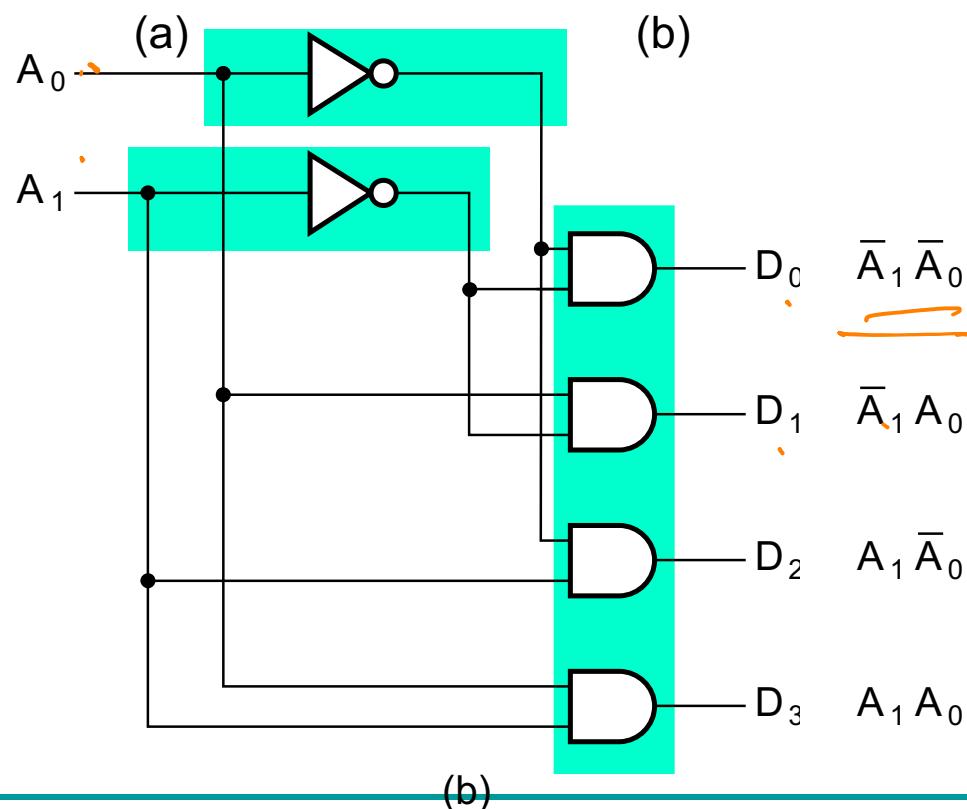
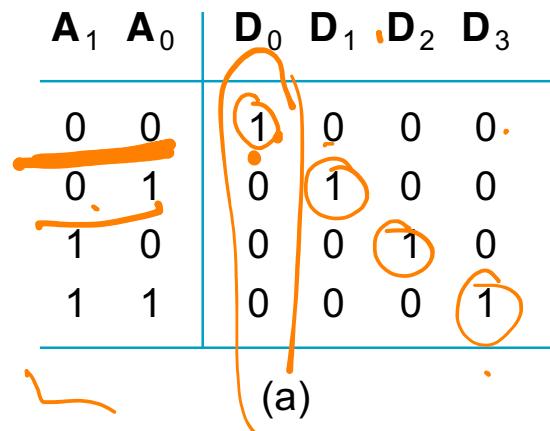


# Decoder

- 1-to-2-Line Decoder



- 2-to-4-Line Decoder



# Encoding Function

---

- Encoding - the opposite of decoding
  - Conversion of  $m$ -bit input to  $n$ -bit output
- Circuits that perform encoding are called *encoders*
  - An encoder has  $2^n$  (or fewer) input lines and  $n$  output lines which generate the binary code corresponding to the input values
  - Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corresponding to the position in which the 1 appears.



# Encoder

---

- A decimal-to-BCD encoder
    - Inputs: 10 bits corresponding to decimal digits 0 through 9, ( $D_0, \dots, D_9$ )
    - Outputs: 4 bits with BCD codes
    - Function: If input bit  $D_i$  is a 1, then the output ( $A_3, A_2, A_1, A_0$ ) is the BCD code for  $i$ ,
  - The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly.
- 



# Encoder

- Input  $D_i$  is a term in equation  $A_j$  if bit  $A_j$  is 1 in the binary value for  $i$ .

- Equations:

$$A_3 = D_8 + D_9$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

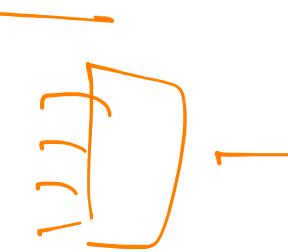
$D_0$	$D_1$	$D_2$	$D_3$	$D_4$	$D_5$	$D_6$	$D_7$	$D_8$	$D_9$
0000									
0001									
0010									
0011									
0100									
0101									
0110									
0111									
1000									
1001									



# Selection Function

---

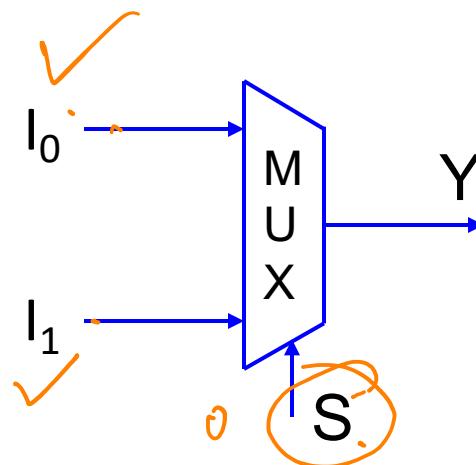
- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform Selecting have:
  - A set of information inputs from which the selection is made
  - A single output ✓
  - A set of control lines for making the selection
- Logic circuits that perform selecting are called **multiplexers**



# Multiplexers

---

- A **multiplexer** selects one input line and transfers it to output
  - $n$  control inputs ( $S_{n-1}, \dots S_0$ ) called *selection inputs*
  - $m \leq 2^n$  information inputs ( $I_{2^n-1}, \dots I_0$ )
  - output  $Y$



# 2-to-1-Line Multiplexer

- Since  $2 = 2^1$ ,  $n = 1$
- The single selection variable  $S$  has two values:
  - $S = 0$  selects input  $I_0$
  - $S = 1$  selects input  $I_1$
- Truth Table

$S$	$I_0$	$I_1$	$Y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

✓  $Y = I_0 \cdot \bar{S} + S \cdot I_1$

• Logic expression

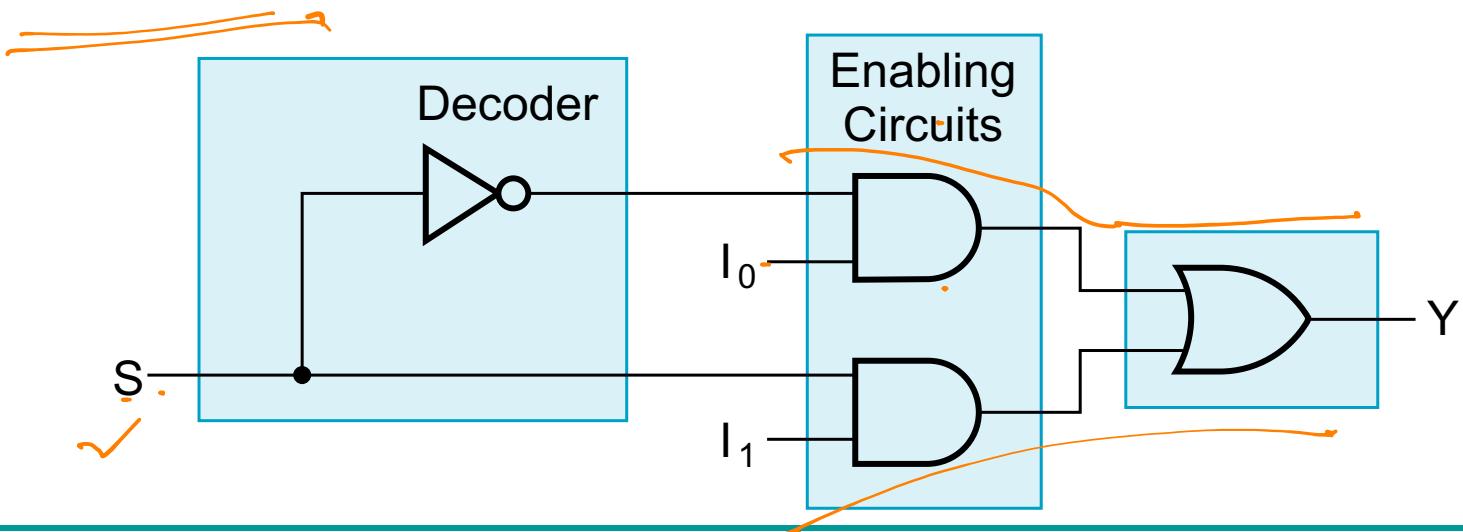
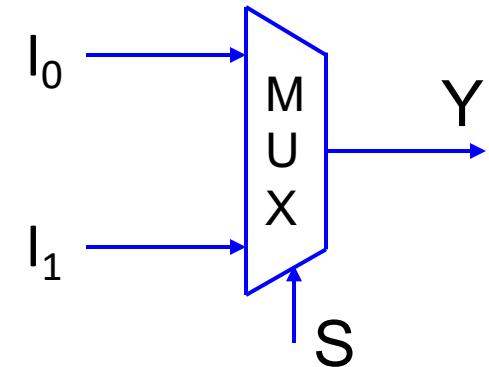
$$Y = \bar{S} \cdot I_0 \cdot \bar{I}_1 + \bar{S} \cdot I_0 \cdot I_1 + S \cdot \bar{I}_0 \cdot I_1 + S \cdot I_0 \cdot I_1$$



# 2-to-1-Line Multiplexer

- The single selection variable S has two values:
  - $S = 0$  selects input  $I_0$
  - $S = 1$  selects input  $I_1$
- The logic equation:

$$Y = I_0 \bar{S} + S \cdot I_1$$



# Thank You



# Logic: Implementation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

*CS-230: Digital Logic Design & Computer Architecture*

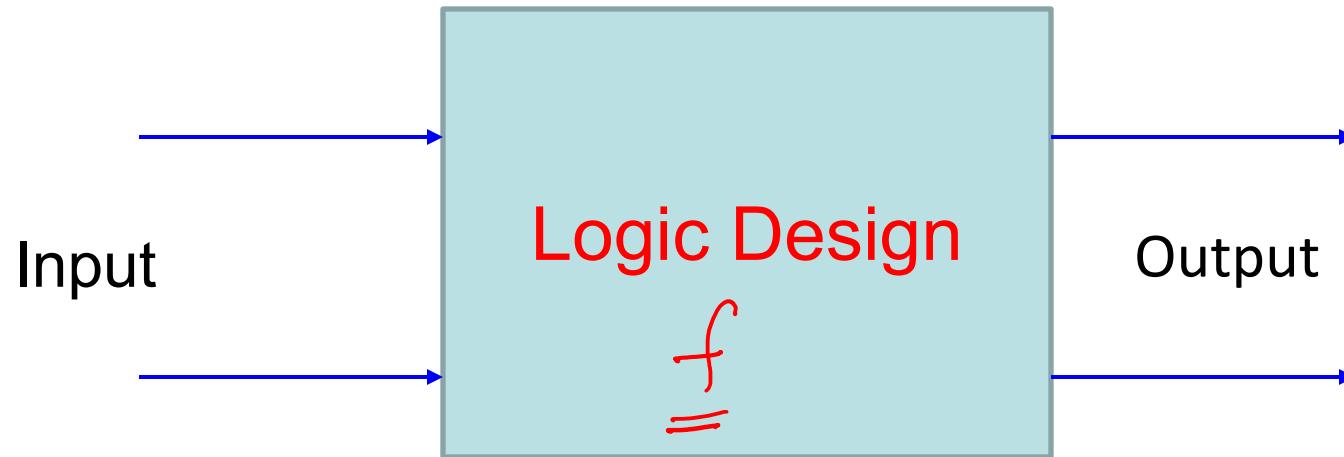
---



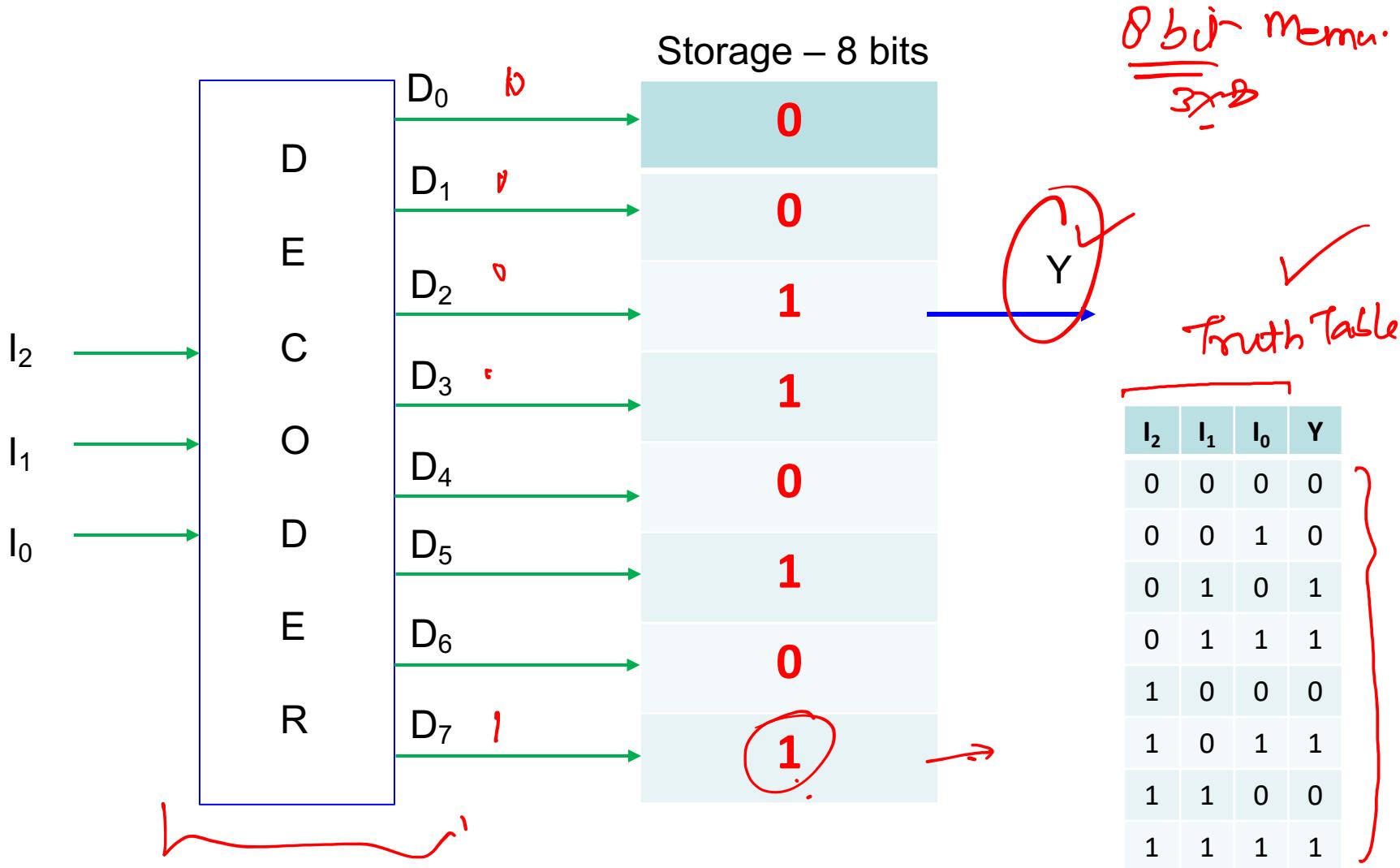
Lecture 4(13) January 2022

**CADSL**

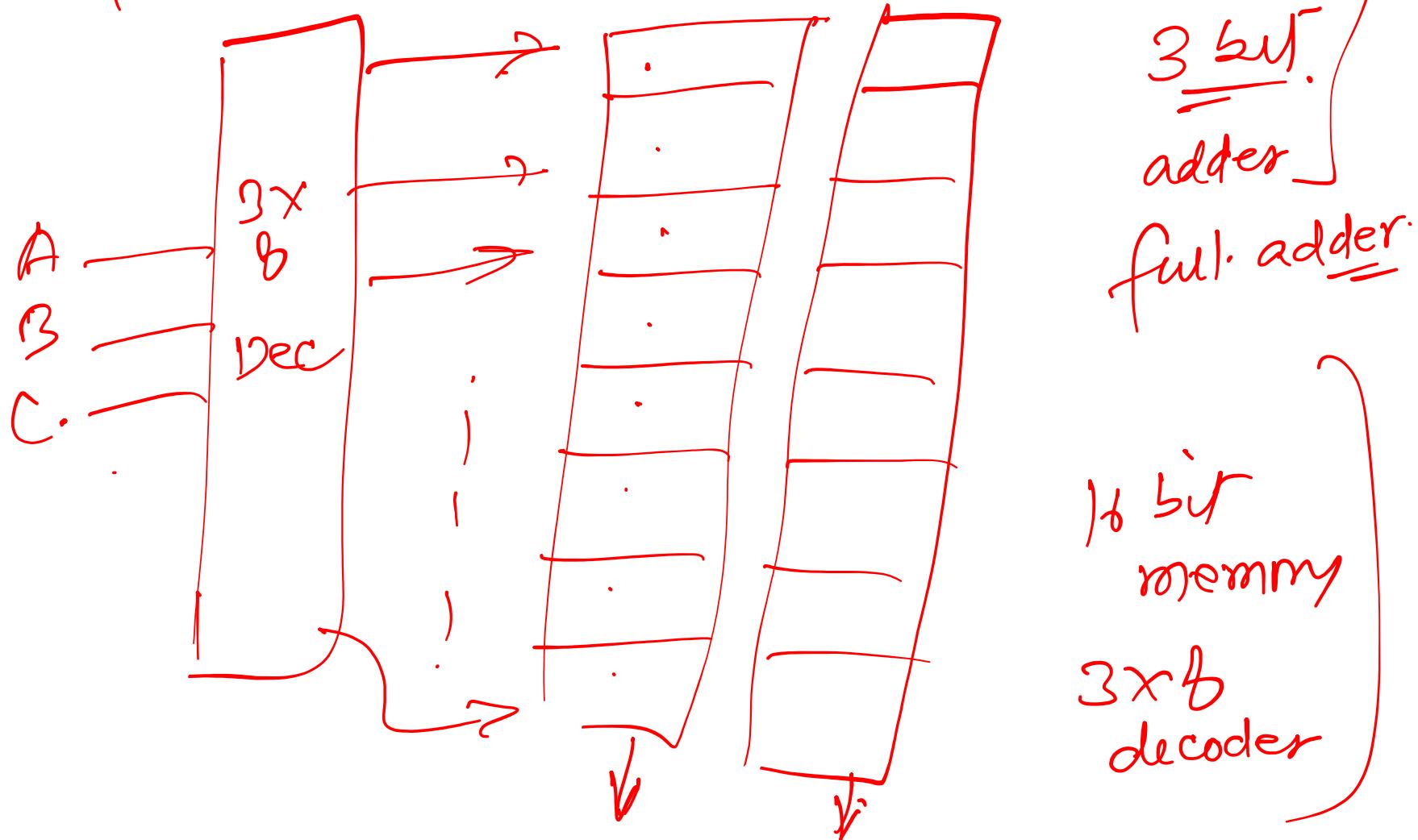
# Digital System



# Using Storage Elements



$10^{24}$   
 $10^6 \times 10^{24}$



# Optimization

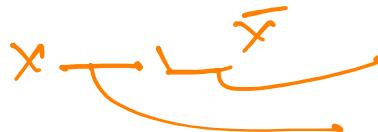


# Specification: Logic Function



Truth Table

X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

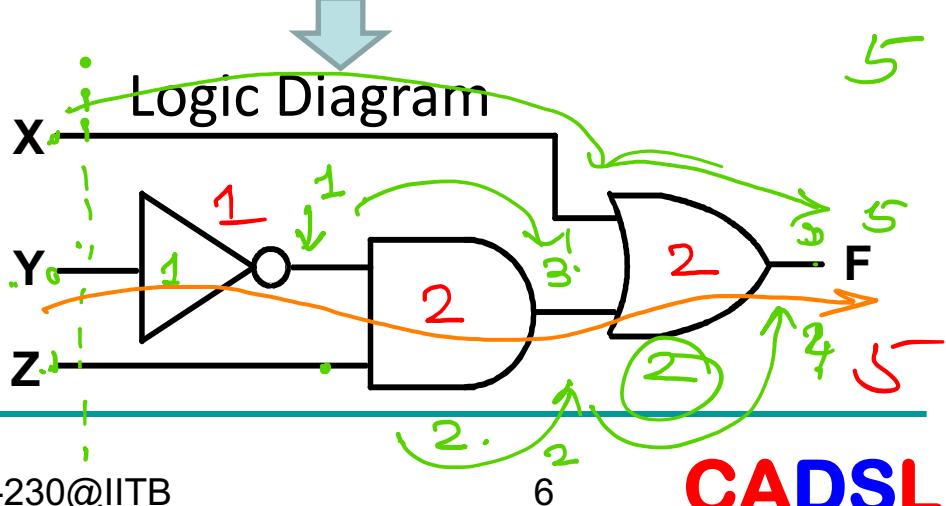


Logic Expression

$$F = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z + X \cdot Y \cdot Z + X \cdot Y \cdot \overline{Z}$$

↓

$$F = X + \overline{Y} \cdot Z$$



# Implementation

---

Logical expressions  $\Rightarrow$  not unique.

minimize logical expressions

Parameters.

cost.



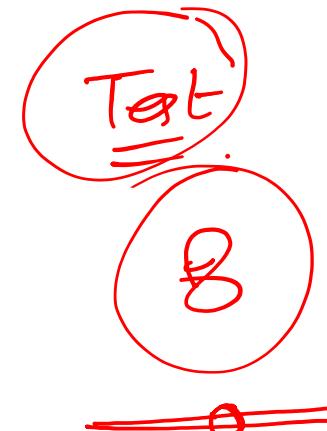
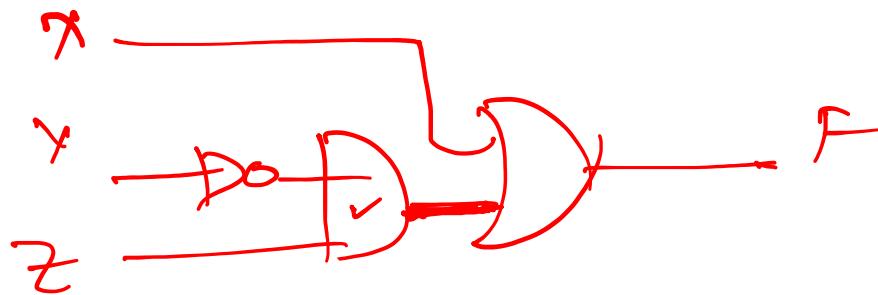
# Optimization Parameters

- ✓ Area: # Switches (Gates) cost 60's
- ✓ Performance (Delay): # Switches in series 70's longest path
- ✓ Power: # Switches
- ✓ Testability: Interconnect network 80's
- ✓ Security
- ✓ Intelligence P.P.T.A.S.I  
lowe Perfr



10<sup>10</sup> input/sec.

10.6 Hz.



↓

$$64+64 = 128$$

64 54)

Test Technique

5-6 sec/sec

few second

$$\frac{128}{10^{10}} \text{ sec} = \frac{120}{10^{10}} = \frac{(2^{10})^2}{10^{10}}$$

$$= \frac{10^{36}}{10^{10}} = \frac{10^{26} \text{ sec}}{60 \times 60 \times 24 \times 365 \times 1.00} = 10^{15} \text{ sec}$$

# ALGEBRA



# Algebra

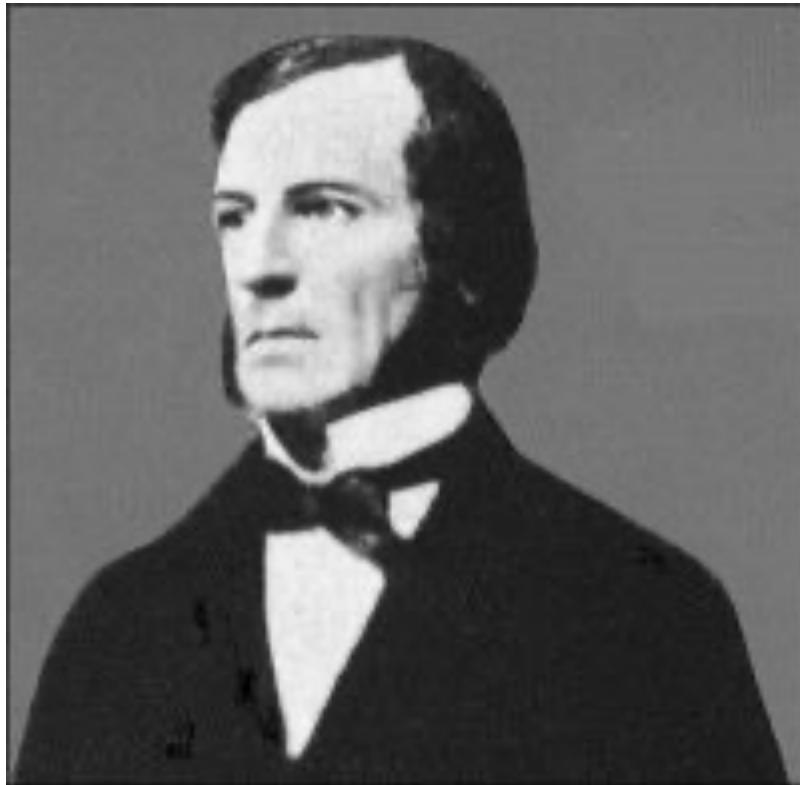
---

- Algebra is defined as
  1. Set of elements ✓
  2. Set of operators ✓
  3. Number of postulates
- A set of elements is any collection of objects having common properties ✓
$$S = \{a, b, c, d\}; a \in S, e \notin S$$
- A binary operator  $\underline{*}$  defined on a set  $S$  of elements is a rule that assigns each pair from  $S$  to a unique pair from  $S$ .  $\underline{a} * \underline{b} = c$



# George Boole (1815-1864) ✓

---



- Born, Lincoln,  
England
  - Professor of Math.,  
Queen's College,  
Cork, Ireland
  - Book, *The Laws of  
Thought*, 1853
- A red curly brace is drawn from the word 'England' to the word 'Thought'.

# Boolean Algebra

---

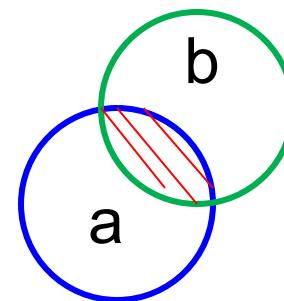
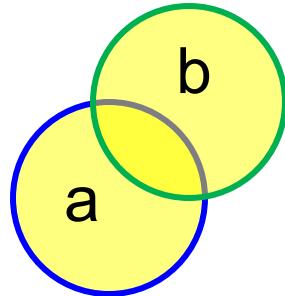
- Boolean Algebra is defined as
  1. Set of elements {0, 1} ✓
  2. Set of operators {+, ., ~}
  3. Number of postulates
- Boolean Algebra: 5-tuple
$$\{B, +, ., \sim, 0, 1\}$$
- Closure: If  $a$  and  $b$  are Boolean then  $(a \cdot b)$  and  $(a + b)$  are also Boolean



# Postulate 1: Commutativity

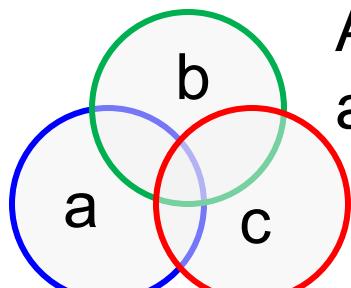
---

- Binary operators  $\underline{+}$  and  $\cdot$  are commutative.
- That is, for any elements  $a$  and  $b$  in  $B$ :
  - $a + b = b + a$
  - $a \cdot b = b \cdot a$



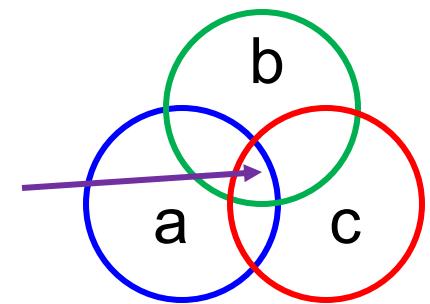
# Postulate 2: Associativity

- Binary operators  $+$  and  $\cdot$  are associative.
- That is, for any elements  $a$ ,  $b$  and  $c$  in  $B$ :
  - $a + (b + c) = (a + b) + c$
  - $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Example: EE department has three courses with student groups  $a$ ,  $b$  and  $c$



All EE students:  
 $a + (b + c)$

EE students in all EE  
courses:  $a \cdot (b \cdot c)$



# Postulate 3: Identity Elements

---

- There exist 0 and 1 elements in B, such that for every element a in B
  - $\underline{a} + 0 = a$  ✓
  - $\underline{a} \cdot 1 = a$  ✓
- Definitions:
  - 0 is the identity element for + operation
  - 1 is the identity element for · operation
- Remember, 0 and 1 here should not be misinterpreted as 0 and 1 of ordinary algebra.



# Postulate 5: Distributivity

---

- Binary operator  $\underline{+}$  is distributive over  $\cdot$  and  $\cdot$  is distributive over  $\underline{+}$ .
- That is, for any elements  $a$ ,  $b$  and  $c$  in  $K$ :
  - $a + (b \cdot c) = (a + b) \cdot (a + c)$
  - $\underline{a} \cdot (b + c) = (\underline{a} \cdot b) + (\underline{a} \cdot c)$
- Remember dot ( $\cdot$ ) operation is performed before  $+$  operation:

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



# Postulate 6: Complement

---

- A unary operation, *complementation*, exists for every element of B.
- That is, for any element a in B:

$$\begin{array}{l} a + \bar{a} = 1 \\ a \cdot \bar{a} = 0 \end{array}$$

- Where, 1 is identity element for  $\underline{\cdot}$   
0 is identity element for  $+$



# The Duality Principle

---

- Each postulate of Boolean algebra contains a pair of expressions or equations such that **one is transformed into the other** and vice-versa by interchanging the operators,  $\overbrace{+ \leftrightarrow \cdot}$ , and identity elements,  $0 \leftrightarrow 1$ .
- The two expressions are called the duals of each other.

$$\begin{array}{c} + \leftrightarrow \cdot \\ \overbrace{\vee \quad \wedge} \\ OR \quad AND \end{array}$$



# Theorems 1

---

Theorem 1: Idempotency

For all elements  $a$  in  $B$ :  $\underline{a + a = a}; \underline{a \cdot a = a}$ .

Theorem 2: Existance of Null Element

- $\underline{\underline{a + 1 = 1}}$ , for  $+$  operator.
- $\underline{\underline{a \cdot 0 = 0}}$ , for  $\cdot$  operator.

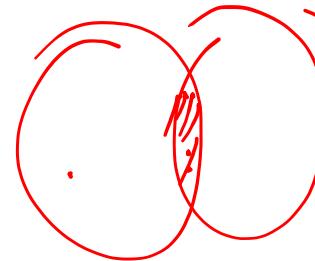
Theorem 3: Involution holds

- $\bar{\bar{a}} = a$



# Theorem 4: Absorption ✓

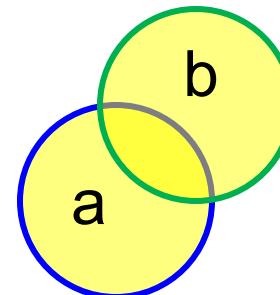
- $a + \underline{a.b} = \underline{\underline{a}}$  ✓
- $a.(a + b) = a$  ✓



• Proof:

$$\begin{aligned} a + a.b &= a.1 + a.b \quad (\text{identity element}) \\ &= a.(1 + b) \quad (\text{distributivity}) \\ &= a.1 \quad (\text{Theorem 2}) \\ &= a \quad (\text{identity element}) \end{aligned}$$

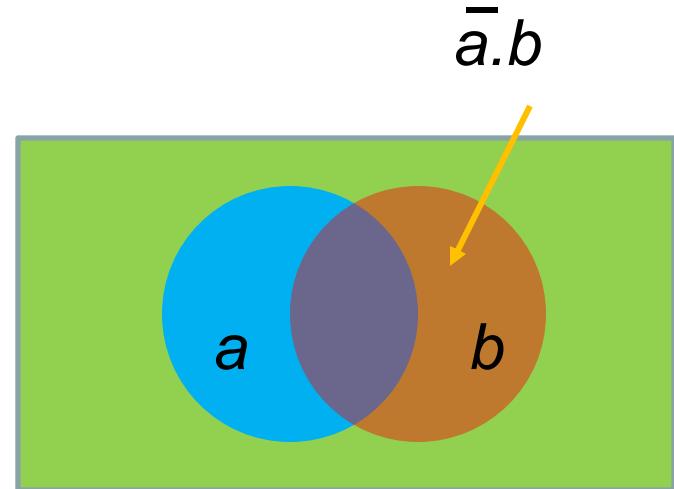
*Similar proof for  $a(a + b) = a$ .*



# Theorems: Adsorption & Uniting

- Theorem 5: Adsorption

$$a + \bar{a}b \equiv a + b$$
$$a(\bar{a} + b) \equiv ab$$



- Theorem 6: Uniting ✓

$$ab + a\bar{b} = a$$
$$(a + b)(a + \bar{b}) \equiv a$$





# Theorem 7: DeMorgan's Theorem

---

- $\overline{a + b} = \bar{a} \cdot \bar{b}, \quad \forall a, b \in B$
- $\overline{a \cdot b} = \bar{a} + \bar{b}, \quad \forall a, b \in B$



1806 - 1871

*Generalization of DeMorgan's Theorem:*

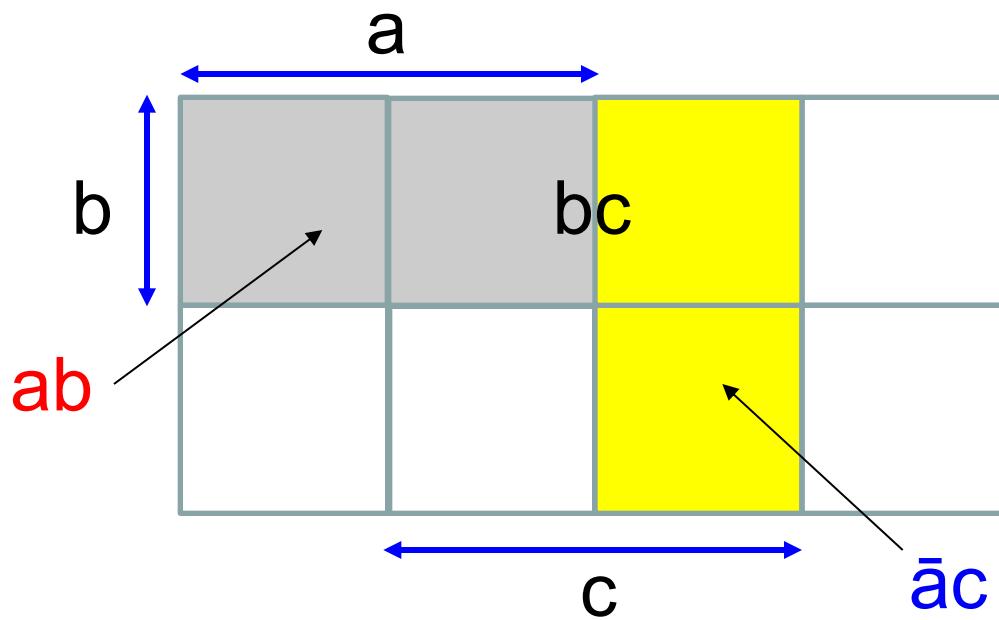
$$\overline{\overline{a + b + \cdots + z}} = \bar{a} \cdot \bar{b} \cdot \cdots \cdot \bar{z}$$
$$\overline{a \cdot b \cdot \cdots \cdot z} = \bar{a} + \bar{b} + \cdots + \bar{z}$$



# Theorem 8: Consensus

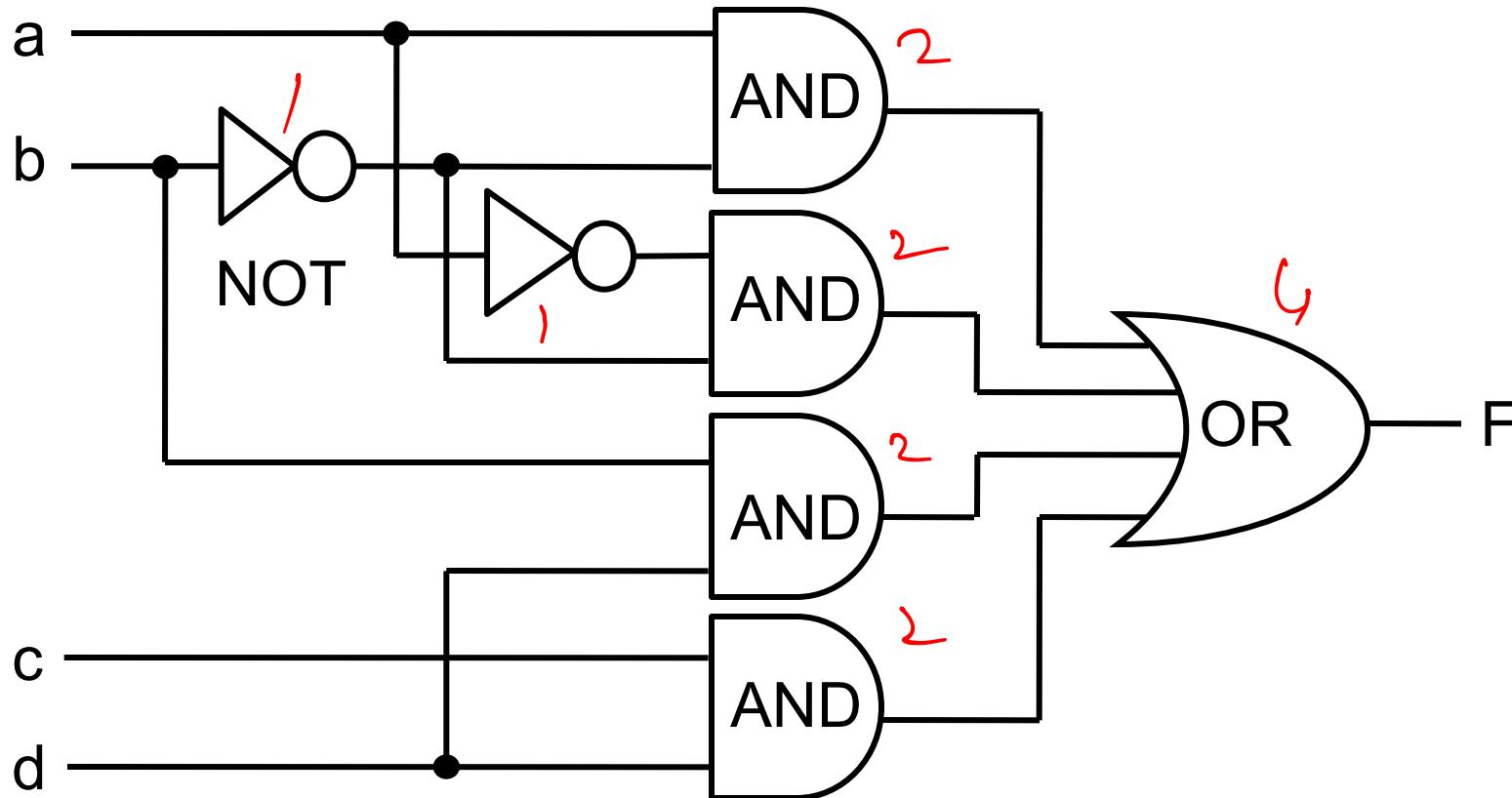
$$\underline{ab} + \underline{\bar{a}\bar{c}} + bc \equiv ab + \bar{a}\bar{c}$$
$$(a+b)(\bar{a}+c)(b+\bar{c}) \equiv (a+b)(\bar{a}+c)$$

Lecture 586



# Understanding Minimization

- Logic function:  $F = \overline{a}\overline{b} + \overline{a}\overline{b} + bd + cd$



# Logic Minimization

- Reducing products:

$$\begin{aligned} F &= \cancel{a\bar{b}} + \cancel{\bar{a}\bar{b}} + \cancel{bd} + \cancel{cd} \\ &= \cancel{b}(a + \bar{a}) + \cancel{bd} + \cancel{cd} \\ &= \cancel{b}1 + \cancel{bd} + \cancel{cd} \\ &= \cancel{b}(c + \bar{c}) + \cancel{bd} + \cancel{cd} \\ &\cancel{=} \cancel{bd} + \cancel{\bar{bc}} + \cancel{cd} + \cancel{\bar{bc}} \\ &= bd + \cancel{\bar{bc}} + \cancel{\bar{bc}} \\ &= bd + \cancel{b}(\cancel{c} + \cancel{\bar{c}}) \\ &= bd + \cancel{b}1 \end{aligned}$$

$$F = \bar{b} + d \quad \checkmark$$

Distributivity

Complementation

Identity

Distributivity

Consensus theorem ✓

Distributivity

Complement, identity

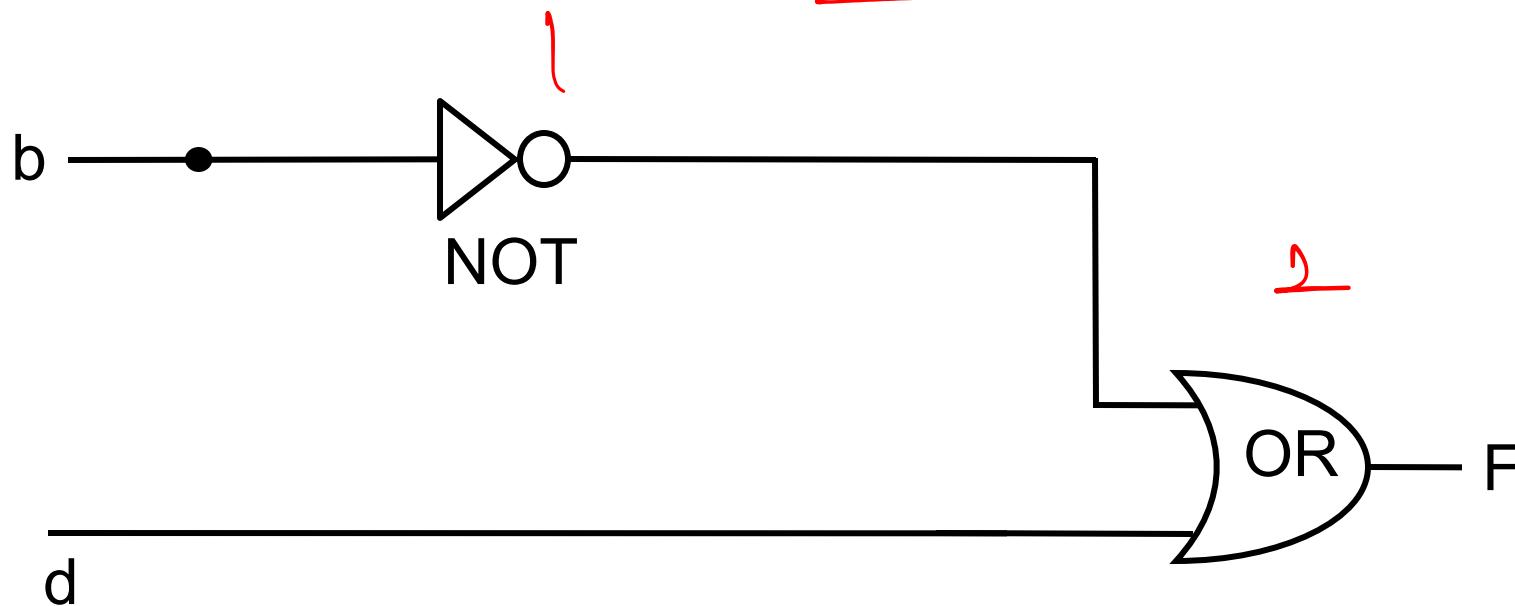
Adsorption



# Logic Minimization

- Minimized expression:

$$F = \bar{b} + d$$



# Thank You



# Logic: Implementation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

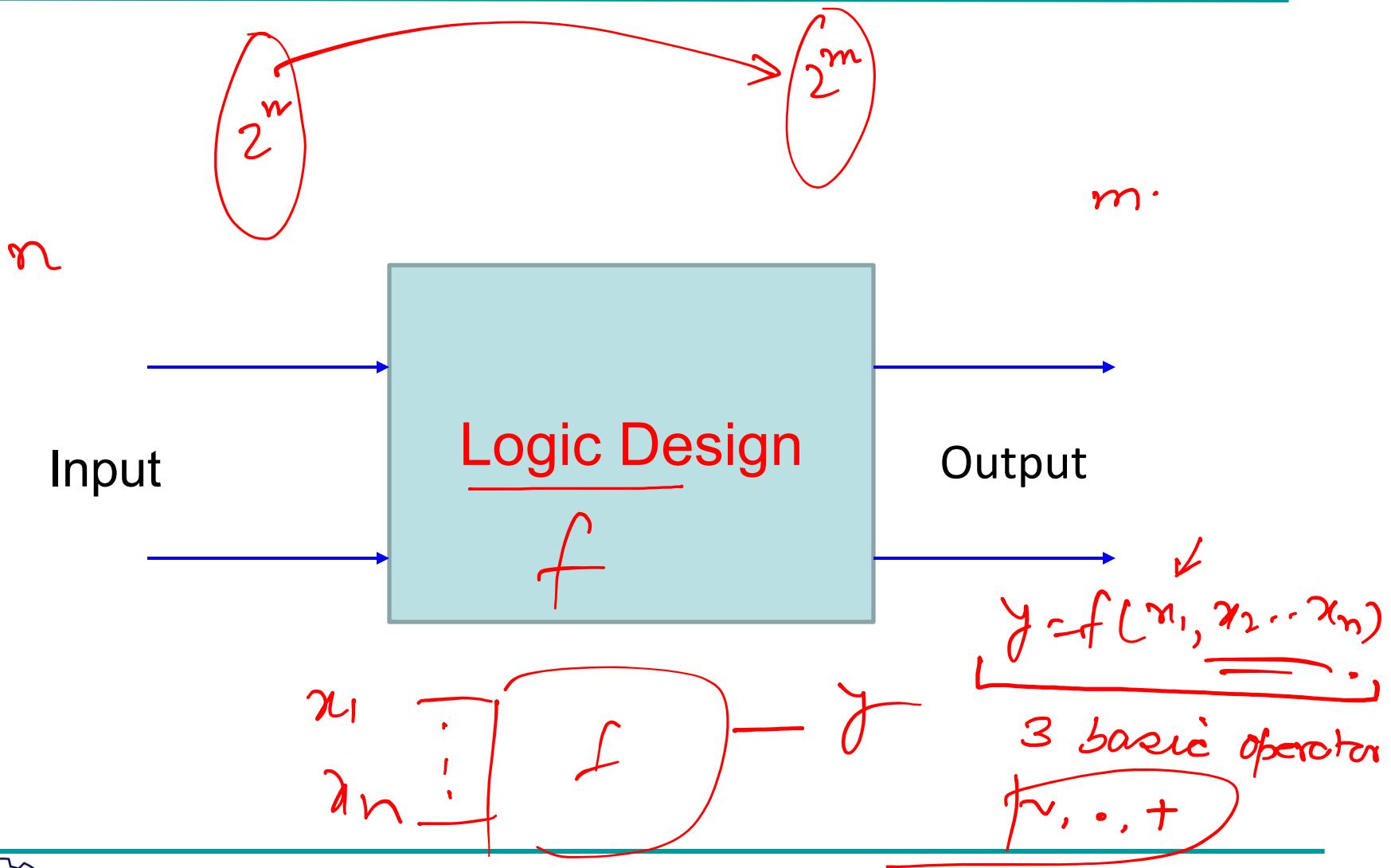
---



Lecture 6 (17 January 2022)

**CADSL**

# Digital System



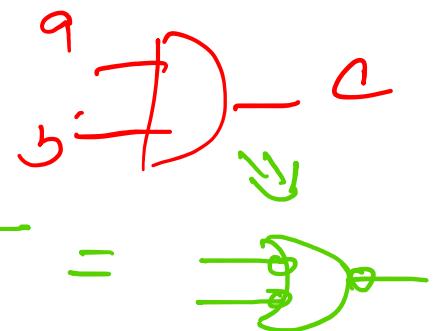
# Minimum Operator Set

- Minimum number of operators
- $\{\sim, (+ \text{ or } .)\} / \{\neg, (\wedge \text{ or } \vee)\}$

Boolean  
Algebra  
DeMorgan's.

$$C = a \cdot b = \overline{(\overline{a} \cdot \overline{b})} \Rightarrow \overline{\overline{a} + \overline{b}}$$

$$\overline{\overline{a} + \overline{b}}$$

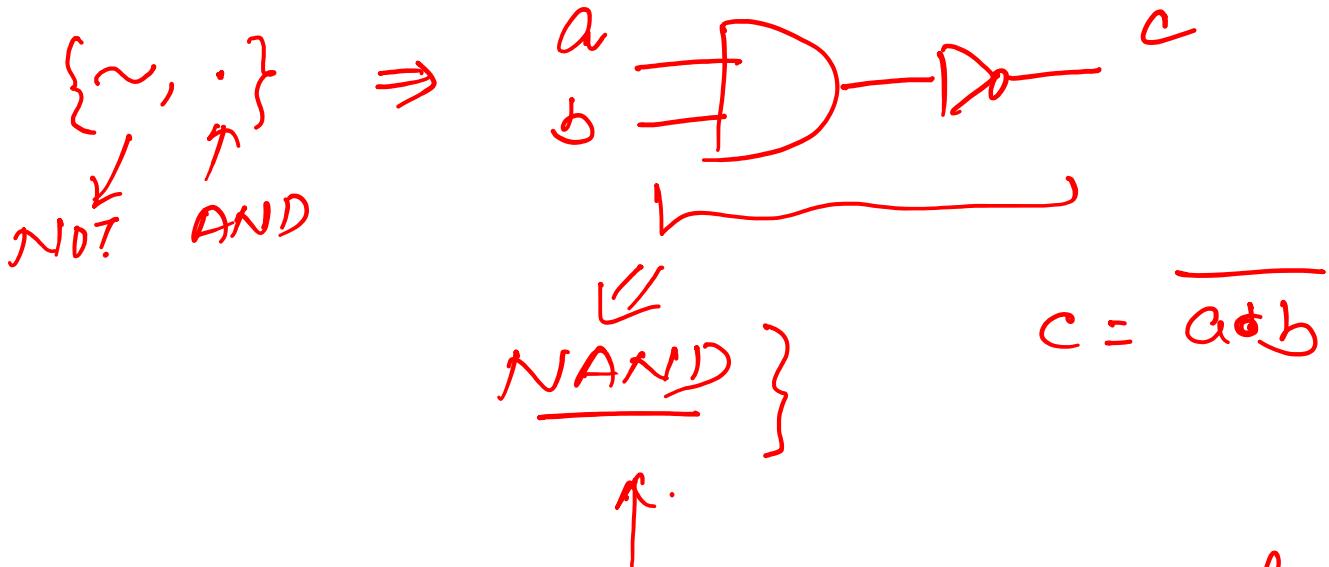


$$C = a + b = \overline{\overline{a} \cdot \overline{b}} = \overline{\overline{a} \cdot \overline{b}} = \overline{\overline{a} \cdot \overline{b}} = \overline{\overline{a} \cdot \overline{b}} = \overline{\overline{a} \cdot \overline{b}}$$

$\{\sim, \cdot\}$

$\{\sim, +\}$

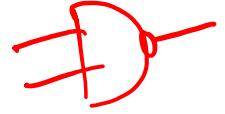




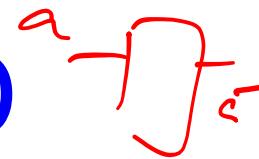
Can we represent any logical functions using only this operators

$$\{\sim, \cdot, +\}$$





# Universal Operator: NAND



- NAND: Composite operator (AND and NOT)

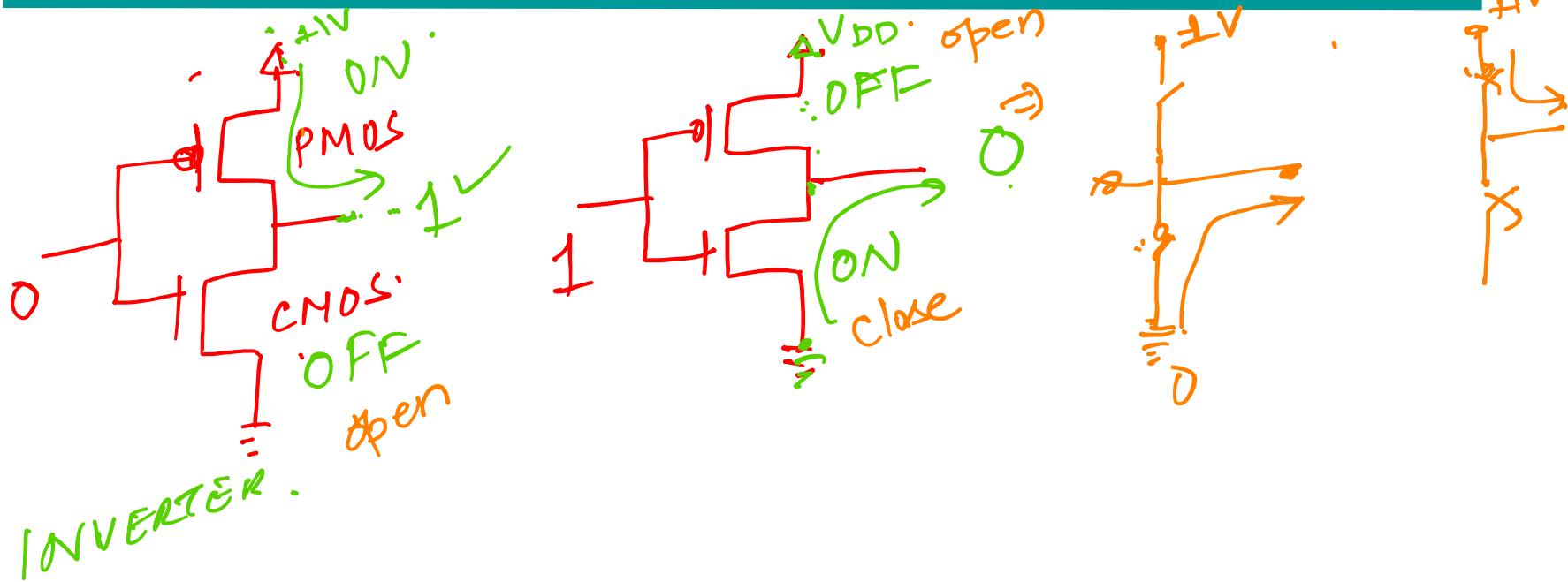
$$\bullet \bar{a} = \overline{\underline{a} \cdot \underline{a}} \Rightarrow \begin{array}{c} a \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ \bar{a} \end{array}$$

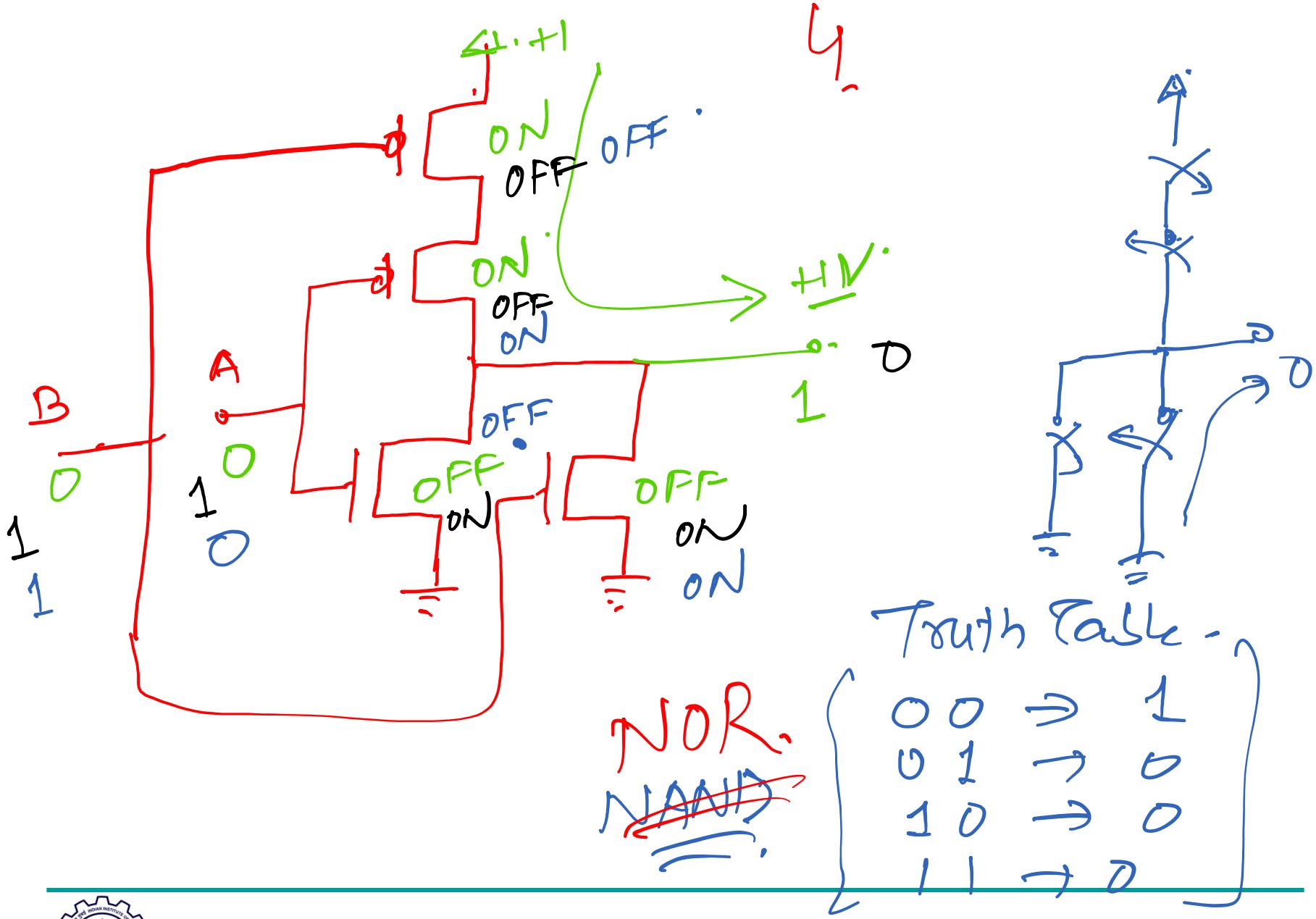
$$\bullet a \cdot b = \overline{\underline{\underline{a} \cdot \underline{\underline{b}}}} = \overline{\underline{(a \cdot b)} \cdot \underline{(a \cdot b)}} \quad \begin{array}{c} a \cdot b \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ a \cdot b \end{array}$$

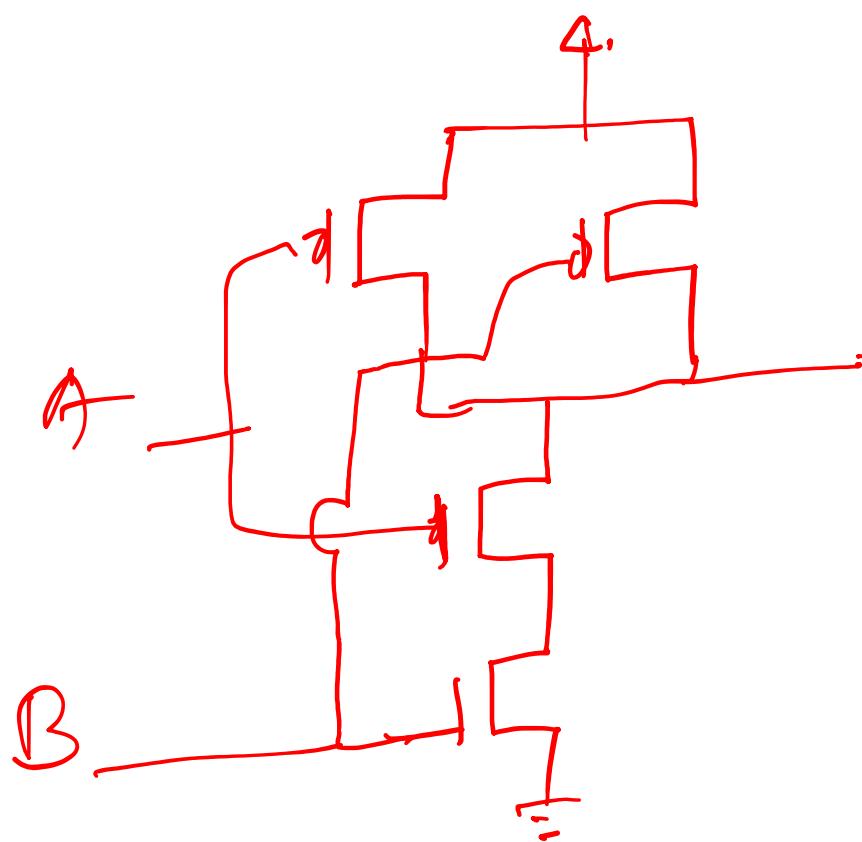
$$\bullet a + b = \overline{\underline{\bar{a} \cdot \bar{b}}} = \overline{\underline{\underline{(a \cdot a)} \cdot \underline{\underline{(b \cdot b)}}}} \quad \begin{array}{c} a + b \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ \text{NAND gate} \\ \text{---} \\ a + b \end{array}$$



# Universal Operator: NAND







Y

2n

NAND

AND ·  
NAND + NOT

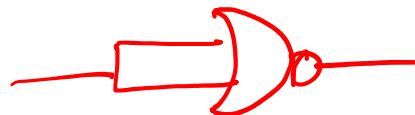




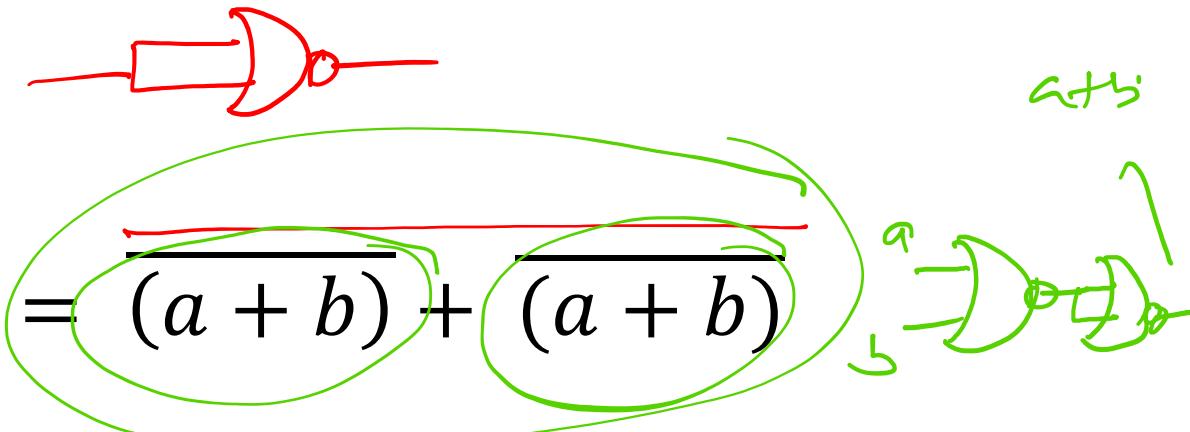
# Universal Operator: NOR $\{\sim, +\}$

- NOR: Composite operator (OR and NOT)

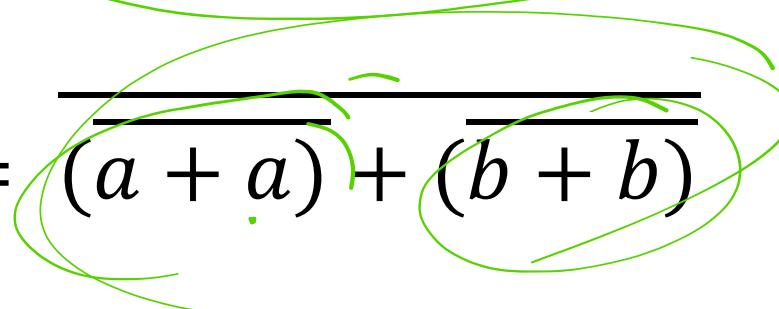
- $\overline{a} = \overline{a + a}$



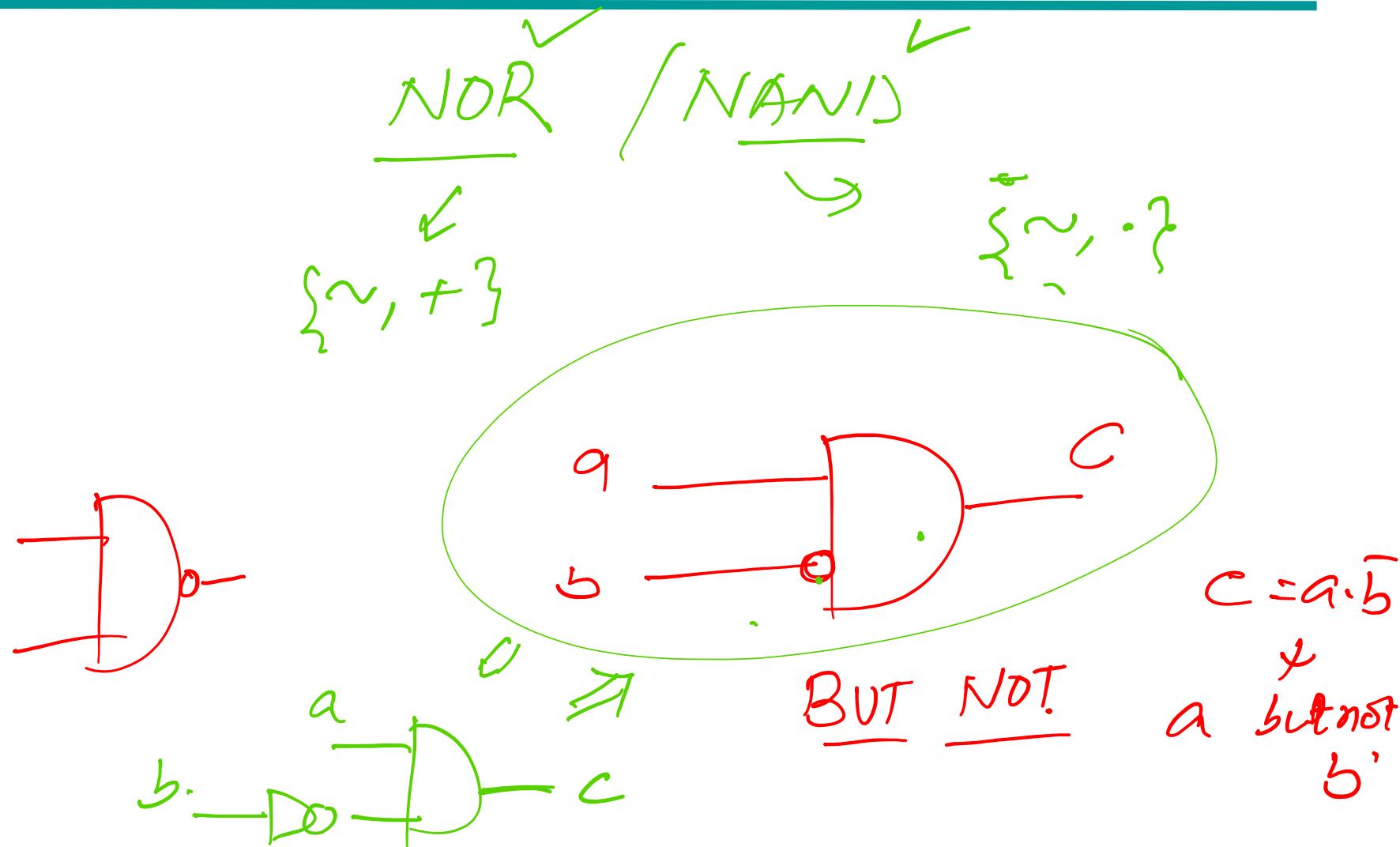
- $a + b = \overline{\overline{a + b}} = \overline{(a + b)} + \overline{(a + b)}$



- $a \cdot b = \overline{\overline{a}} + \overline{\overline{b}} = \overline{\overline{(a + a)}} + \overline{\overline{(b + b)}}$



# Universal Operator: NOR



# Complementing Functions

- Use DeMorgan's Theorem to complement a function:



1. Interchange AND and OR operators



2. Complement each constant value and literal

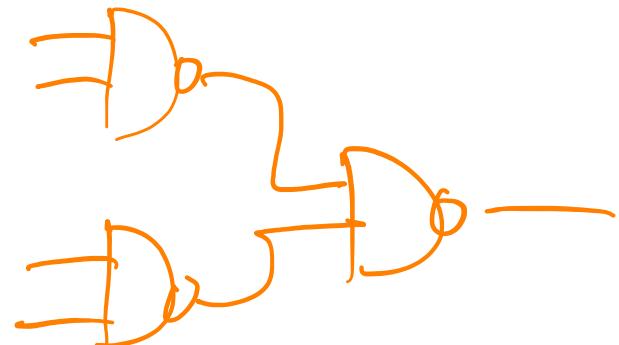
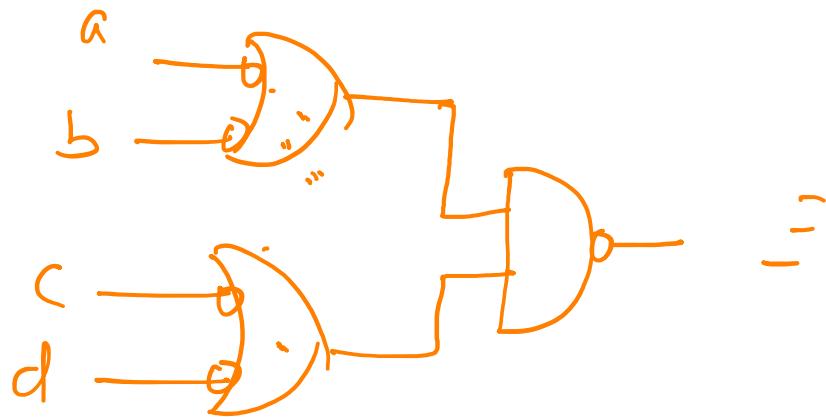
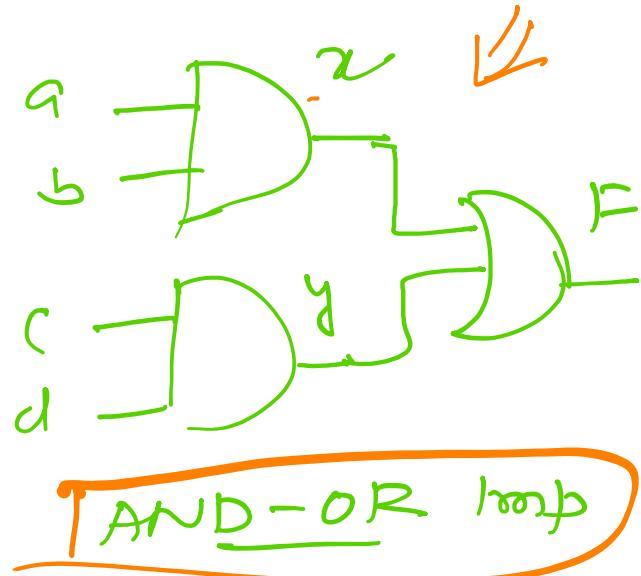
- Example: Complement  $F = \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z}$

$$\bar{F} = (\dot{x} + \bar{\dot{y}} + z)(\bar{\dot{x}} + y + z)$$



# Logic Expression (SOP)

- $F = a \cdot b + c \cdot d$        $\bar{F} = \overline{\bar{a} \cdot \bar{b} + \bar{c} \cdot \bar{d}}$
- $\bar{F} = (\bar{a} + \bar{b}) \cdot (\bar{c} + \bar{d})$
- $\bar{\bar{F}} = F = (\bar{a} + \bar{b}) \cdot (\bar{c} + \bar{d})$



# Logic Expression (SOP)

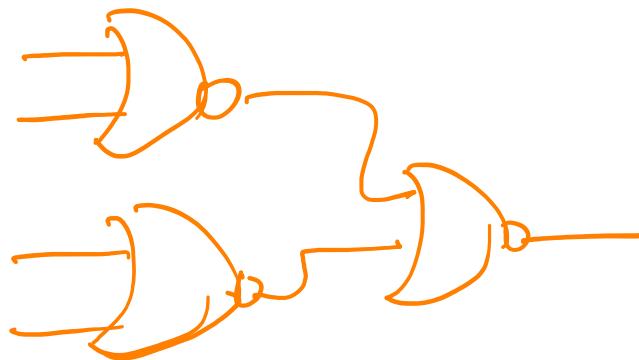
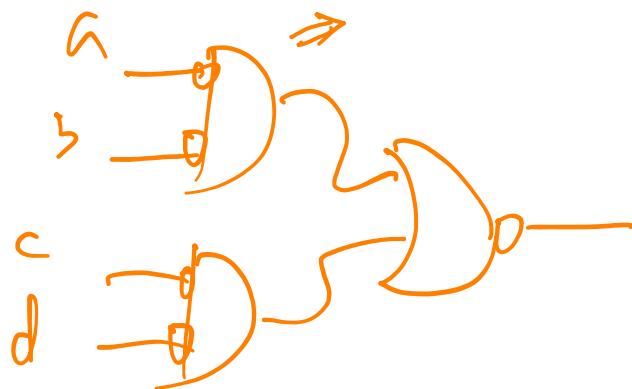
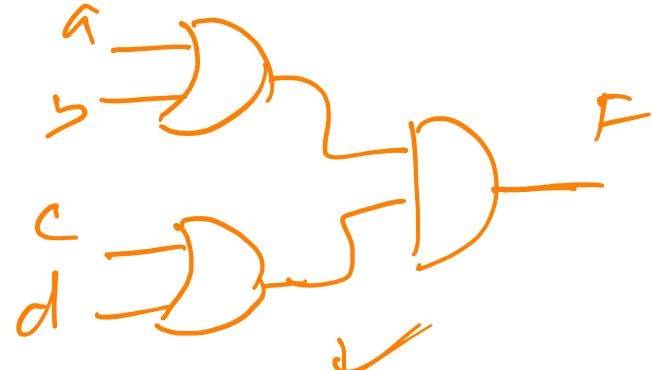
---

- $F = a \cdot b + c \cdot d$
- $F = \overline{(\overline{a} + \overline{b}) \cdot (\overline{c} + \overline{d})}$



# Logic Expression (POS)

- $F = (a + b) \cdot (c + d)$
- $\overline{F} = \overline{(a + b)} + \overline{(c + d)}$
- $\overline{F} = (\overline{a} \cdot \overline{b}) + (\overline{c} \cdot \overline{d})$
- $\overline{\overline{F}} = F = (\overline{a} \cdot \overline{b}) + (\overline{c} \cdot \overline{d})$



# Logic Expression (POS)

---

- $F = (a + b) \cdot (c + d)$
- $F = \overline{(\overline{a} \cdot \overline{b})} + \overline{(\overline{c} \cdot \overline{d})}$



# Truth Table

Truth Table

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

↑ Canonical  
Storage.      ⑧ bits

Logic Expression

$$F = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z$$

$$\overbrace{\quad\quad\quad}^{X \cdot Y \cdot \overline{Z}} + \overbrace{\quad\quad\quad}^{X \cdot Y \cdot Z}$$

→ <sup>g<sup>W</sup> bit</sup>      <sup>64 bit adder</sup>  
 $64 + 64 = 128$

$$2^{128} \text{ bits} \approx 2^{20} = 10^{36} \text{ bits}$$



# Min Terms

Truth Table

X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

0  
1  
2  
3  
4  
5  
6  
7

↑

$\bar{x} \cdot \bar{y} \cdot z$

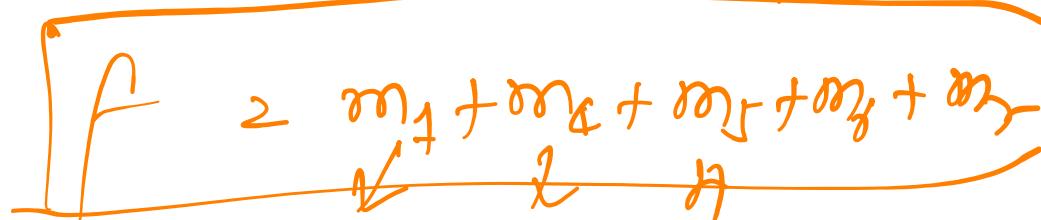
term.

Min

$\bar{x} \bar{y} z$

$$= \sum 1, 4, 5, 6, 7$$

$m_1, m_4, m_5, m_6, m_2$



$\bar{x} \bar{y} z$     $x \bar{y} z$     $x \bar{y} \bar{z}$

SOP



$$x + y + z$$

$$\bar{x} + \bar{y} + z$$

# Max Terms

$$x + \bar{y} + \bar{z}$$

Truth Table

X	Y	Z	F
0	0	0	0
1	0	1	1
2	1	0	0
3	1	1	0
4	1	0	1
5	1	1	1
6	1	0	1
7	1	1	1

$$F = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + x\bar{y}z$$

$$F: \underline{x+y+z} F = 1$$



$$M_0 \cdot M_2 \cdot M_3 \rightarrow$$

$$F: (x+y+z) \cdot (\bar{x}+\bar{y}+z) \cdot (\bar{x}+\bar{y}+\bar{z})$$

M<sub>0</sub>

POS

M<sub>2</sub>    M<sub>3</sub>

'Unique'



# Thank You



# Logic: Representation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 7 (18 January 2022)

**CADSL**

Components

$$\boxed{X \leftarrow \text{NOT } Y}$$

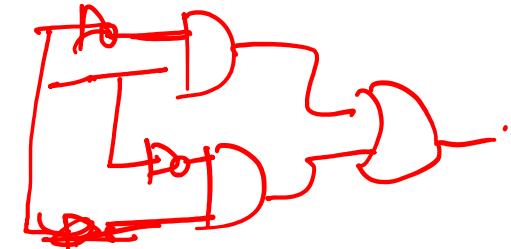
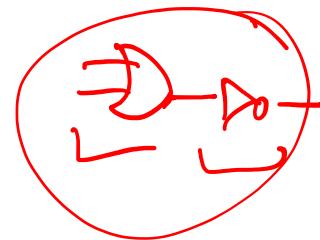
$$X \leftarrow Y \text{ AND } Z$$

$$X \leftarrow Y \text{ OR } Z$$



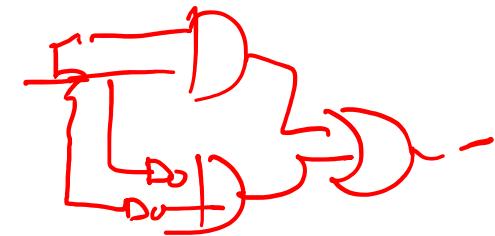
NOT, OR, AND

NAND, NOR, XOR, XNOR;



(VHDL file)

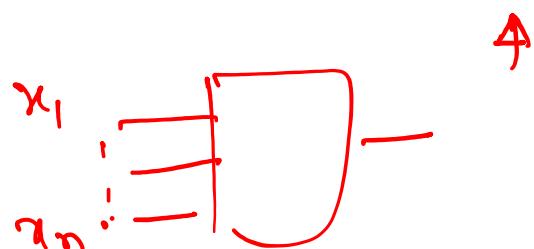
~~Project~~



# Truth Table/ Min Term/Max Term

Truth Table

	X Y Z	F
0 →	0 0 0	0 ✓
1	0 0 1	1
2	0 1 0	0
3	0 1 1	0
4	1 0 0	1
5	1 0 1	1
6	1 1 0	1
7	1 1 1	1



$$\left. \begin{aligned}
 & \text{SOP} \\
 & \frac{x \cdot y + y \cdot z}{PDS} \\
 & (x+y) \cdot (y+z)
 \end{aligned} \right] \text{Not Canonical}$$

$$\left. \begin{aligned}
 & \frac{x \cdot y + z \cdot (x+y)}{\square} \checkmark
 \end{aligned} \right]$$

←  
Canonical

$$\left. \begin{aligned}
 & \boxed{1, 4, 5, 6, 7} \} \text{ True} \\
 & \boxed{0, 2, 3} \} \perp
 \end{aligned} \right.$$

B

# Truth Table/ Min Term/Max Term

Truth Table

X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

Canonical.

$$\bar{x} \cdot \bar{y} \cdot z$$

$$F(x, y, z)$$

$$\bar{x} \cdot \bar{y} \cdot z \rightarrow F.$$

Implicant

$$F = m_1 + m_4 + m_5 + m_6 + m_7$$

$$x \bar{y} \bar{z}$$



# Truth Table/ Min Term/Max Term

Truth Table

X	Y	Z	F
0	0	0	0 ✓
0	0	1	1
0	1	0	0 ✓
0	1	1	0 ✓
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$\bar{F} = (\bar{x} \cdot \bar{y} \cdot \bar{z}) + (\bar{x} \cdot y \cdot \bar{z}) + (\bar{x} \cdot y \cdot z)$$

$$F = \bar{F} = ( )$$

clause.

$$(x+y+z) \cdot (x+\bar{y}+z) \cdot (x+y+\bar{z})$$

↑

M<sub>0</sub>

↑

M<sub>2</sub>

↑

M<sub>3</sub>

$$F = M_0 \cdot M_2 \cdot M_3$$

(POS)



# Logic Expression

Truth Table

X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

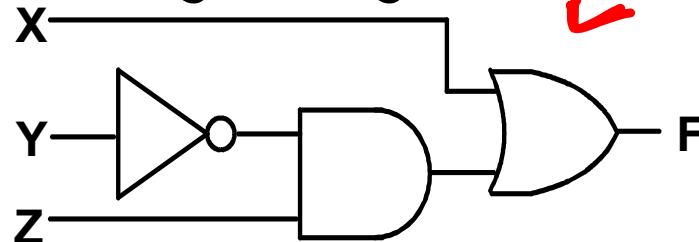
Logic Expression

$$F = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z$$
$$\quad \quad \quad \overline{X} \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$$

Logic Expression *Boole's Algebra*

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

Logic Diagram



6  
 $\overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z$   
 SOP  
 AND - OR  
 NAND - NAND

SOP

OR-AND. NOR-nor.

2N

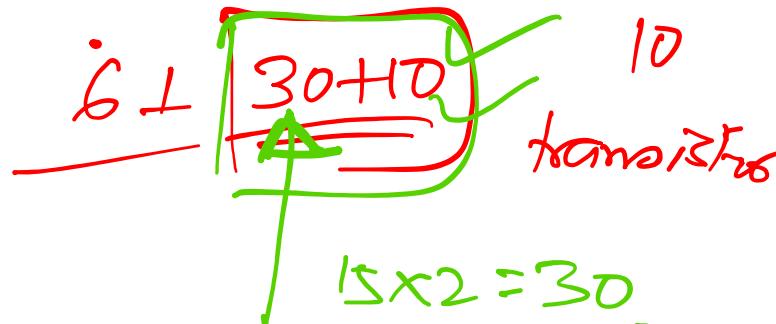
# What to Minimize?

## Logic Expression

$$F = \overline{\underline{\overline{X} \cdot \overline{Y} \cdot Z}} + \overline{\underline{X \cdot \overline{Y} \cdot \overline{Z}}} + \overline{\underline{X \cdot \overline{Y} \cdot Z}} + \overline{\underline{X \cdot Y \cdot \overline{Z}}} + \overline{\underline{X \cdot Y \cdot Z}}$$

AND - OR

NAND - NAND

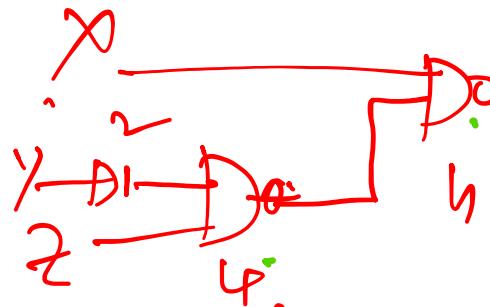


↓ Boolean Algebra

Min { # literals +  
#.prod. terms }

## Logic Expression

$$F = \overline{X} + \overline{Y} \cdot Z$$



10

8



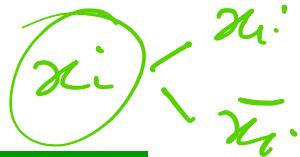
# Graphical Method: Binary Decision Diagram



$$y = f(x_1, x_2, \dots, x_n)$$



# Binary Decision Diagram



- ❖ BDD is canonical form of representation

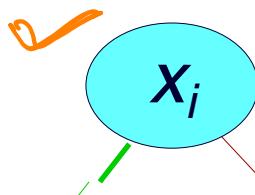
- ❖ Shanon's expansion theorem

$$f(x_1, x_2, \dots, x_i, \dots, x_n)$$

$$f(x_1, x_2, \dots, x_i, \dots, x_n) = x_i \cdot f(x_1, x_2, \dots, x_i=1, \dots, x_n) + x_i' \cdot f(x_1, x_2, \dots, x_i=0, \dots, x_n)$$

$$\begin{aligned} & x_1 \cdot f_{x_1} = x_2 + x_3 \\ & x_1' \cdot f_{\bar{x}_1} = 0 \end{aligned}$$

$f_{x_i} \uparrow$  Cofactor



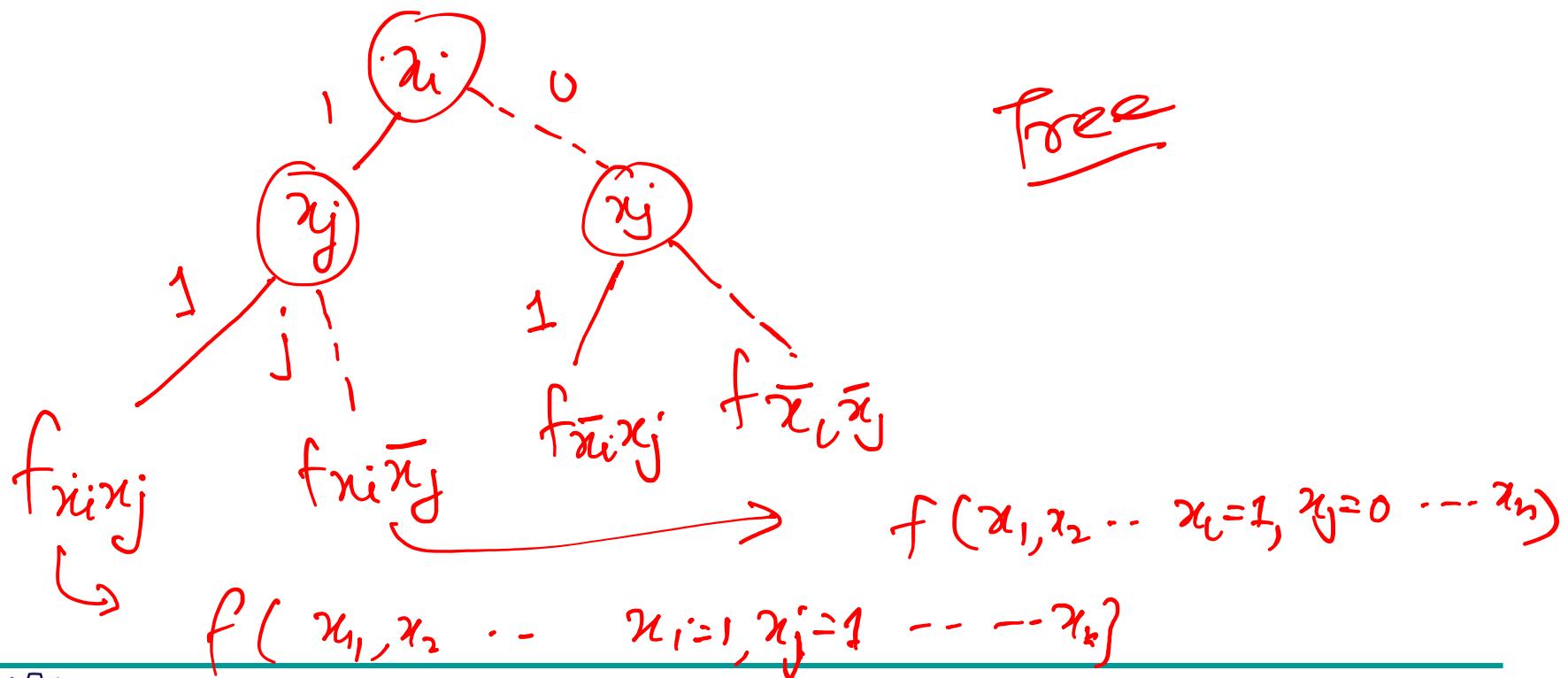
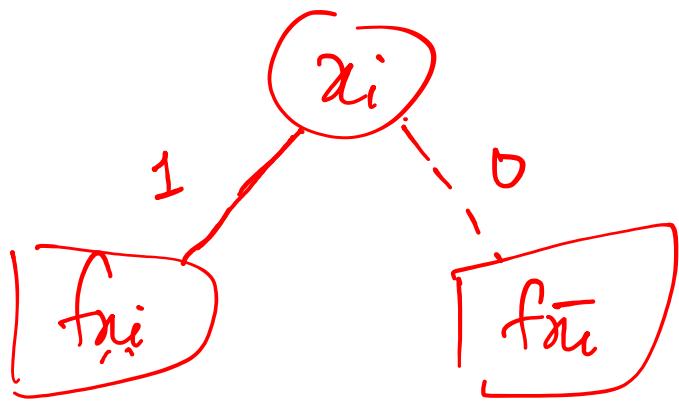
$$x_1 \cdot (x_2 + x_3) + \bar{x}_1 \cdot 0 = x_1 x_2 + x_1 x_3$$

✓  $f(x_1, x_2, \dots, x_i=1, \dots, x_n)$

✓  $f(x_1, x_2, \dots, x_i=1, \dots, x_n)$

$$f_{\bar{x}_i}$$





# Decision Structures

$x_1, x_2, x_3$

ordered,  
 $x_1 < x_2 < x_3$ .

Truth Table

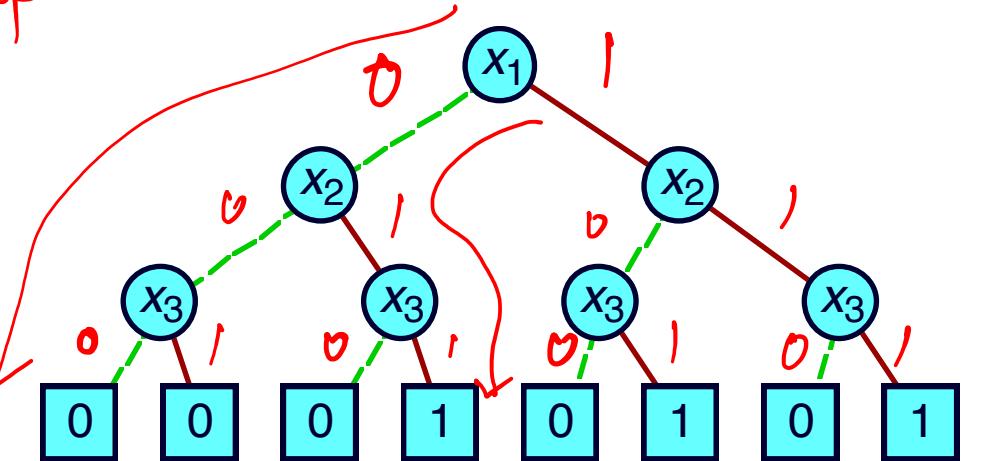
$x_1$	$x_2$	$x_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$2^n$

$2^{nH}$

Graphical

Decision Tree



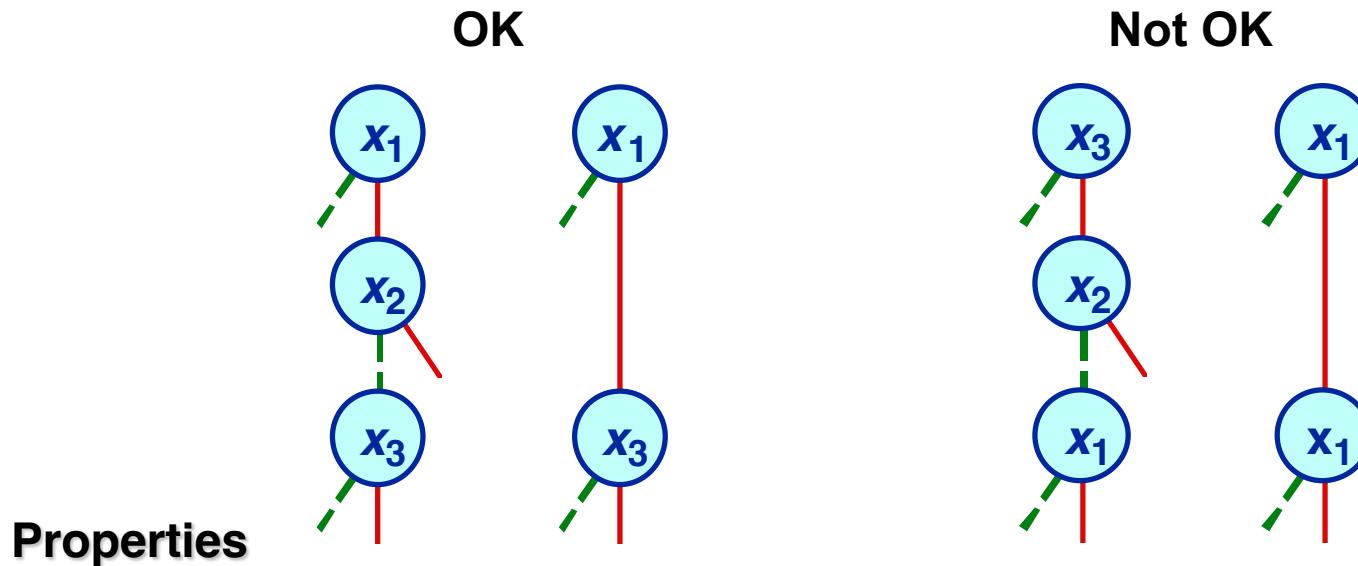
- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

$2^n$

leaf  
evaluat-

# Variable Ordering

- ❖ Assign arbitrary total ordering to variables Go
- e.g.,  $x_1 < x_2 < x_3$
- ❖ Variables must appear in ascending order along all paths



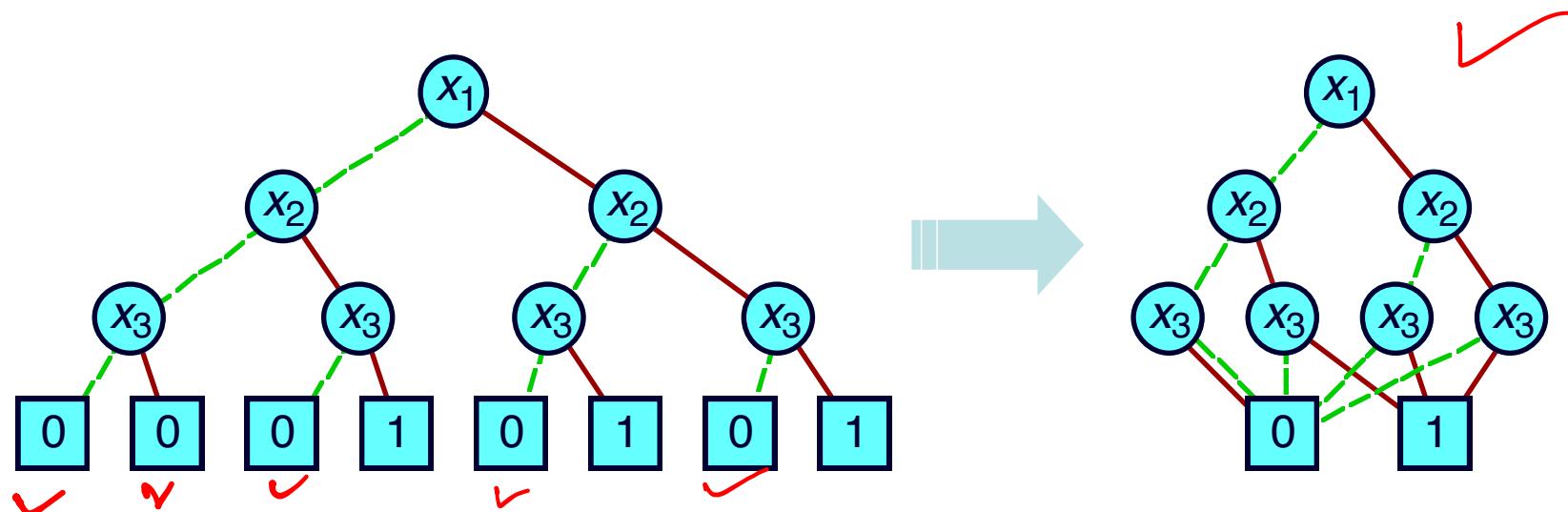
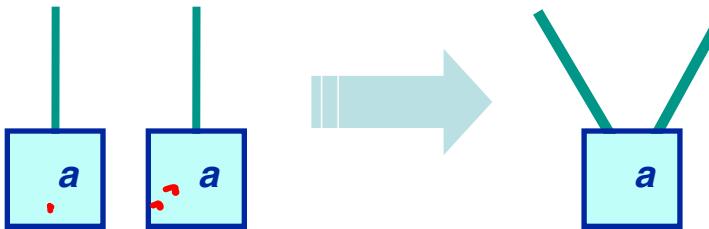
- No conflicting variable assignments along path
- Simplifies manipulation



# Reduction Rule #1

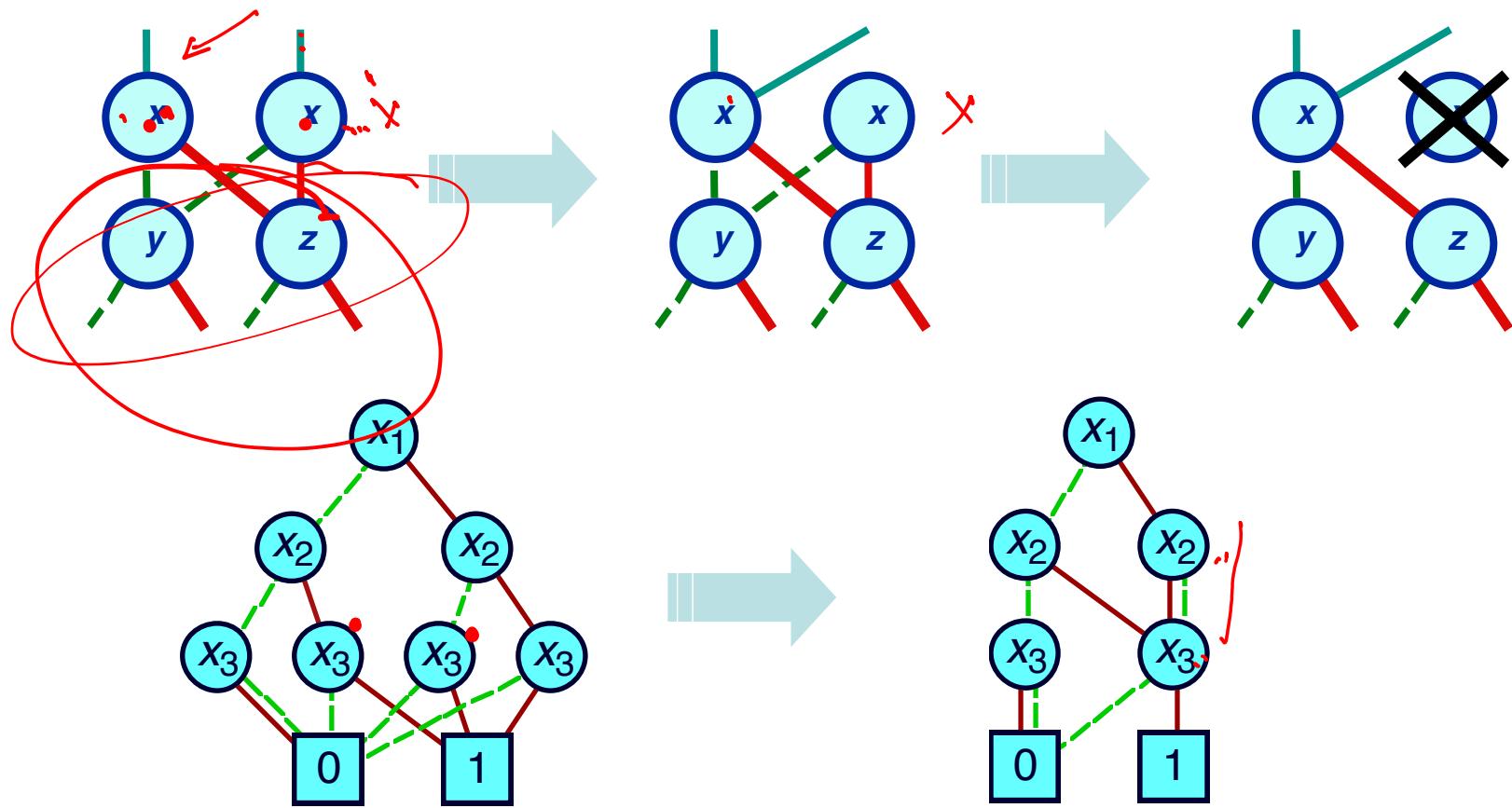
---

Merge equivalent leaves



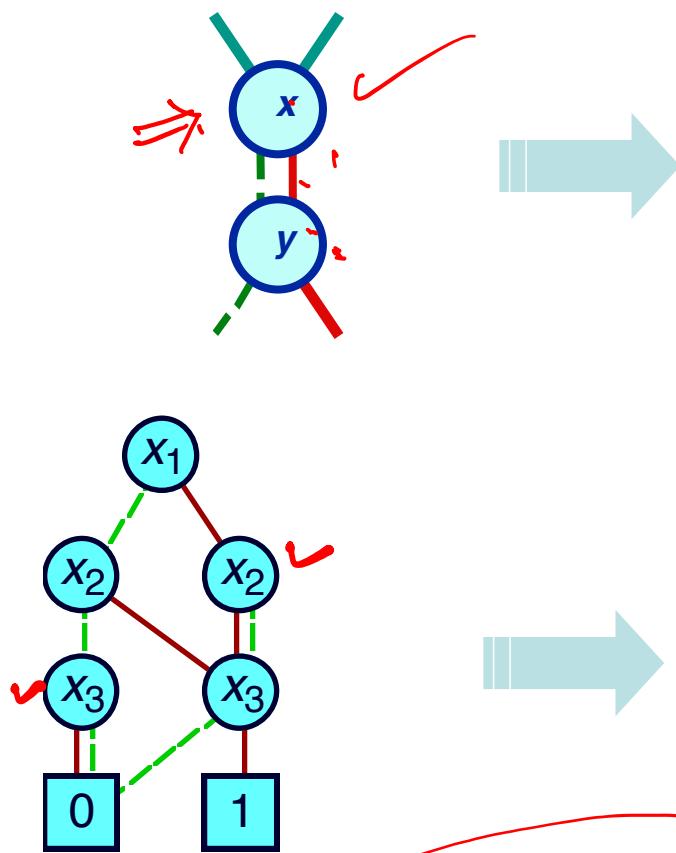
# Reduction Rule #2

Merge isomorphic nodes



# Reduction Rule #3

Eliminate Redundant Tests



Binary  
Ordered Decision Tree Graph.  
Reduced, ordered, binary decision diagram  
(ROBDD)  
Compact  
Canonical.



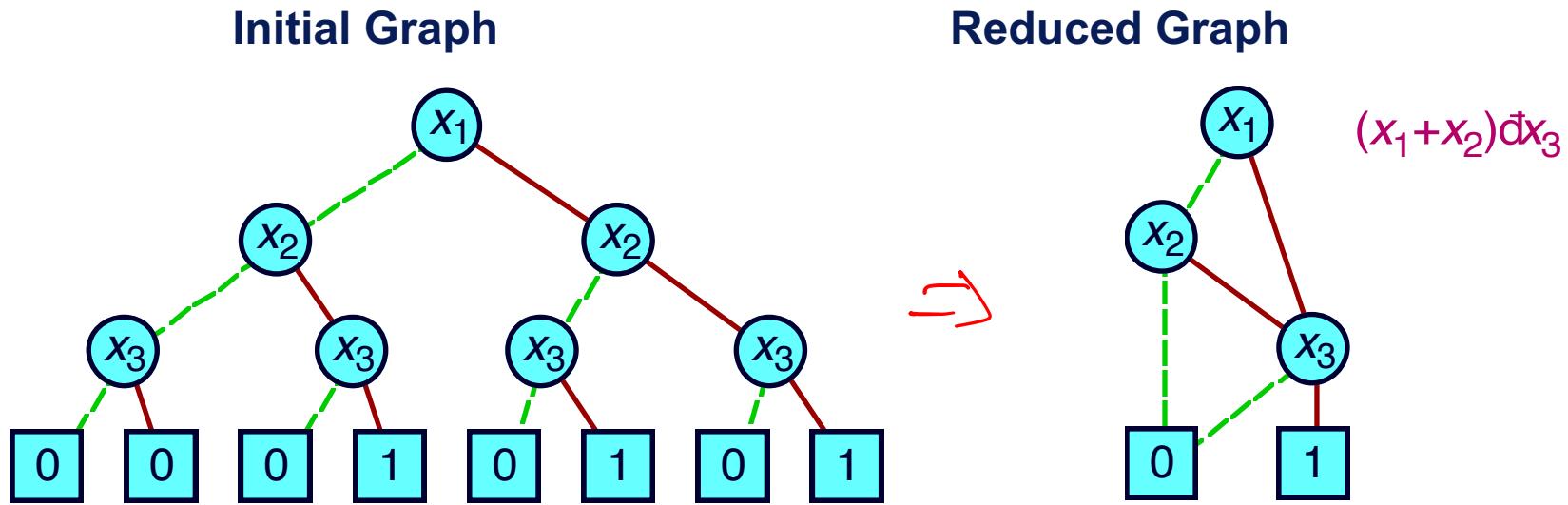
$x_1, x_2 + \bar{x}_2 \cdot x_3$   
18 Jan 2022

$\bar{x}_1 \cdot x_2$   
CS-230@IITB

$x_1, x_3 + \bar{x}_1 \cdot x_2 \cdot x_3$   
15

Unique CADSL

# Example OBDD

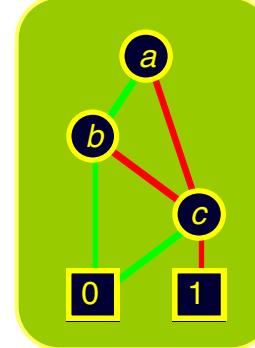
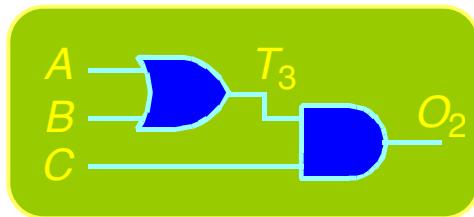
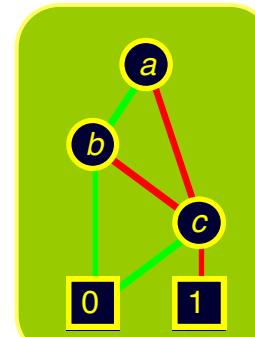
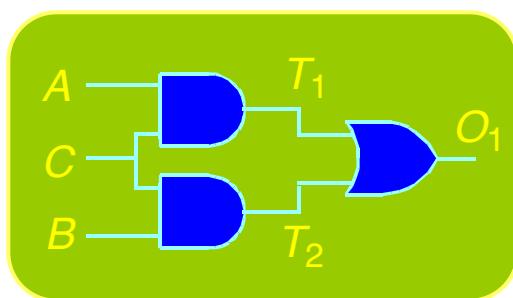


- Canonical representation of Boolean function
  - ❖ For given variable ordering
  - Two functions equivalent if and only if graphs isomorphic
    - o Can be tested in linear time
  - Desirable property: *simplest form is canonical.*



# Binary Decision Diagram

- Generate Complete Representation of Circuit Function
  - Compact, canonical form



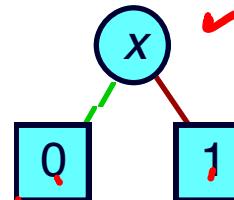
- Functions equal if and only if representations identical
- Never enumerate explicit function values
- Exploit structure & regularity of circuit functions

# Example Functions

## Constants

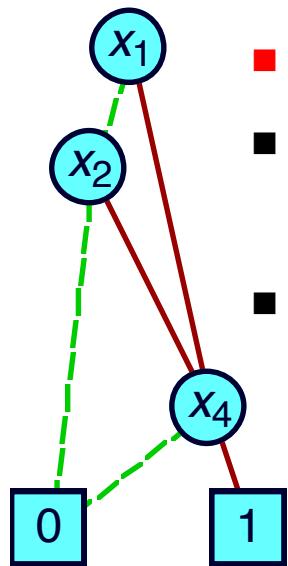
- ✓  Unique unsatisfiable function
- ✓  Unique tautology

## Variable



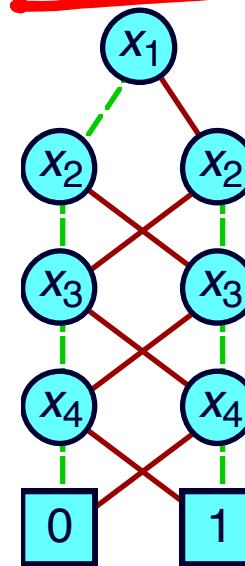
Treat variable  
as function

## Typical Function



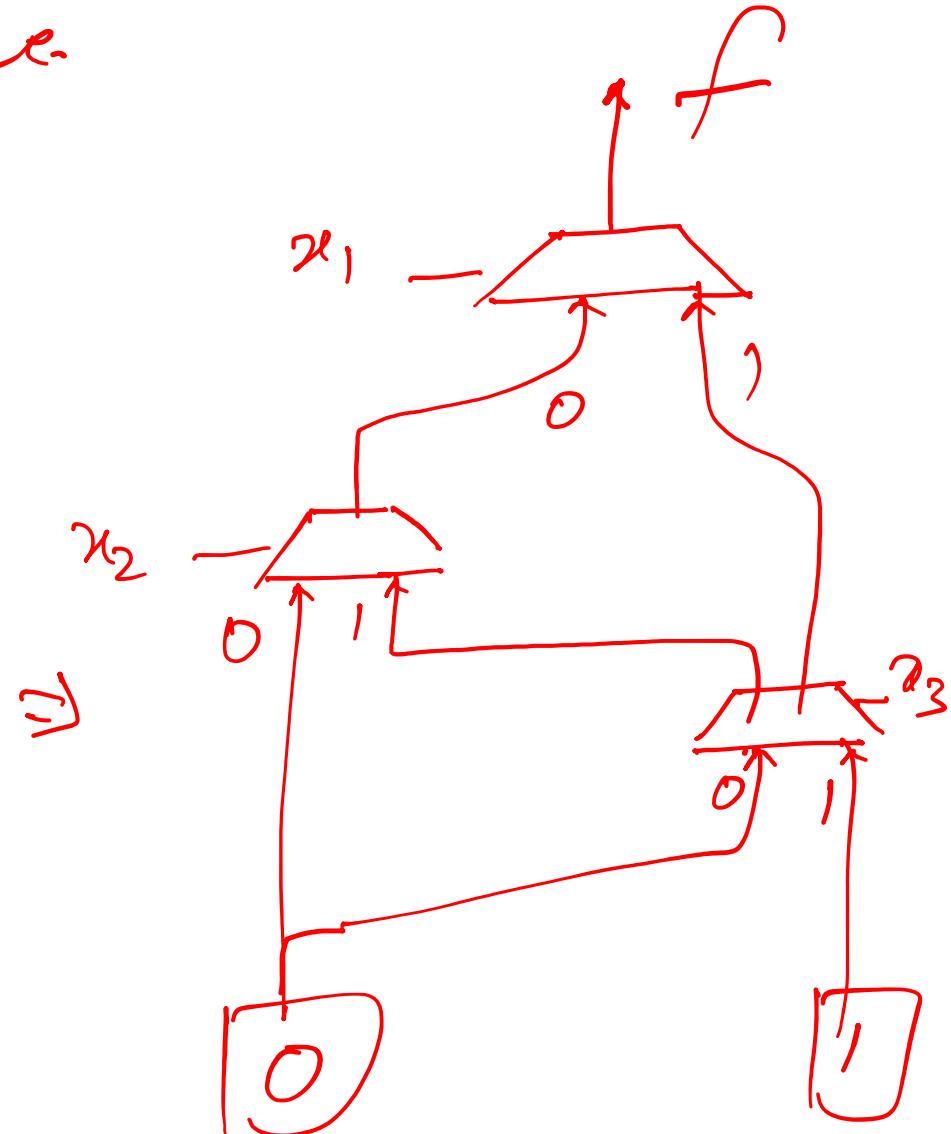
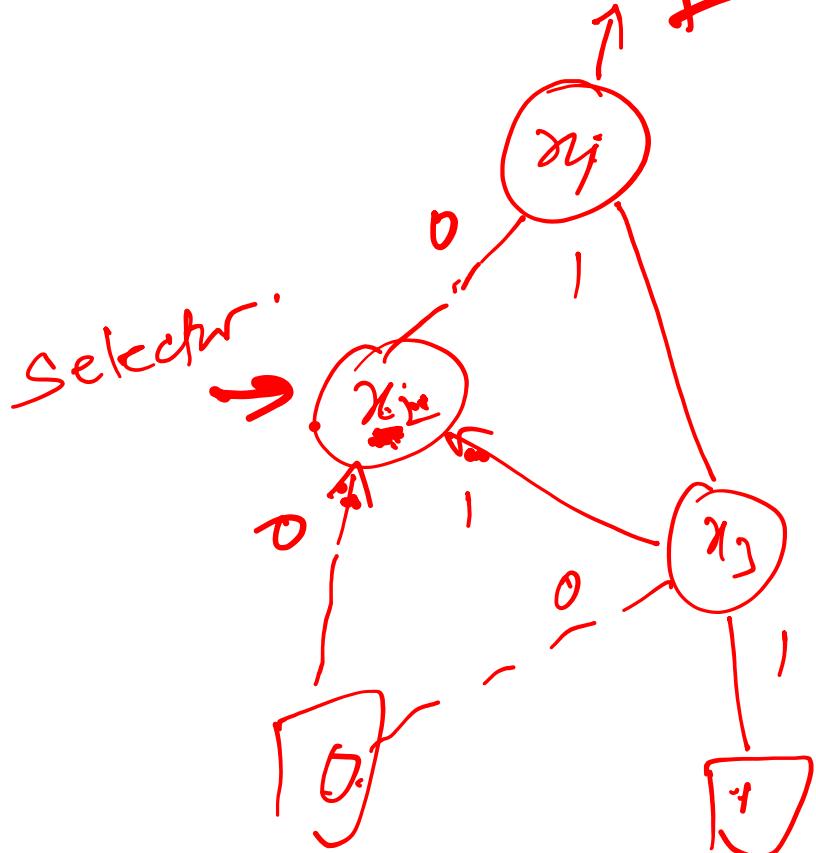
- $(x_1 + x_2) \cdot x_4$
- No vertex labeled  $x_3$ 
  - ◆ independent of  $x_3$
- Many subgraphs shared

## Odd Parity



$x_1 \oplus x_2 \oplus x_3 \oplus x_4$   
sym.  
Linear representation

Synthesize  
 $f$



# Thank You



# Logic: Representation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 8 (20 January 2022)

**CADSL**

## Representations

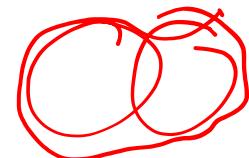
Truth Table

Sum Of Minterm.  $\leftarrow \underline{\text{SOP}}$  ✓ true.

Product Of Maxterms  $\leftarrow \underline{\text{POS}}$  ✓ false.

Canonical.

Z:Y .



Union



# Graphical Method: Binary Decision Diagram



# Binary Decision Diagram

- ❖ BDD is canonical form of representation

- ❖ Shanon's expansion theorem

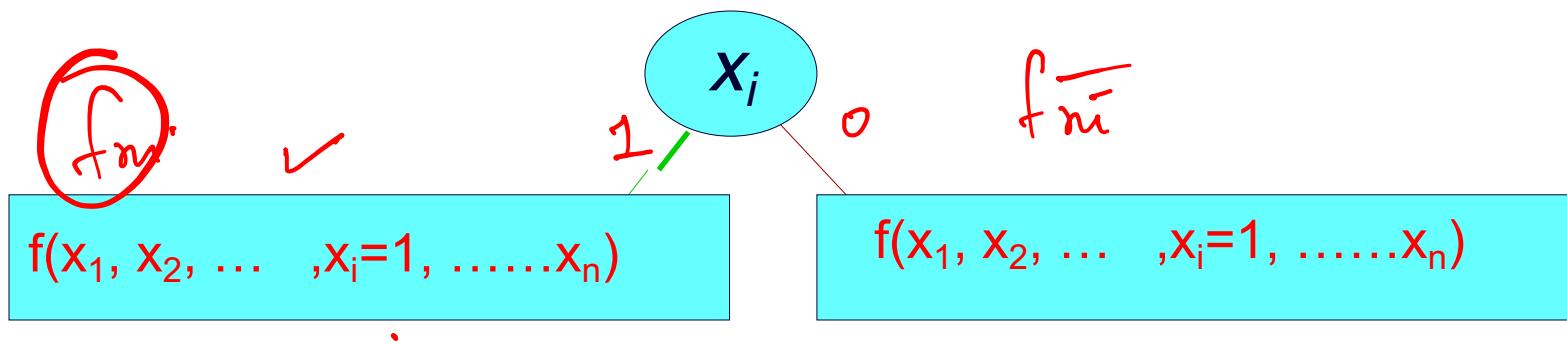
- ❖  $f(x_1, x_2, \dots, x_i, \dots, x_n) =$

$$x_i \cdot f(x_1, x_2, \dots, x_i=1, \dots, x_n) +$$

$$x_i' \cdot f(x_1, x_2, \dots, x_i=0, \dots, x_n)$$

$$\underline{x_i} \cdot \underline{f_{xi}} + \bar{\underline{x_i}} \cdot \underline{\bar{f}_{xi}}$$

$$f = \frac{f_{xi}}{f_{\bar{xi}}}$$

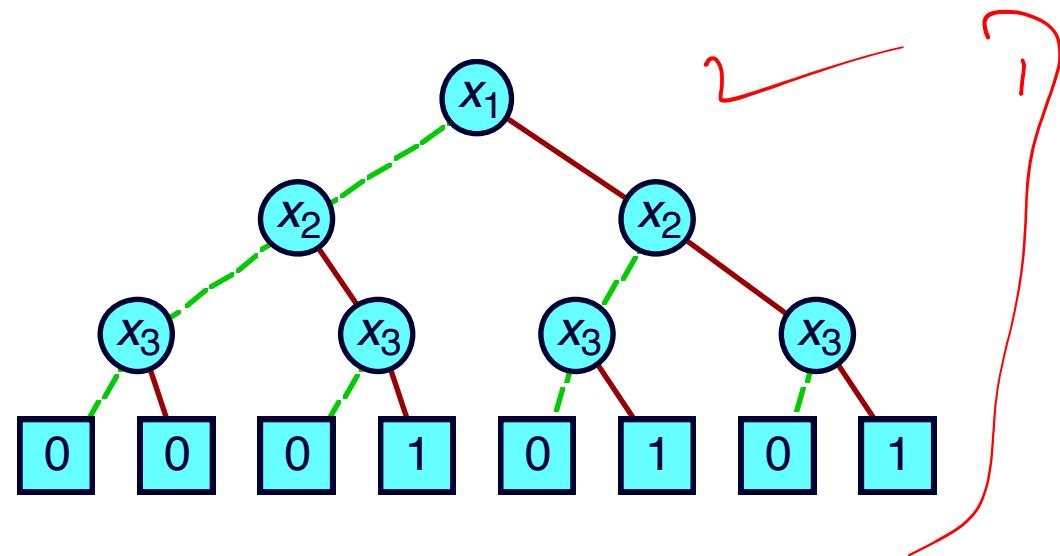


# Decision Structures

Truth Table

$x_1$	$x_2$	$x_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

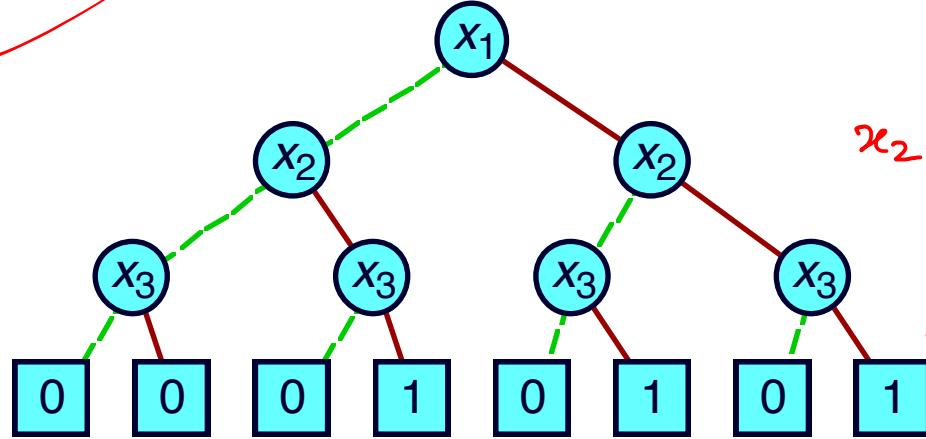


# Example OBDD

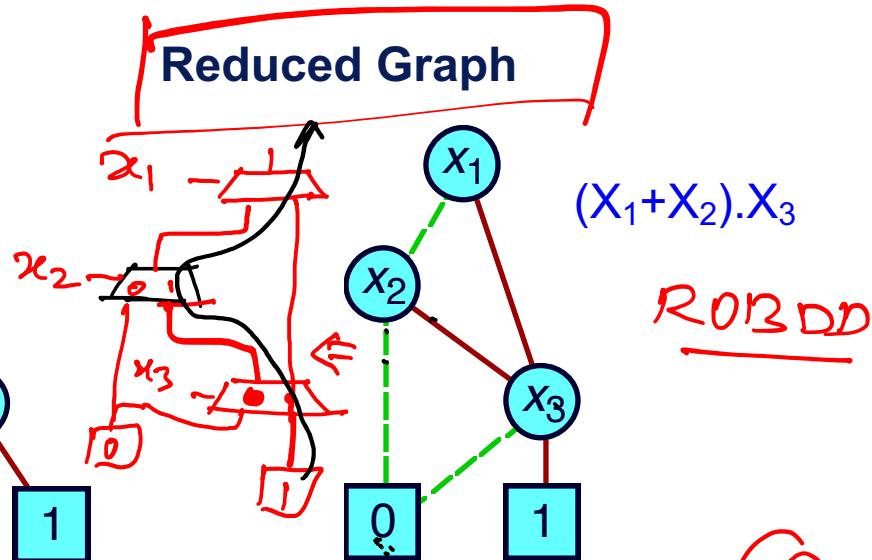
$$x_1 < x_2 < x_3$$

$$f_1 \equiv f_2$$

Initial Graph



Reduced Graph

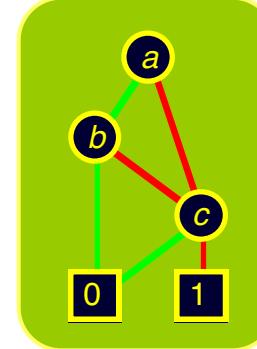
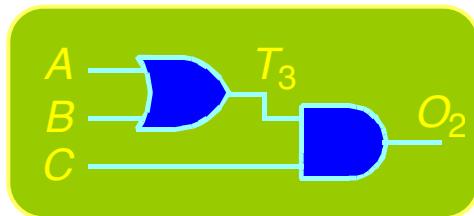
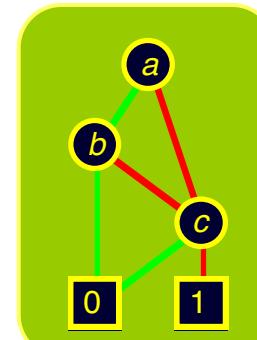
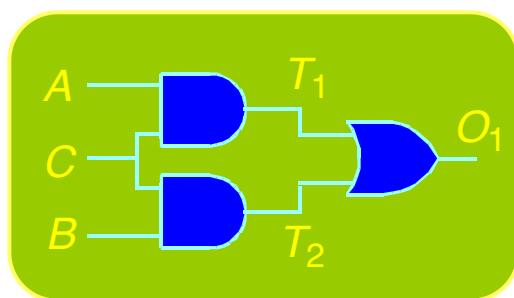


- Canonical representation of Boolean function
  - ❖ For given variable ordering
  - Two functions equivalent if and only if graphs isomorphic
    - o Can be tested in linear time
  - Desirable property: *simplest form is canonical.*



# Binary Decision Diagram

- Generate Complete Representation of Circuit Function
  - Compact, canonical form



- Functions equal if and only if representations identical
- Never enumerate explicit function values
- Exploit structure & regularity of circuit functions



cost  $\rightarrow$  # MUX

# nodes (in graph)

ROBDD  
 $\equiv$

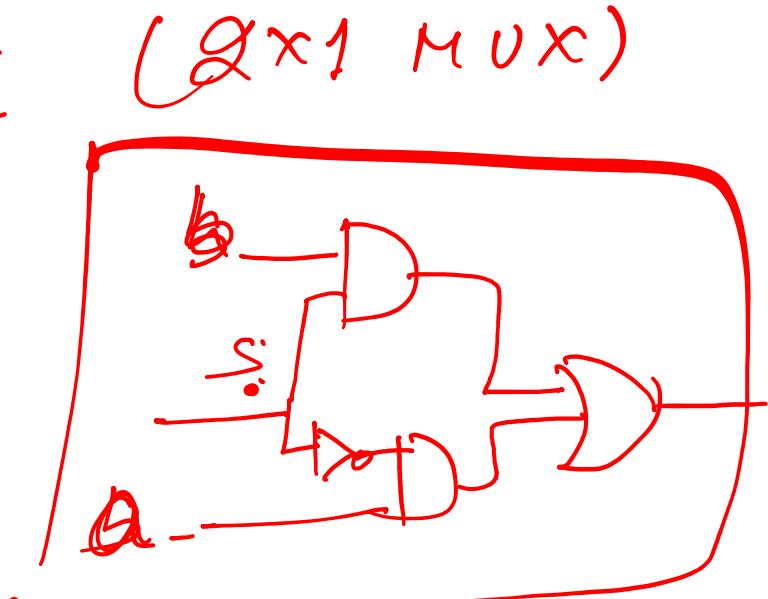
Delay (Performance)

(longest path)

$\Downarrow$  3 MUX delay)

bound

$\equiv$  MUX delay



$$f = a \cdot \bar{s} + b \cdot s$$

Selector

1 MUX delay = t

N.T.



Shape & size of ROBDD

depends on Order of variable)

Th.

NP Complete

N nodes :

$N'$  node =

$\lceil \frac{N'-N}{2} \rceil$

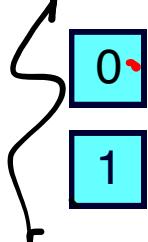
Synthesis ✓

VERIFICATION



# Example Functions

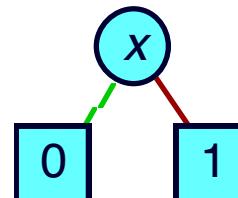
## Constants



Unique unsatisfiable function

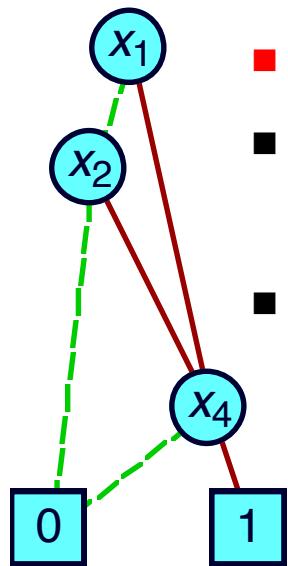
Unique tautology

## Variable



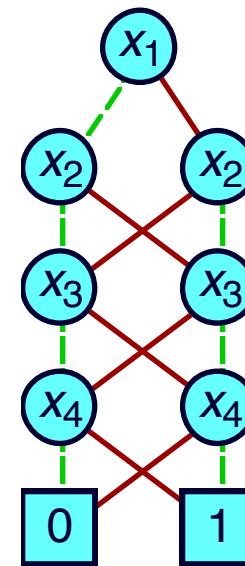
Treat variable as function

## Typical Function



- $(x_1 + x_2) \cdot x_4$
- No vertex labeled  $x_3$ 
  - ◆ independent of  $x_3$
- Many subgraphs shared

## Odd Parity



Linear representation

# Operations with BDD

---

- ❖ Let  $v_1, v_2$  denote root nodes of  $f_1, f_2$  respectively , with  $\text{var}(v_1) = x_1$  and  $\text{var}(v_2) = x_2$
- ❖ If  $v_1$  and  $v_2$  are leafs,  $f_1 \text{ OP } f_2$  is a leaf node with value  $\text{val}(v_1) \text{ OP } \text{val}(v_2)$

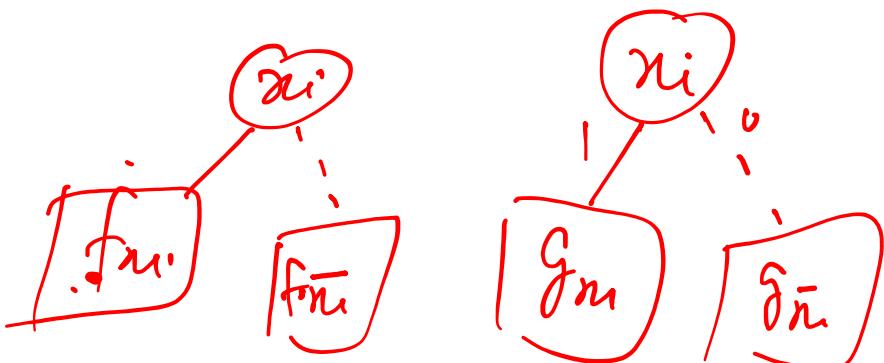


$$f(x_1, x_2 \dots x_i \dots x_n)$$

$$g(x_1, x_2 \dots x_i \dots x_n)$$

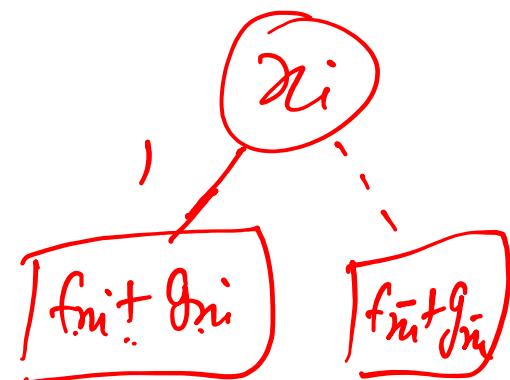
$$x_i f_{xi} + \bar{x}_i \cdot f_{\bar{x}_i}$$

$$x_i g_{xi} + \bar{x}_i \cdot g_{\bar{x}_i}$$



OR

$$\underline{f+g}$$



$$x_i f_{xi} + \bar{x}_i f_{\bar{x}_i} + x_i g_{xi} + \bar{x}_i g_{\bar{x}_i}$$

$$\Rightarrow x_i (f_{xi} + g_{xi}) + \bar{x}_i (f_{\bar{x}_i} + g_{\bar{x}_i})$$

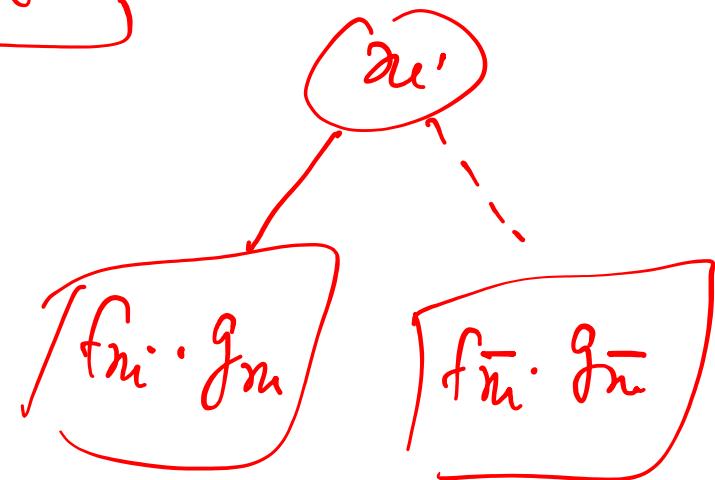


AND

f · g

$$(\pi_i f_{ni} + \bar{\pi}_i \bar{f}_{ni}) \cdot (\pi_j g_{nj} + \bar{\pi}_j \bar{f}_{nj})$$

$$\pi_i \underbrace{f_{ni} g_{nj}}_{\downarrow \cdot} + \bar{\pi}_i \underbrace{\bar{f}_{ni} \cdot g_{nj}}_{\uparrow}$$

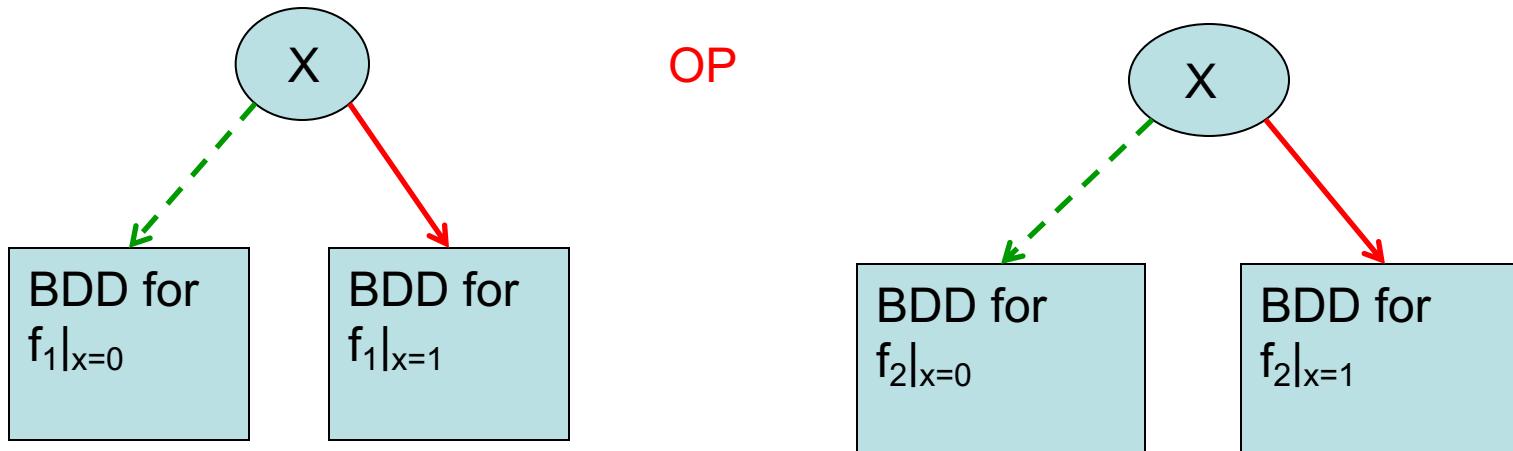


# Operations with BDD

---

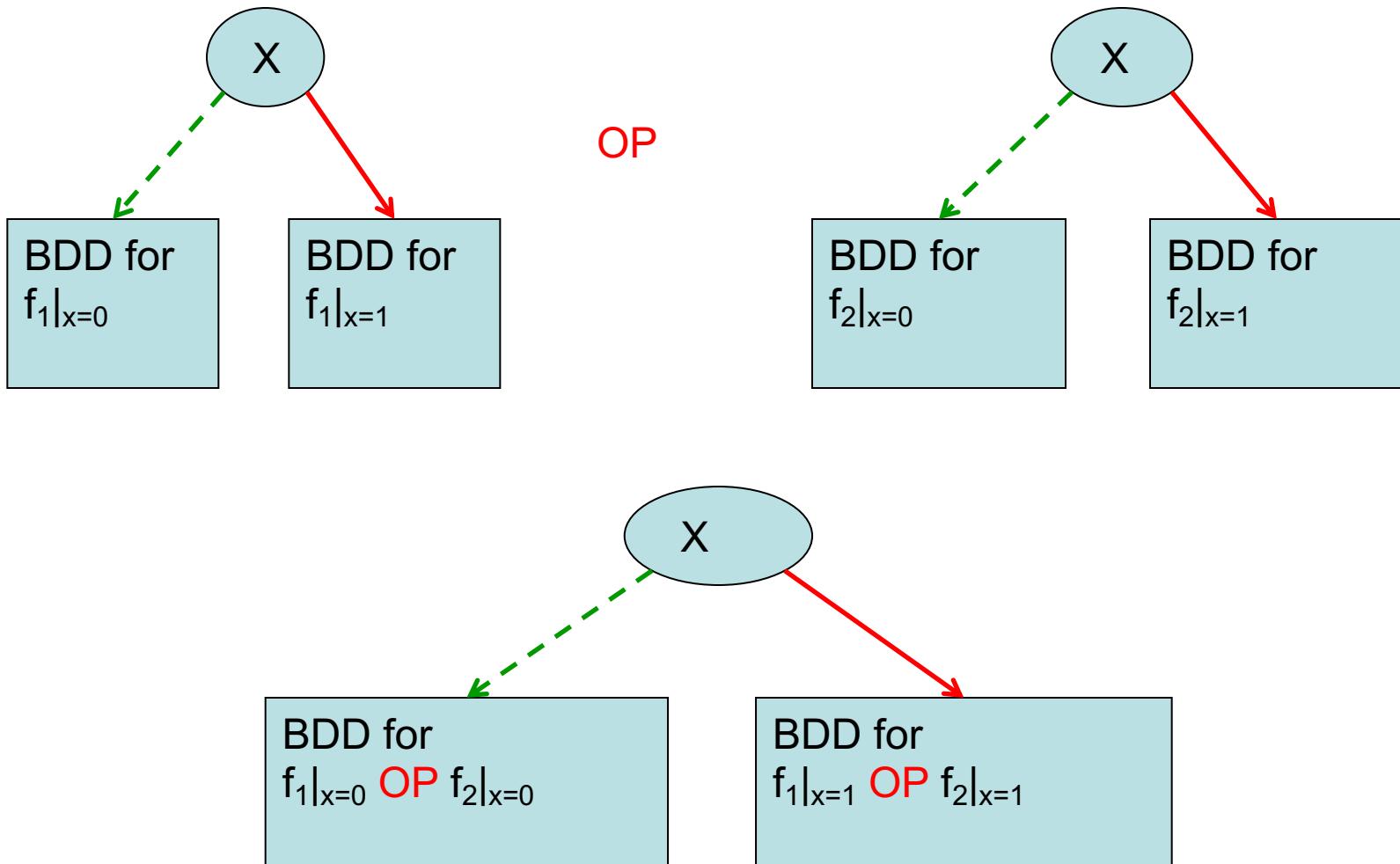
- ❖ If  $x_1 = x_2 = x$ , apply Shannon's expansion

$$f_1 \text{ OP } f_2 = x' \cdot (f_1|_{x=0} \text{ OP } f_2|_{x=0}) + x \cdot (f_1|_{x=1} \text{ OP } f_2|_{x=1})$$



# Operations with BDD

---



$$f(x_1, x_2, \dots, x_n) \quad g(\underline{x_1 x_3 \dots x_n})$$

$$(x_1 \cdot f_m + \bar{x}_1 \bar{f}_{\bar{m}}) \cdot g.$$

$$x_1 \cdot (f_m \cdot g) + \bar{x}_1 (\bar{f}_{\bar{m}} \cdot g)$$

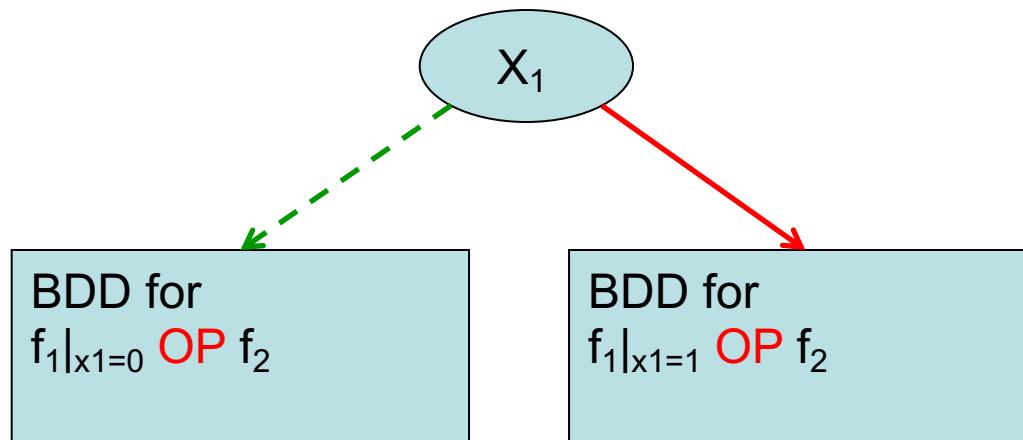


# Operations with BDD

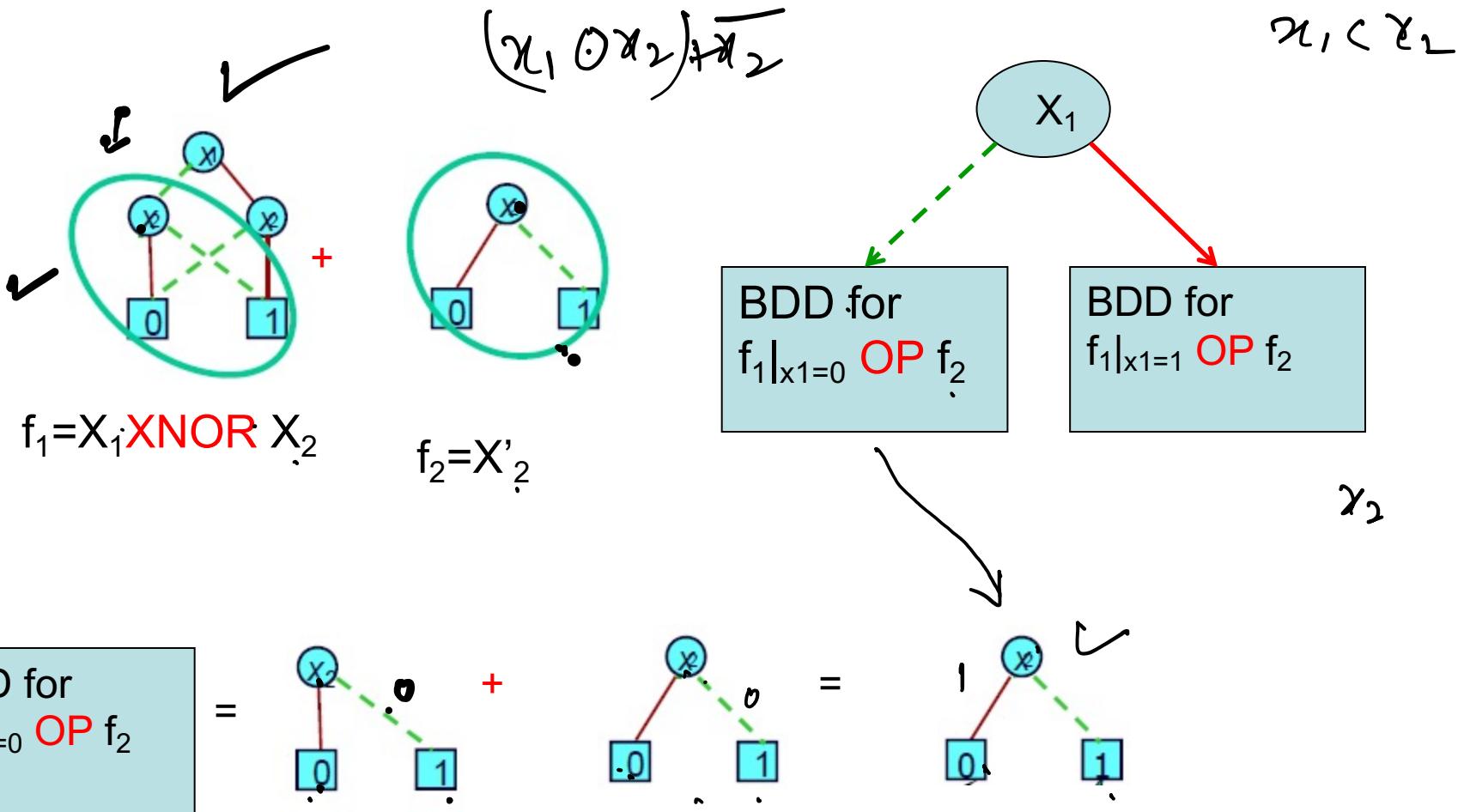
---

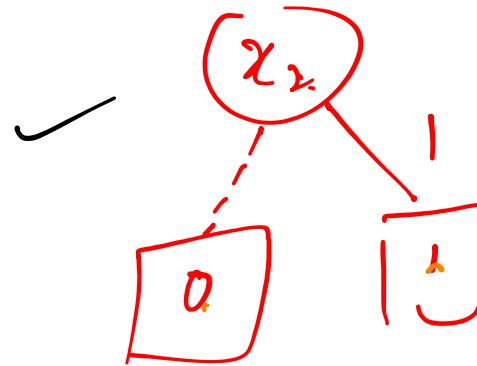
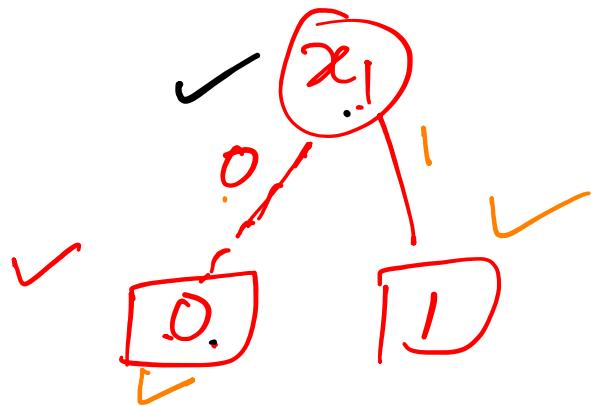
- ❖ Else suppose  $x_1 < x_2 = x$ , in variable order

$$f_1 \text{ OP } f_2 = x'_1 (f_1|_{x_1=0} \text{ OP } f_2) + x_1 (f_1|_{x_1=1} \text{ OP } f_2)$$



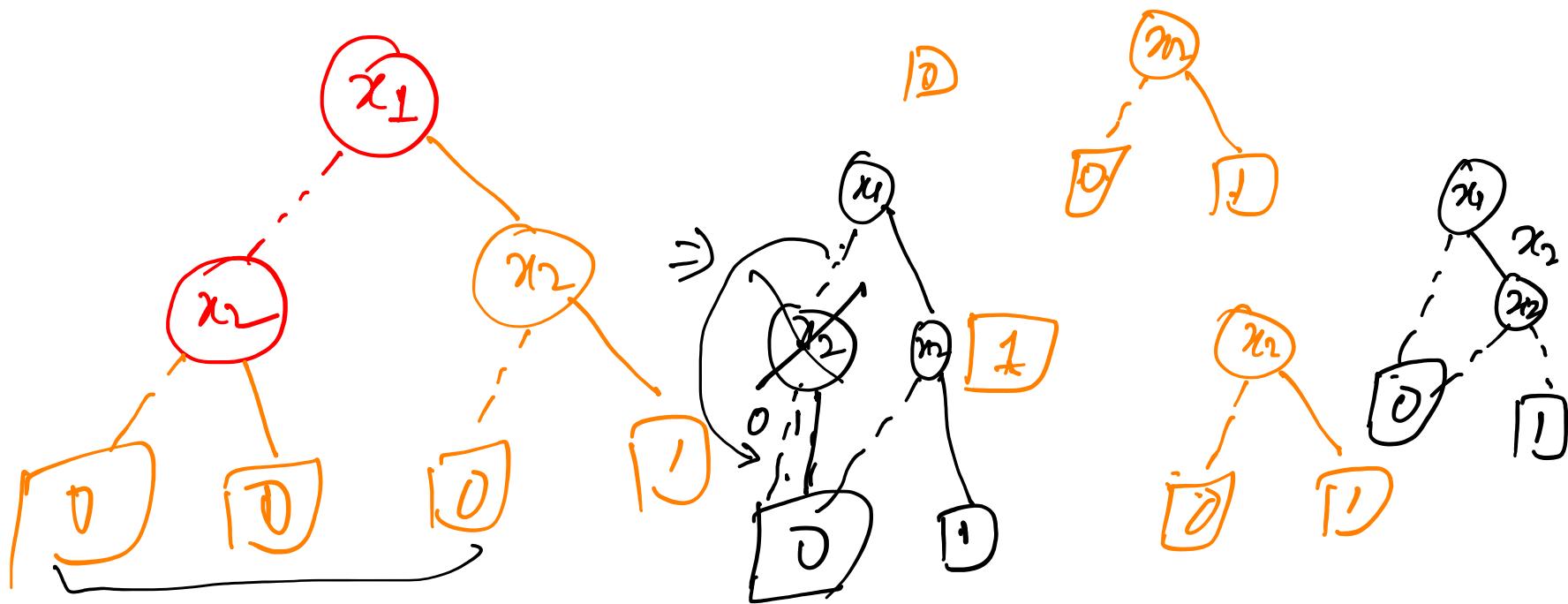
# Operations with BDD: Example



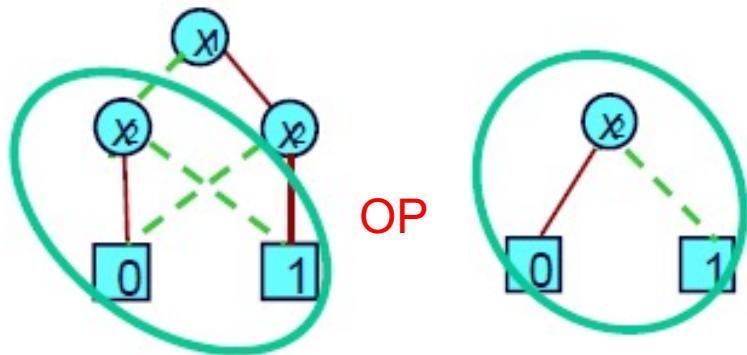


$x_1 < x_2$   
order ✓

$$f = \frac{x_1 - x_2}{-}$$



# Operations with BDD: Example

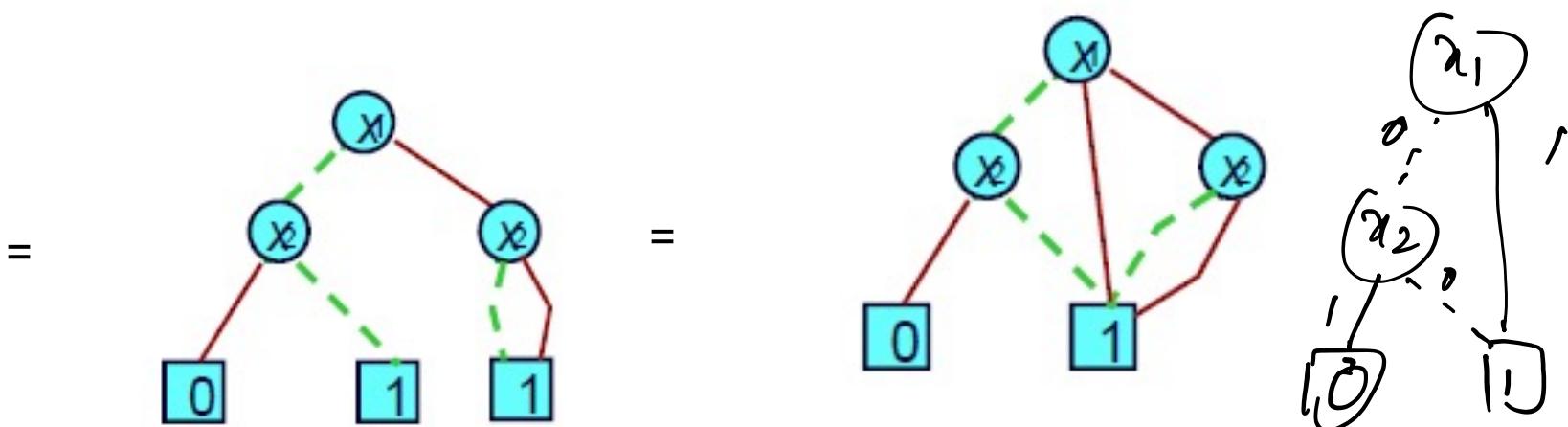
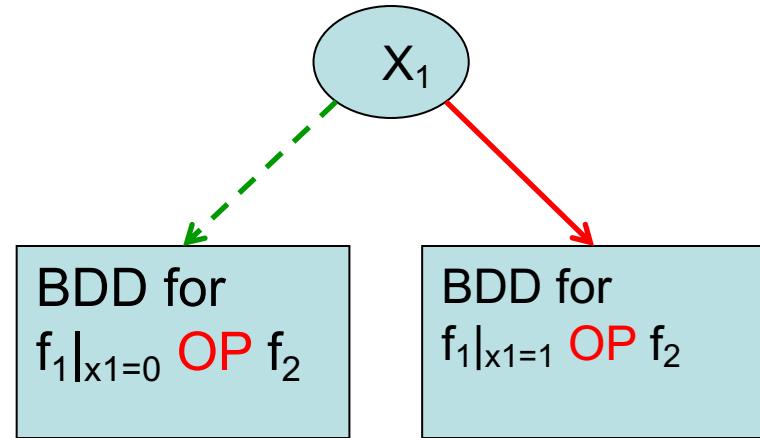


OP

$$f_1 = X_1 \text{XNOR} X_2$$

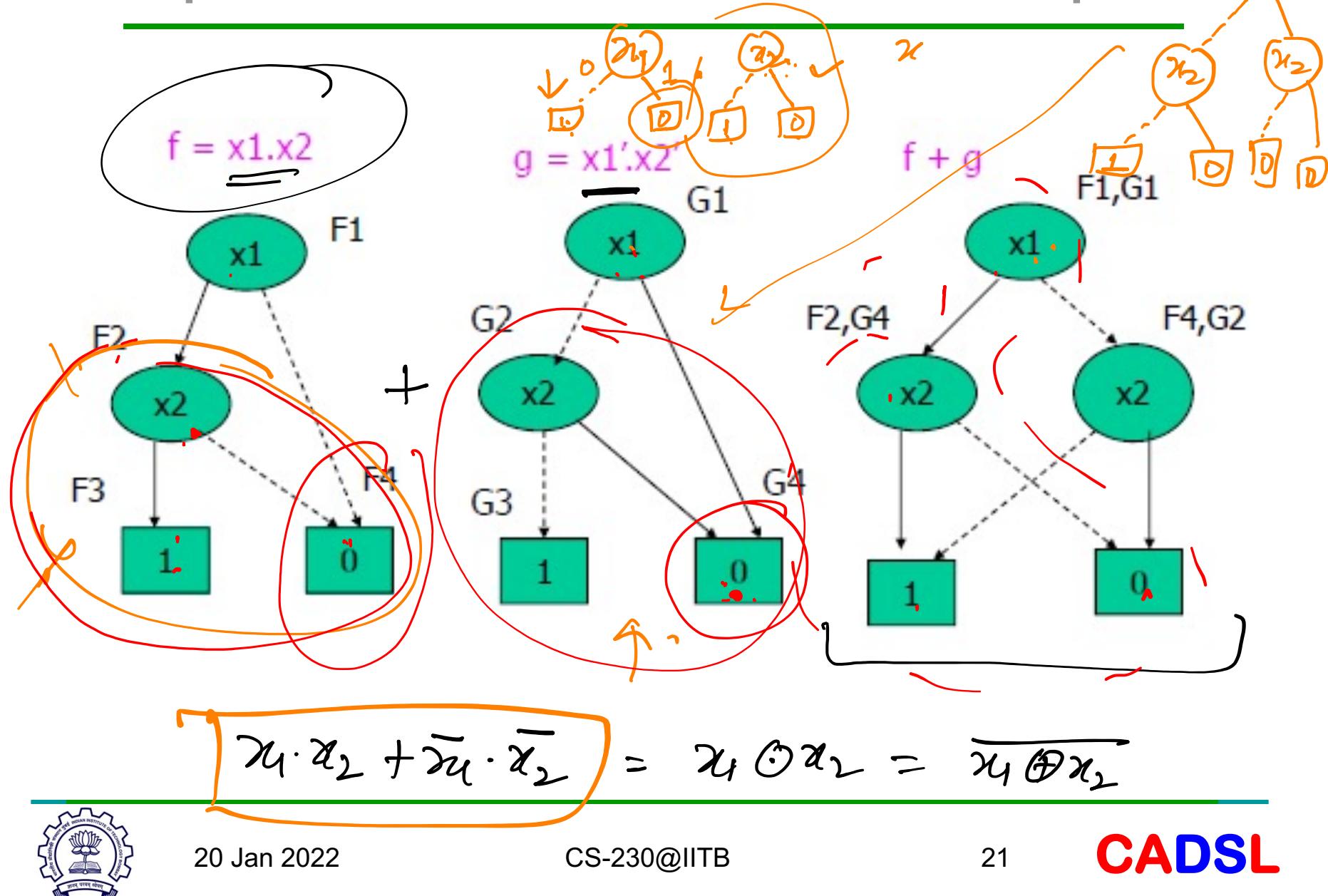
$$f_2 = \bar{X}_2$$

=

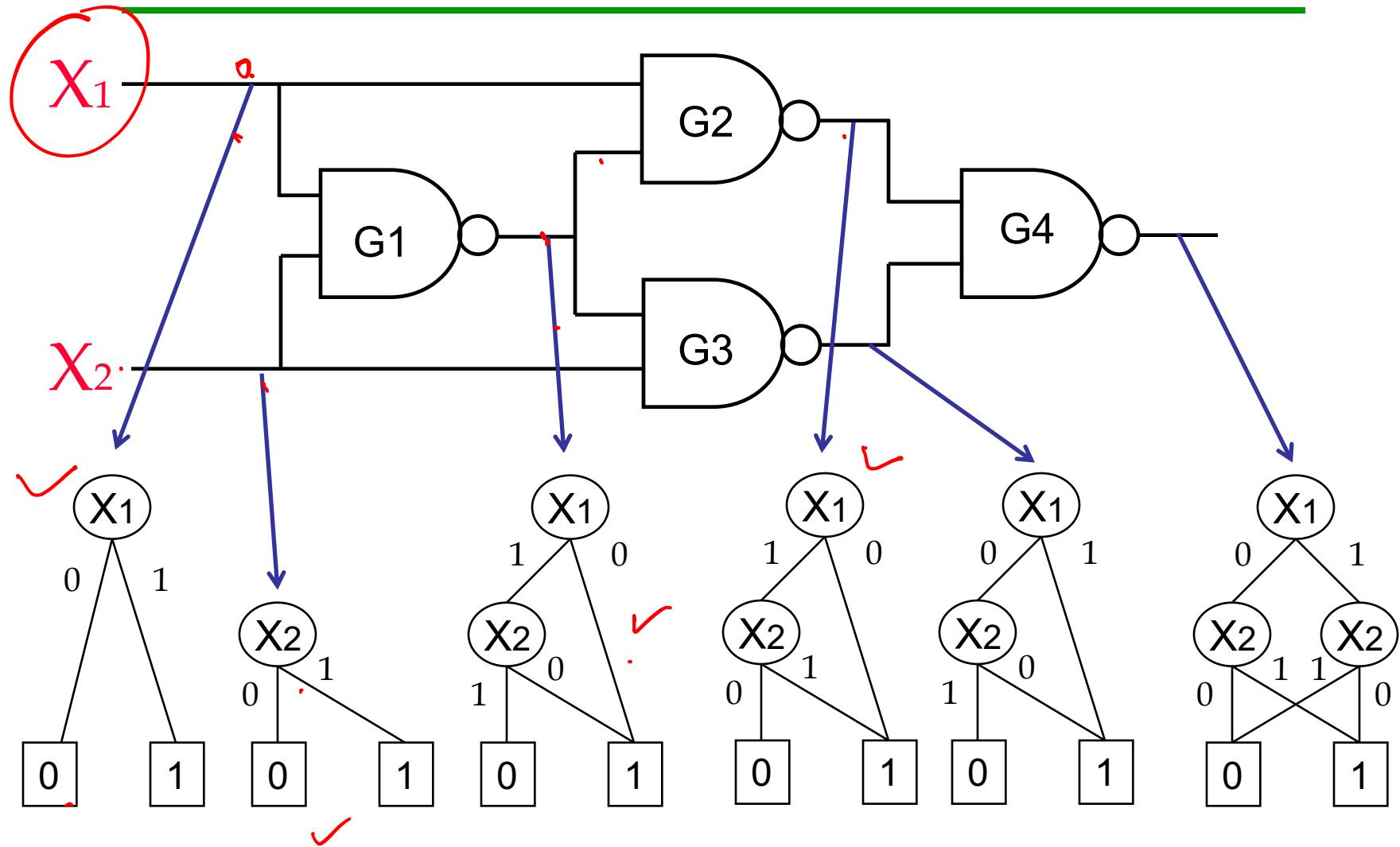


# Operations with BDD: Example

$x_1 < x_2$



# From Circuits to BDD

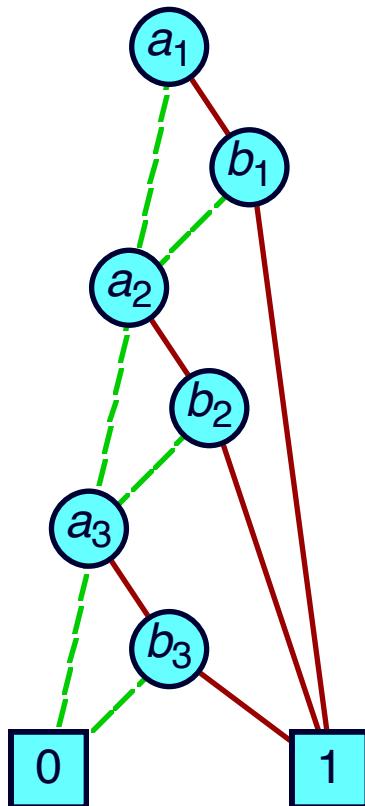


# Effect of Variable Ordering

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

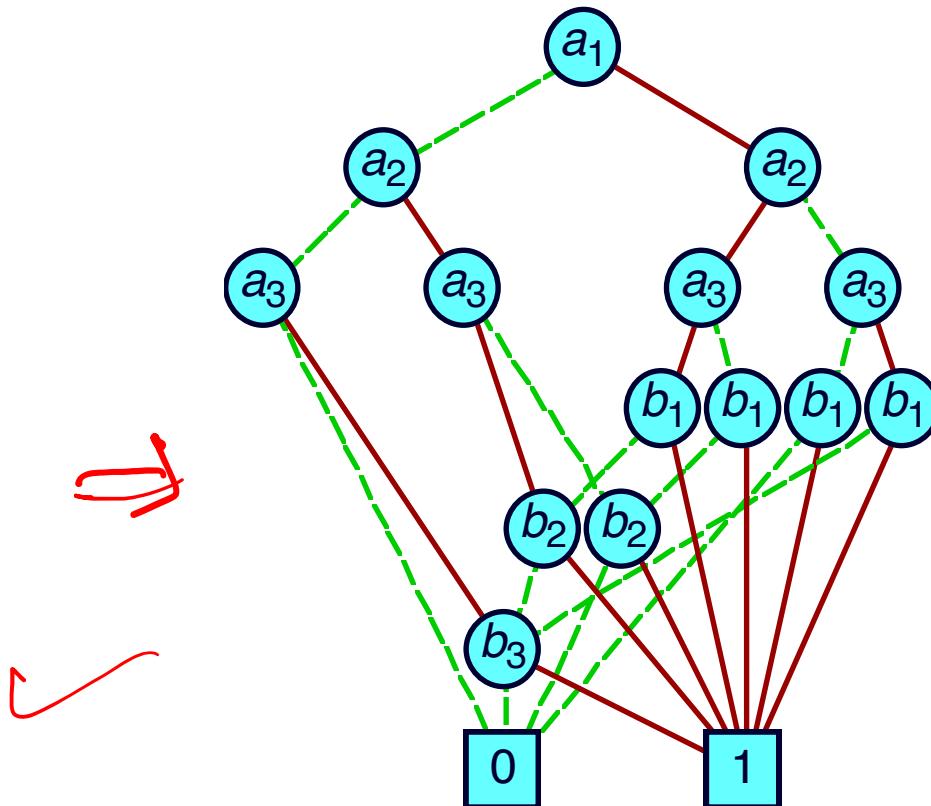
$a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$

Good Ordering



Linear Growth

Bad Ordering



Exponential Growth



# Selecting Good Variable Ordering

---

- Intractable Problem
  - Even when problem represented as OBDD
    - i.e., to find optimum improvement to current ordering
- Application-Based Heuristics
  - Exploit characteristics of application
  - e.g., Ordering for functions of combinational circuit
    - Traverse circuit graph depth-first from outputs to inputs
    - Assign variables to primary inputs in order encountered



# Selecting Good Variable Ordering

---

## 🟡 Static Ordering

- Fan In Heuristic
- Weight Heuristic

## 🟡 Dynamic Ordering

- Variable Swap
- Window Permutation
- Sifting



# Dynamic Variable Reordering

---



Richard Rudell, Synopsys

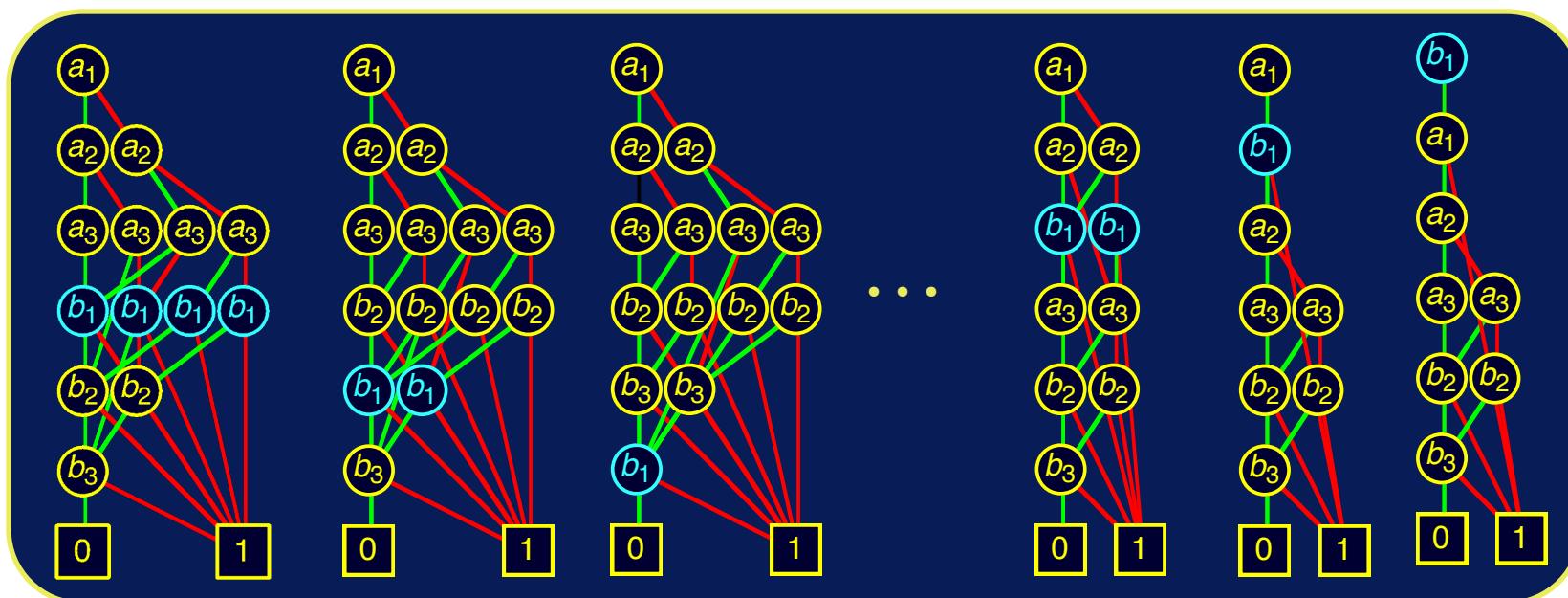
- 🟡 Periodically Attempt to Improve Ordering for All BDDs
  - ❖ Move each variable through ordering to find its best location
- 🟡 Has Proved Very Successful
  - ❖ Time consuming but effective



# Dynamic Reordering By Sifting

- Choose candidate variable
- Try all positions in variable ordering
  - Repeatedly swap with adjacent variable
- Move to best position found

Best Choices



# ROBDD Sizes & Variable Ordering

---

- Bad News 💣
  - Finding optimal variable ordering NP-Hard
  - Some functions have exponential BDD size for all orders e.g. multiplier
- Good News 😊
  - Many functions/tasks have reasonable size ROBDDs
  - Algorithms remain practical up to 1,000,000 node OBDDs
  - Heuristic ordering methods generally satisfactory
- What works in Practice 🤝
  - Application-specific heuristics e.g. DFS-based ordering for combinational circuits
  - Dynamic ordering based on variable sifting (*R. Rudell*)

WXFRHU



# Thank You



# Logic Optimization

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 9 (24 January 2022)

**CADSL**

## Representations

- ① Truth Table
- ② Sum of min terms
- ③ Product of Max terms

UNIQUE  
↑  
Comoni  
cat.

## Graphical.

- ④ Binary Decision Diagram (BDD) ← Canonical
- ⑤ Reed Muller Representation - canonical.  
= (AND and XOR)
- ⑥ And Inverter Graph (AIG) ↗ not canonical  
=  $\overline{D} \cdot D$   
ABC ↗  
T
- ⑦ SOP
- ⑧ POS ↗ non-canonical
- ⑨ Hybrid 2 CADSL



# Function Minimization

$$\left\{ \begin{array}{l} \text{SOP} \rightarrow \text{2level.} \\ \text{POS} \Rightarrow \text{2level} \end{array} \right. \quad \begin{array}{l} \text{AND} \rightarrow \text{OR} \oplus \\ \text{OR} \rightarrow \text{AND} \oplus \end{array} \quad \begin{array}{l} \text{NAND-NAND} \\ \text{NOR-NOR} \end{array}$$

minimize  $\rightarrow$  Boolean Algebra ✓

$\Rightarrow$  Not scalable approach ]

✓ Automatic ✓

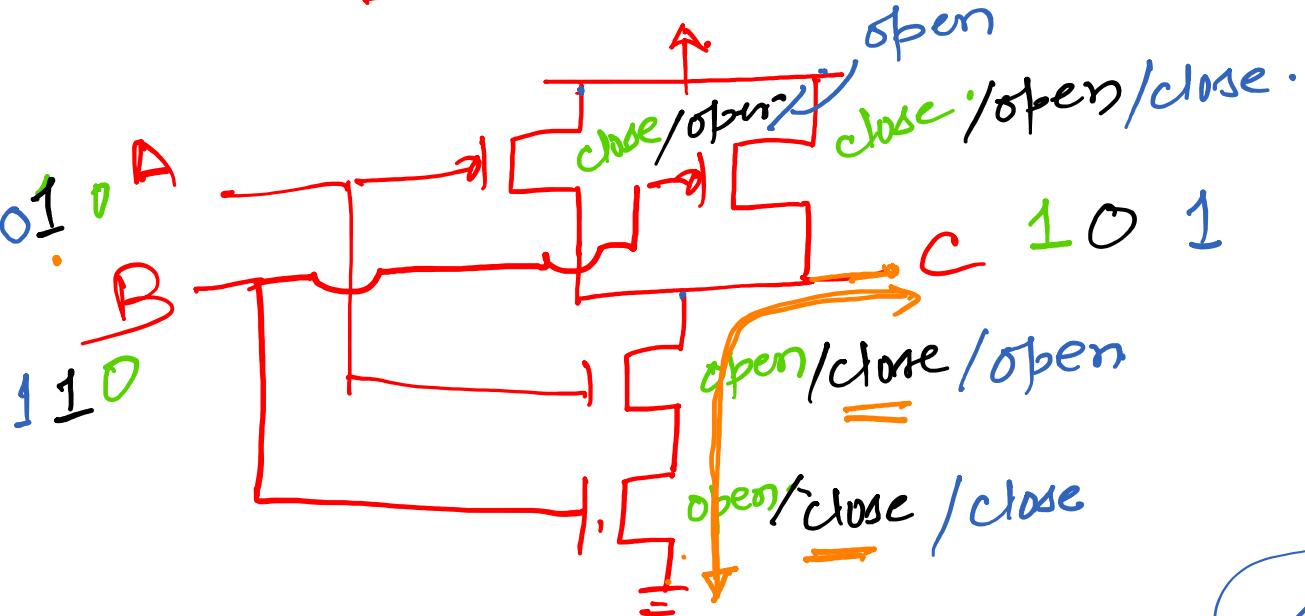
Algorithmic approach.

① Graphical Method. (K-Map)

② Tabular Method. (QM.)



## NAND



0 0 - 1
0 1 - 1
1 0 - 1
1 1 - 0

NAND =

Delay.

[delay of one switch is  $\tau$ ]

Delay of 2 input

NAND is  $2\tau$

for  $N$  input  $\underline{\text{NAND}} = \underline{N\tau}$

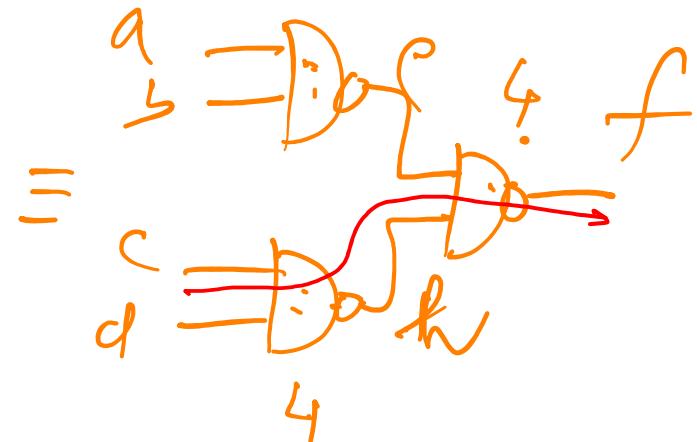
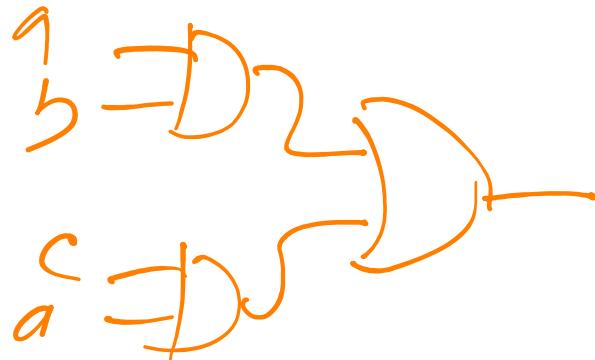
4 switches

$N \text{ input} = 2N$   
switches  
transistors



COST of  $N$  input NAND =  $2N$  ·  
delay of  $N$  input-NAND =  $N\tau_c$ .

$$f(a,b,c,d) = \underline{\overline{ab} + \overline{cd}} \quad 4$$

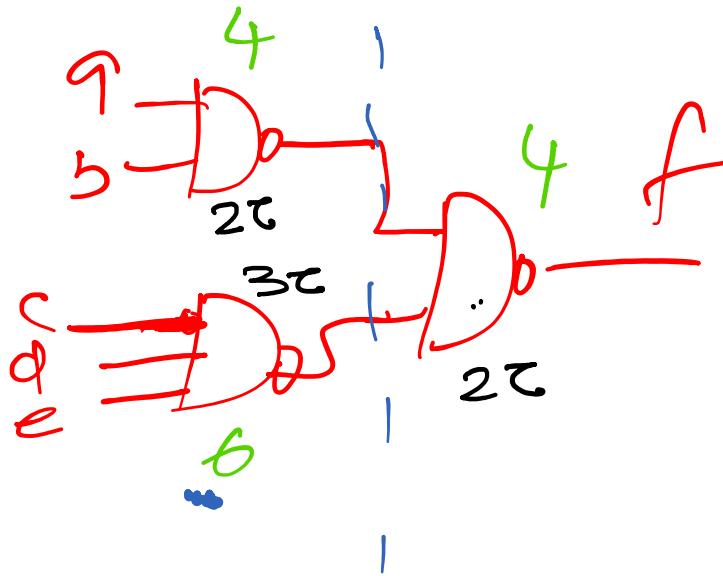


COST =  $2 \times \# \text{ literals} + 2 \times \# \text{ product terms}$  ·

~~2~~



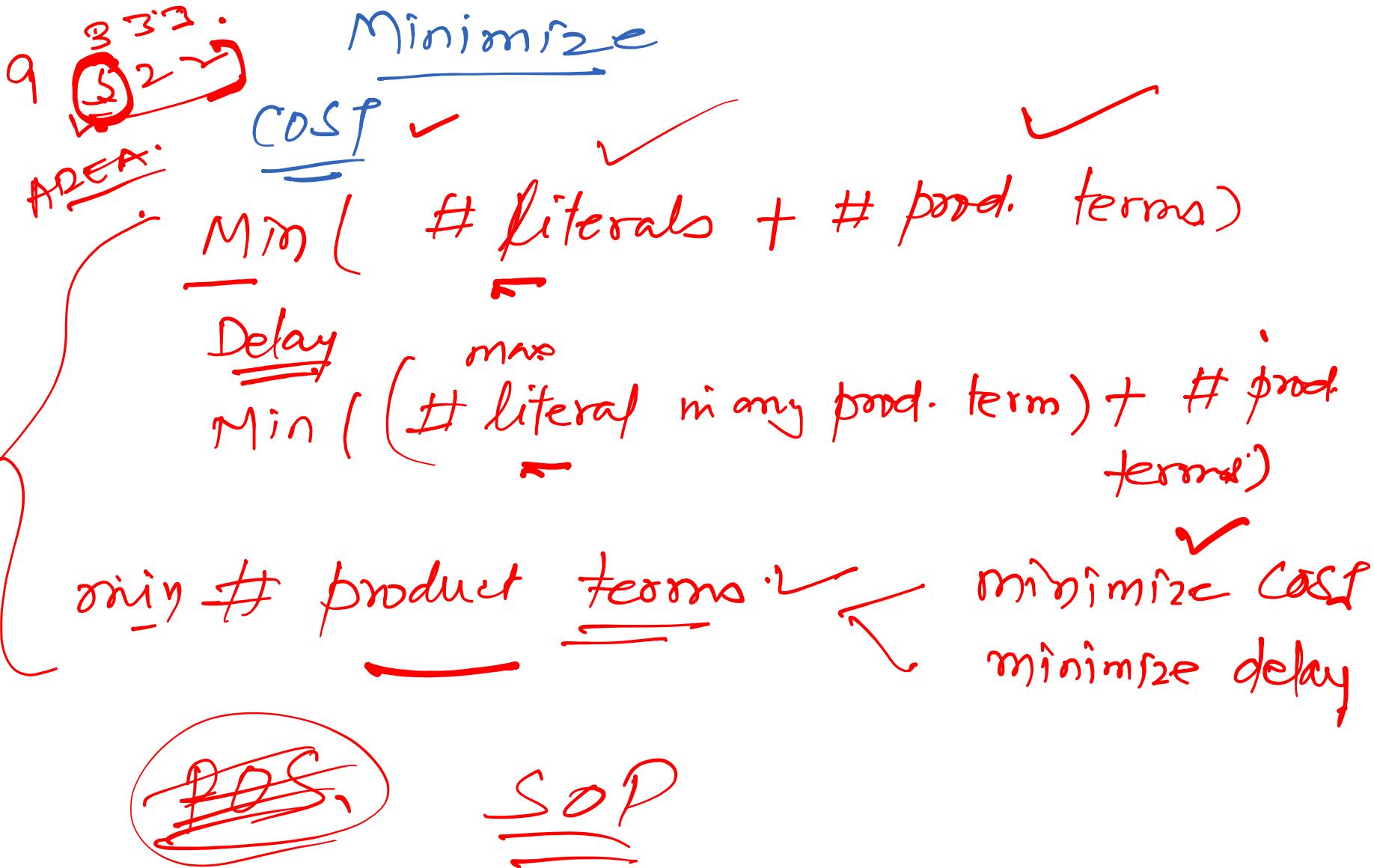
$$f(a,b,c,d,e) = \underline{ab} + \underline{cde}$$



delay = Max delay of first level + delay of second level  
~~=  $\{10, 2\tau\} + \max\{2\tau, 3\tau\}$~~  +  $2\tau$ . =  $5\tau$

(Max literals in any prod. term + # prod. term)





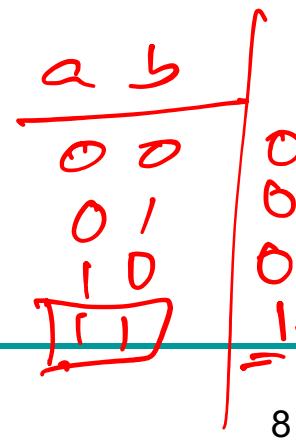
# Logic Minimization

- Generally means
    - In SOP form:
      - Minimize number of products (reduce gates) and
      - Minimize literals (reduce gate inputs)
    - In POS form:
      - Minimize number of sums (reduce gates) and
      - Minimize literals (reduce gate inputs)

$$a=1 \quad b=1 \quad \Rightarrow f=1 \quad f(1,1)$$

~~$a \cdot b \rightarrow f$~~

impllicants



# Product or Implicant or Cube

- Any set of literals ANDed together.
- Minterm is a special case where all variables are present. It is the largest product.
- A minterm is also called a 0-implicant of 0-cube.
- A (1-implicant) or 1-cube is a product with one variable eliminated:
  - Obtained by combining two adjacent 0-cubes
  - $\overline{ABCD} + \overline{ABC} \cdot \overline{D} = ABC(D + \overline{D}) = ABC$

$$ABC \rightarrow f$$



# Function Minimization

---

$$\begin{aligned}f(a,b) &= ab + a\bar{b} \\&= a \cdot \underbrace{(b + \bar{b})}_{1} = a\end{aligned}$$

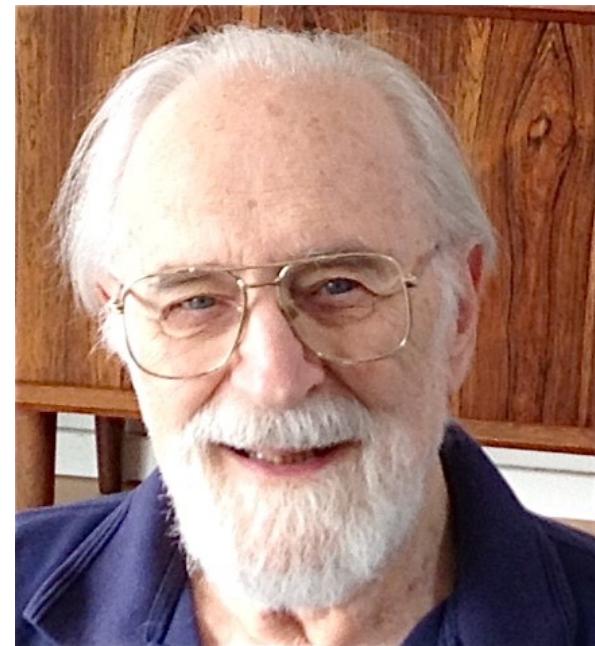


# Graphical Method: Karnaugh Map

---

## Maurice Karnaugh

- American Physicist
- Bell Lab (1952 – 66)
- Developed K-Map in 1954



Maurice Karnaugh  
Born: 4 October 1924

Karnaugh, Maurice (November 1953), "The Map Method for Synthesis of Combinational Logic Circuits". *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*. 72(5): 593–599



# Function Minimization: K-Map

$$f(a,b) = \underline{ab+a\bar{b}} = a \cdot (\underline{b+\bar{b}}) = a \quad a \rightarrow f$$

$$\Rightarrow ab + a\bar{b} \rightarrow a \leq$$

$a$	$\begin{array}{ c c }\hline 0 & 1 \\ \hline 1 & 0 \\ \hline \end{array}$
$\bar{a}$	$\begin{array}{ c c }\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$

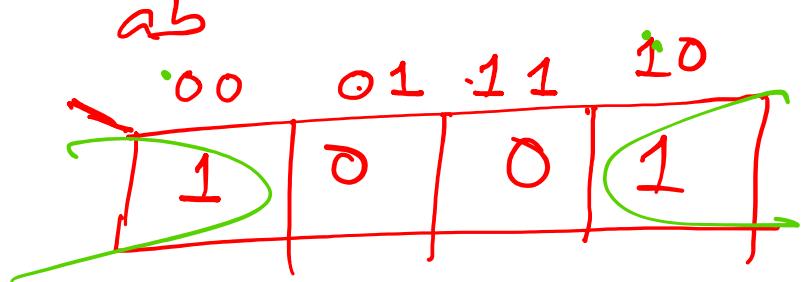
$$f(\underline{a}) = a \\ f(\bar{a}) = \bar{a}$$

$$a \quad \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 1 \\ \hline \end{array} \quad f=1$$

$a$	$b$	$c$
0	0	1
0	1	0
1	0	0
1	1	1

$$a \quad b \quad c \\ 0 \quad 0 \quad 0 \\ 0 \quad 0 \quad 1 \\ 0 \quad 1 \quad 0 \\ 1 \quad 0 \quad 0 \\ 1 \quad 1 \quad 1$$

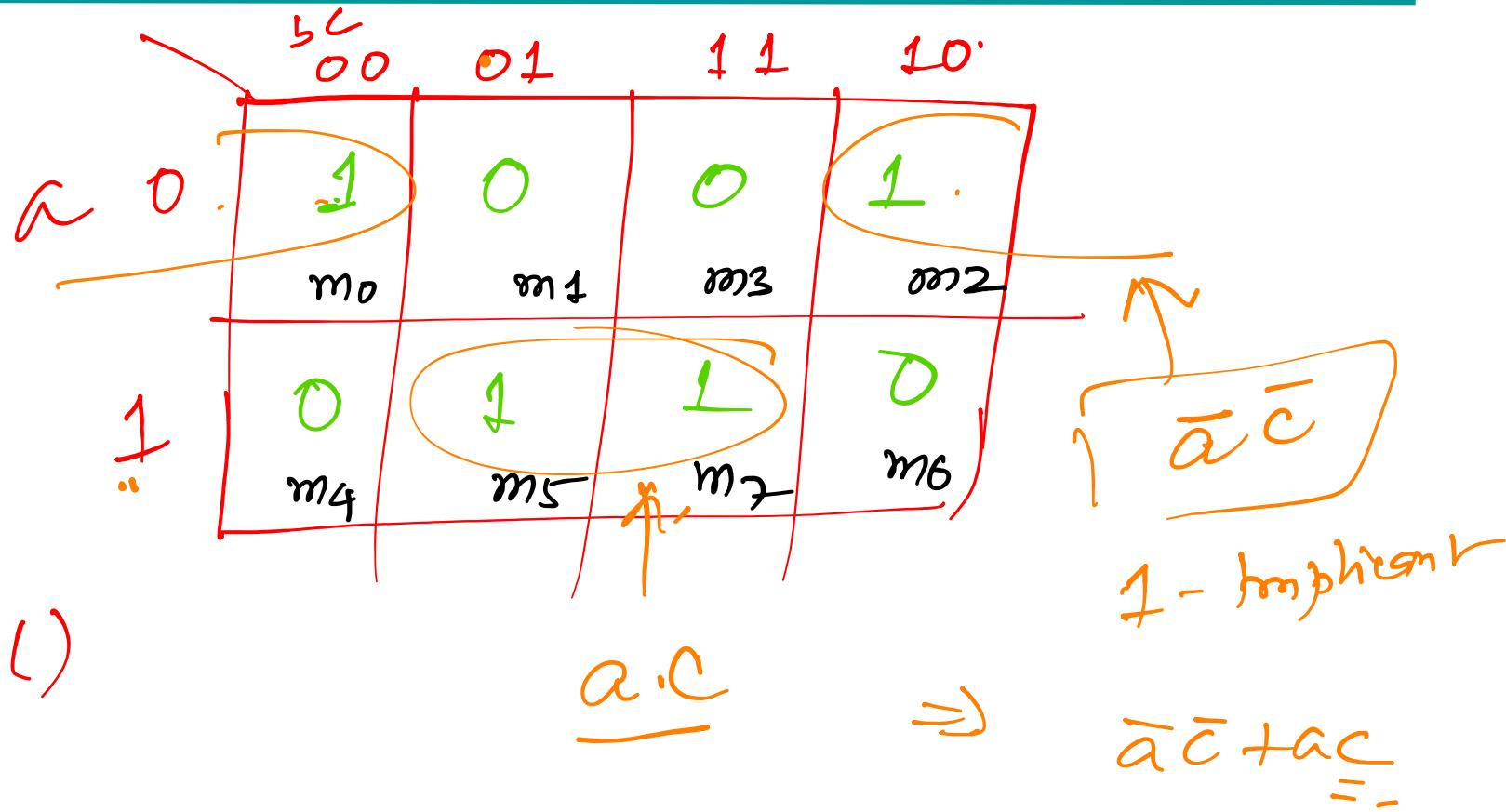
$f(a,b,c) = \bar{a}\bar{b} + a\bar{b} + \bar{a}b + ab = \bar{b}$



$$f(a,b) = \bar{b}$$



# Function Minimization: K-Map



$f(a,b,c)$

$\underline{a \cdot c}$

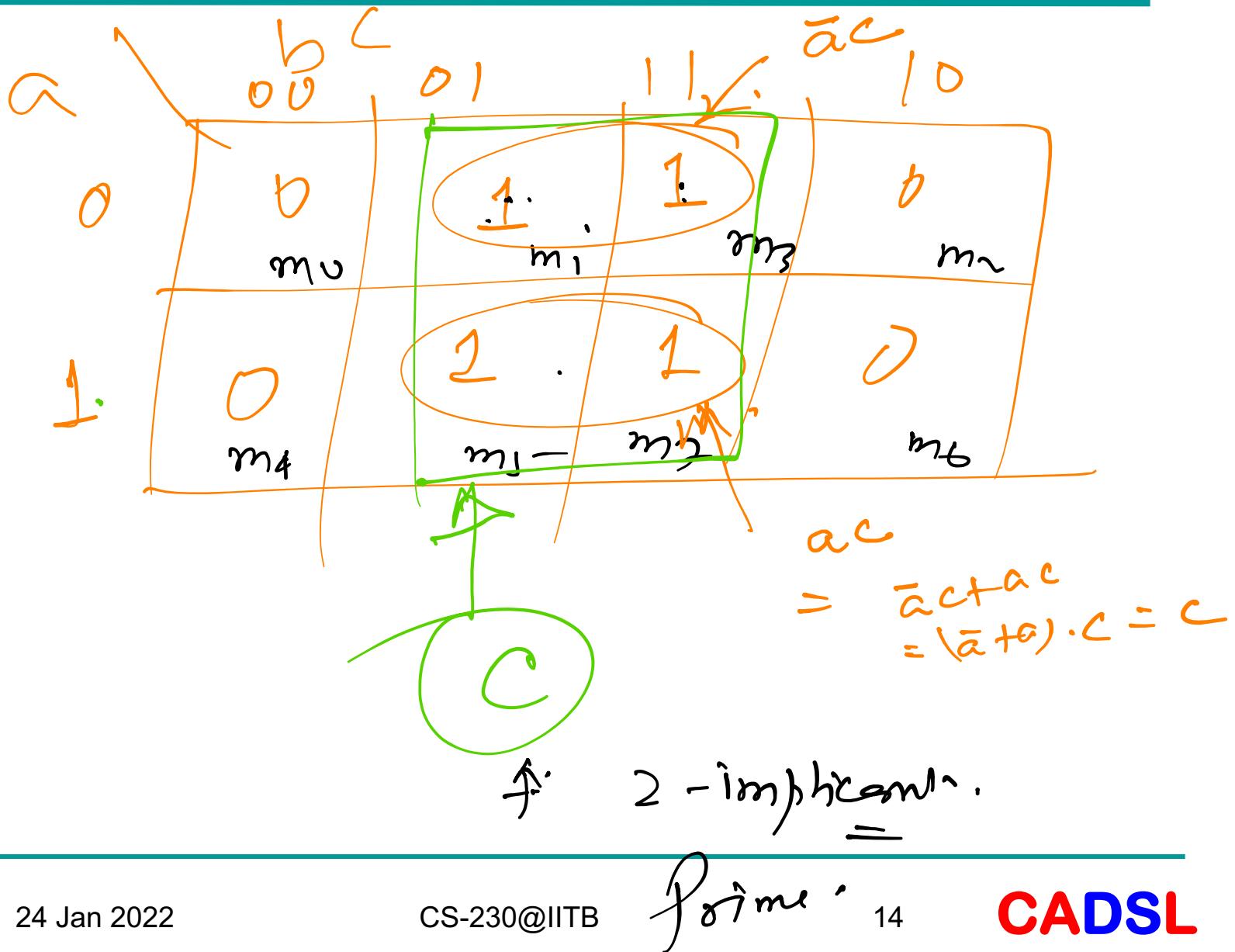
$\Rightarrow$

1 - Implicant

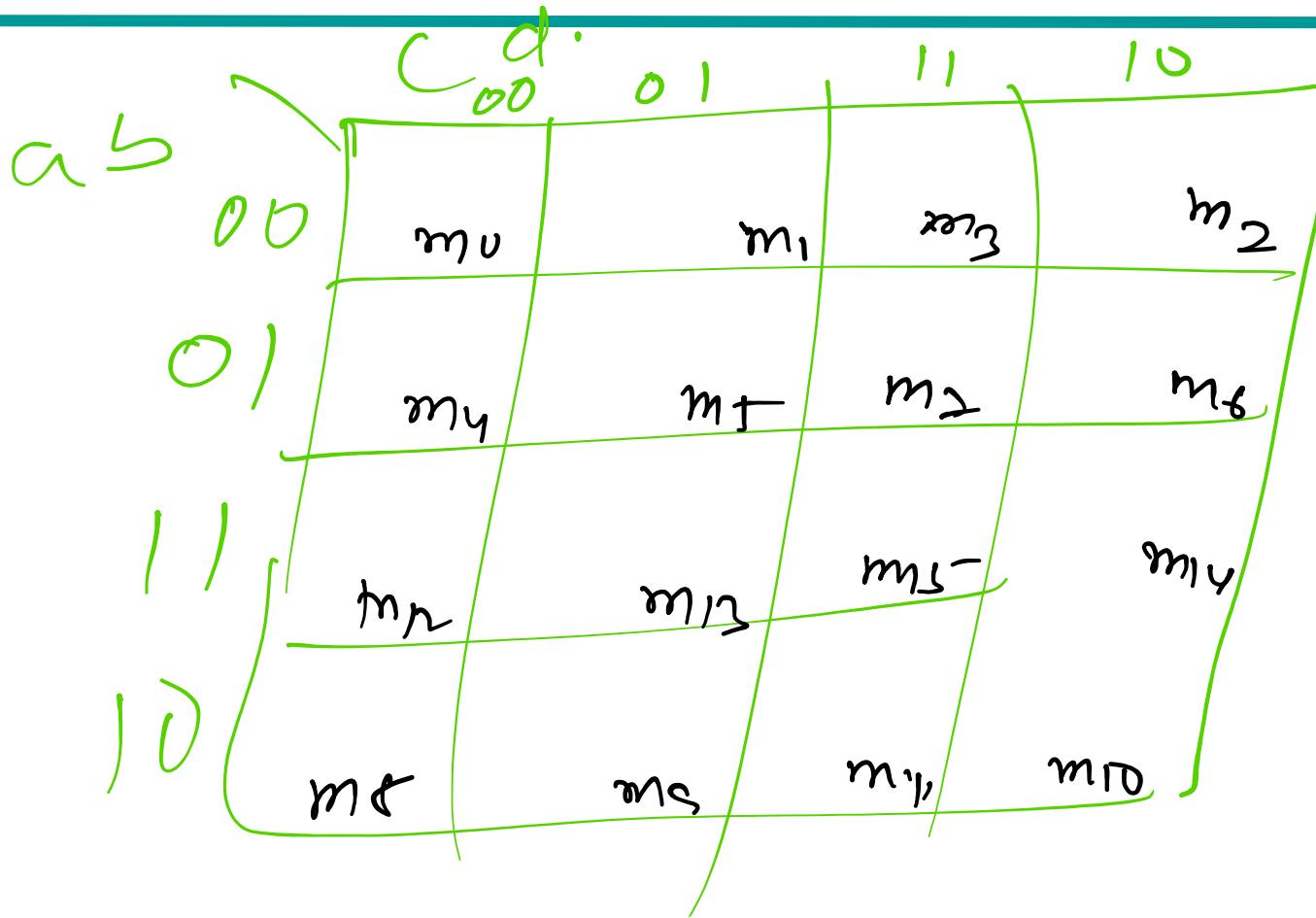
$\bar{a} \bar{c} + a c$



# Function Minimization: K-Map

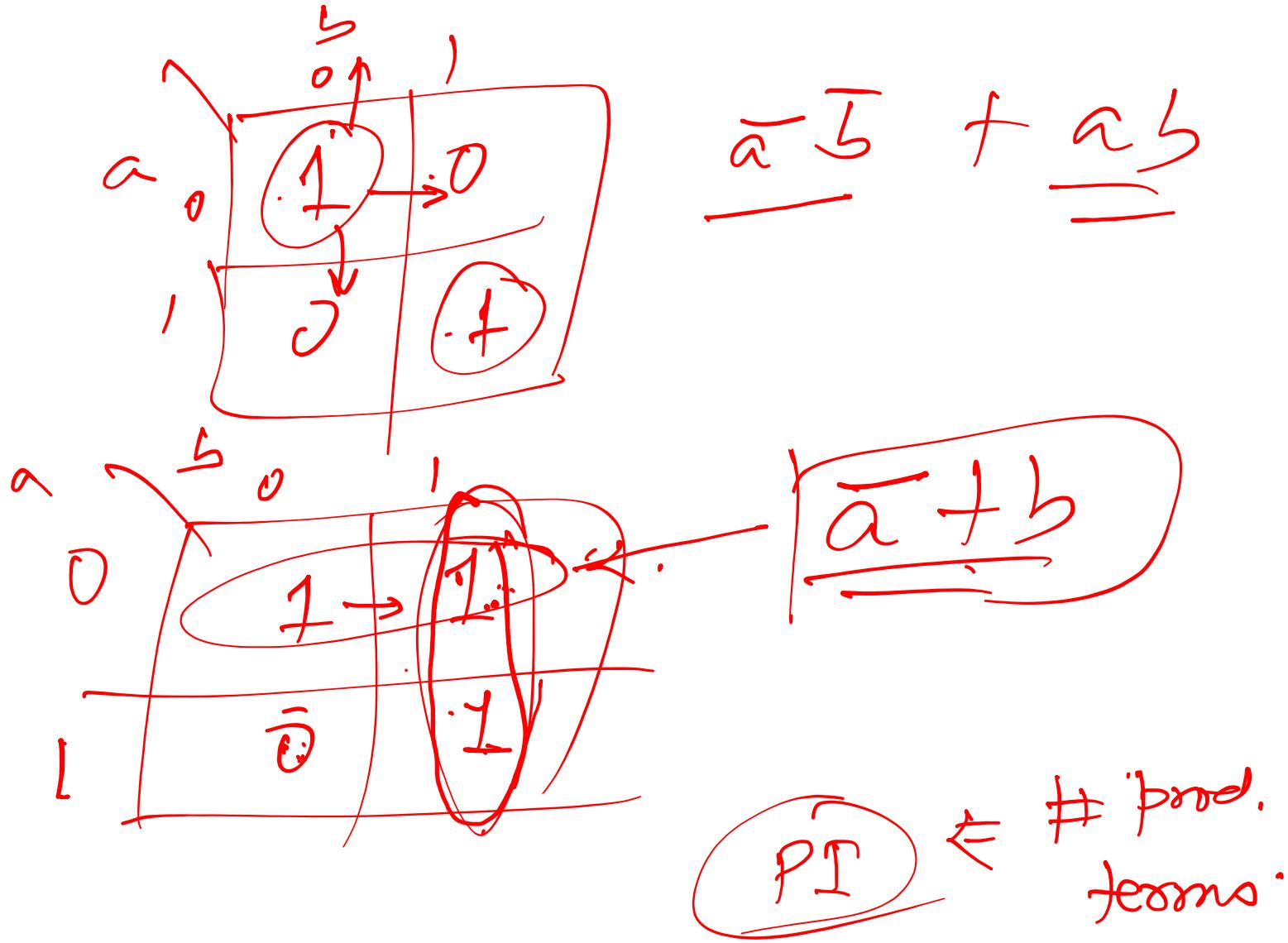


# Function Minimization: K-Map

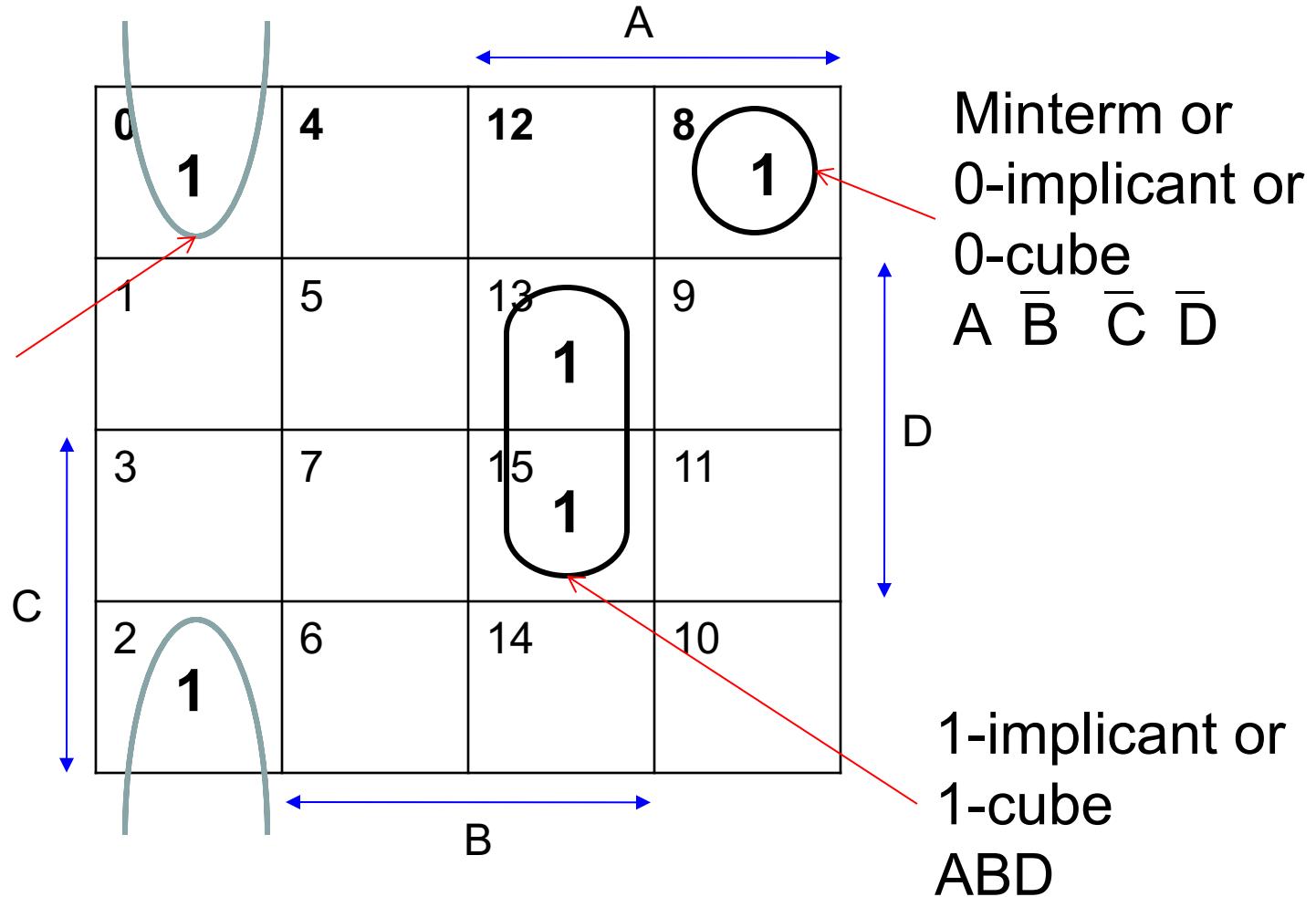


PRIME    IMPLIC.





# Cubes (Implicants) of 4 Variables



# Thank You



# Logic Optimization

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 9 (24 January 2022)

**CADSL**

## K-Map

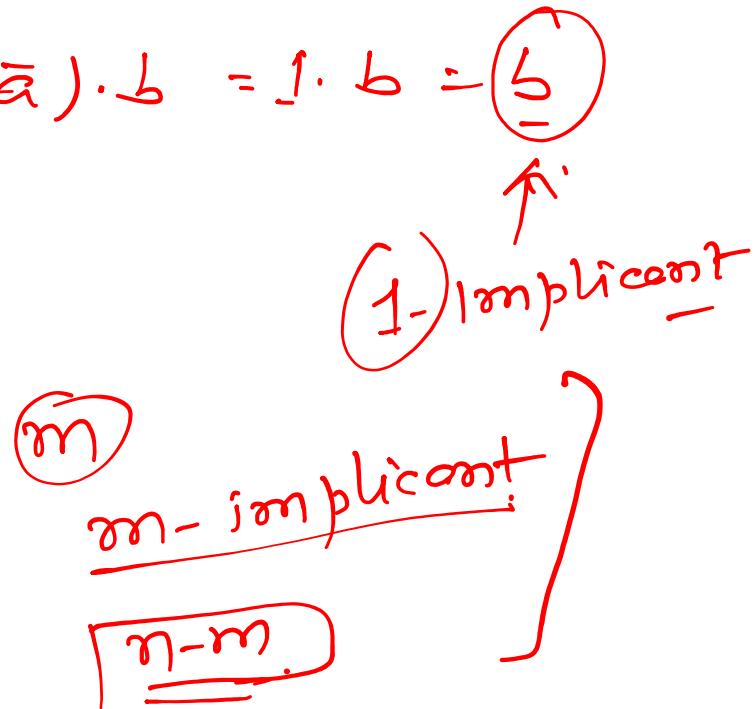
## Graphical

$$f(a,b) = a \cdot b + \bar{a} \cdot b = (a + \bar{a}) \cdot b = 1 \cdot b = b$$

$\uparrow$        $\uparrow$

$\checkmark$ -implicants      0-implicants

$$f(x_1, x_2, \dots, x_n)$$

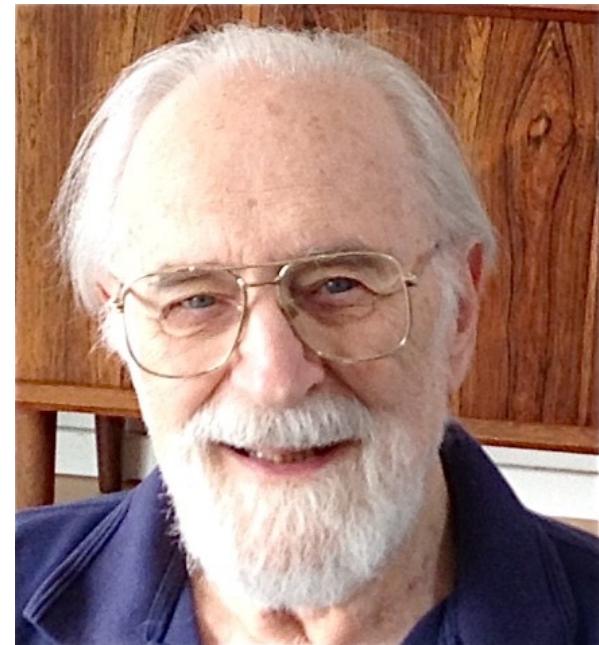


# Graphical Method: Karnaugh Map

---

Maurice Karnaugh

- American Physicist
- Bell Lab (1952 – 66)
- Developed K-Map in 1954



Maurice Karnaugh  
Born: 4 October 1924

Karnaugh, Maurice (November 1953), "[The Map Method for Synthesis of Combinational Logic Circuits](#)". *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5): 593–599

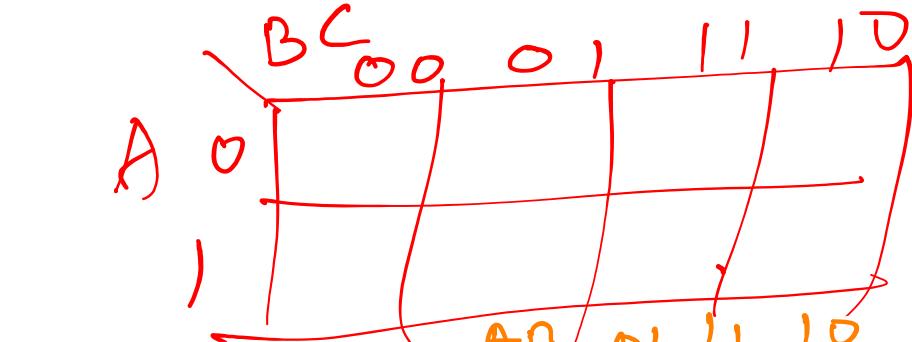


# Function Minimization: K-Map

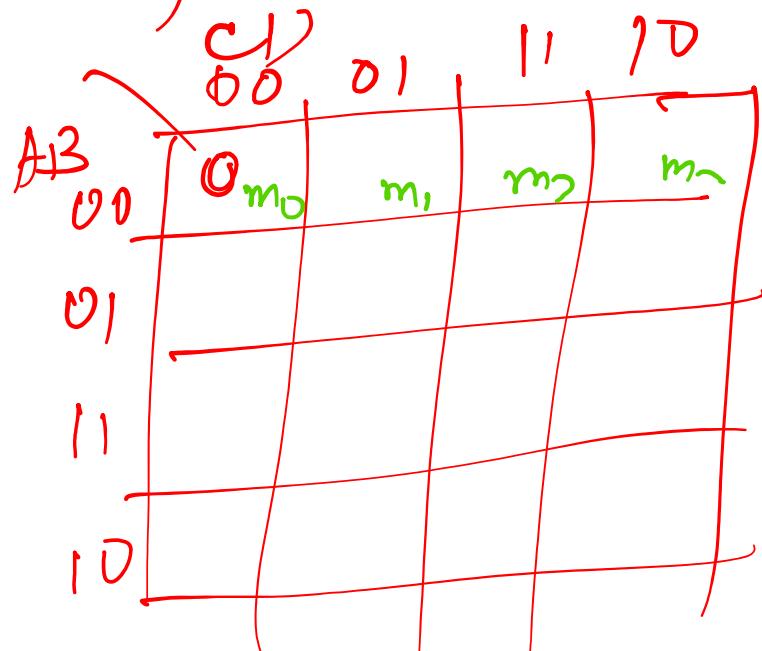
1-variable

2-variable, AB.

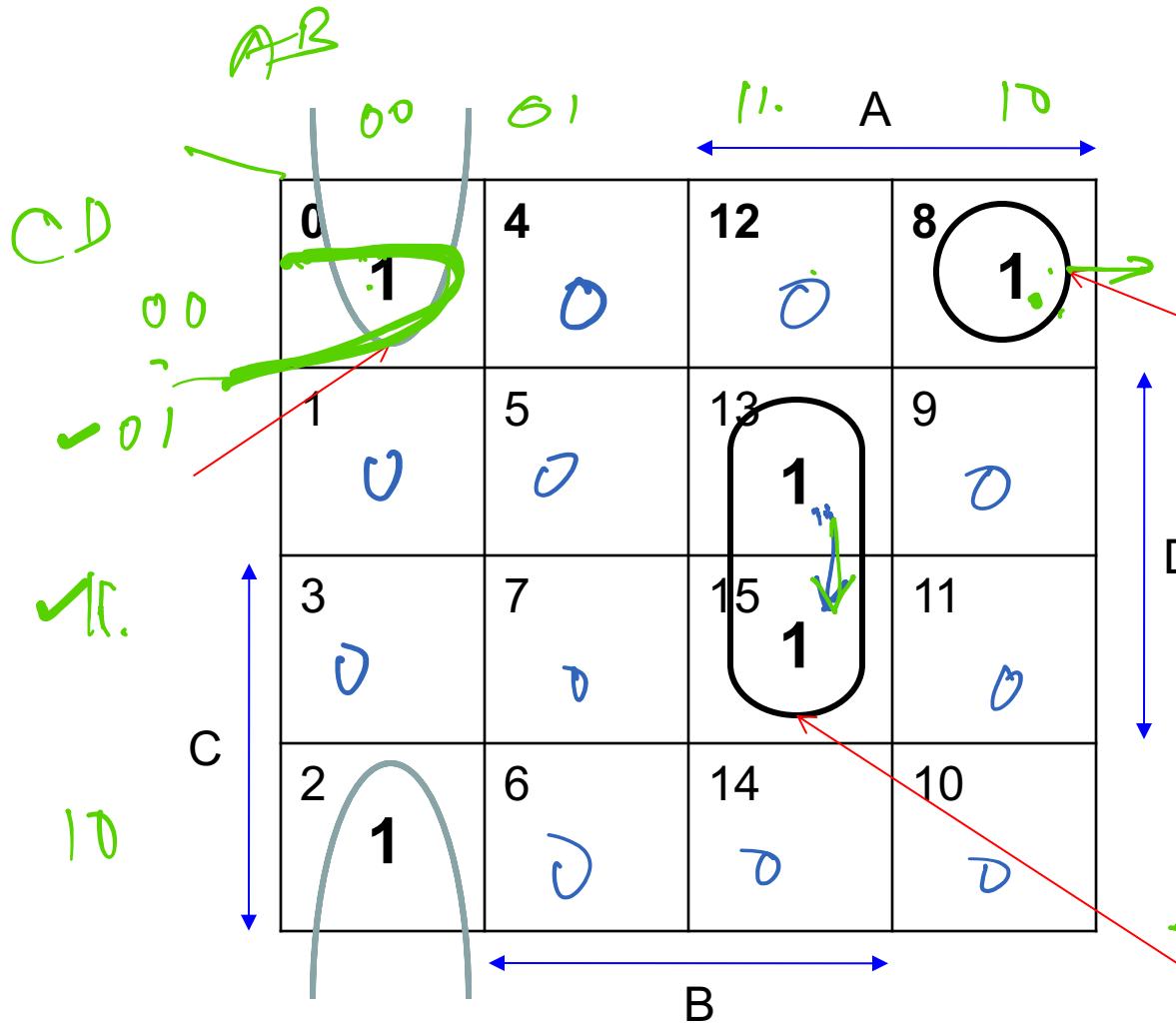
3-variable



$$f(A, B, C, D)$$



# Cubes (Implicants) of 4 Variables



Minterm or  
0-implicant or  
0-cube

$A \bar{B} \bar{C} \bar{D}$

1-implicant

$\bar{B} \bar{C} \bar{D}$

1-implicant or  
1-cube

ABD



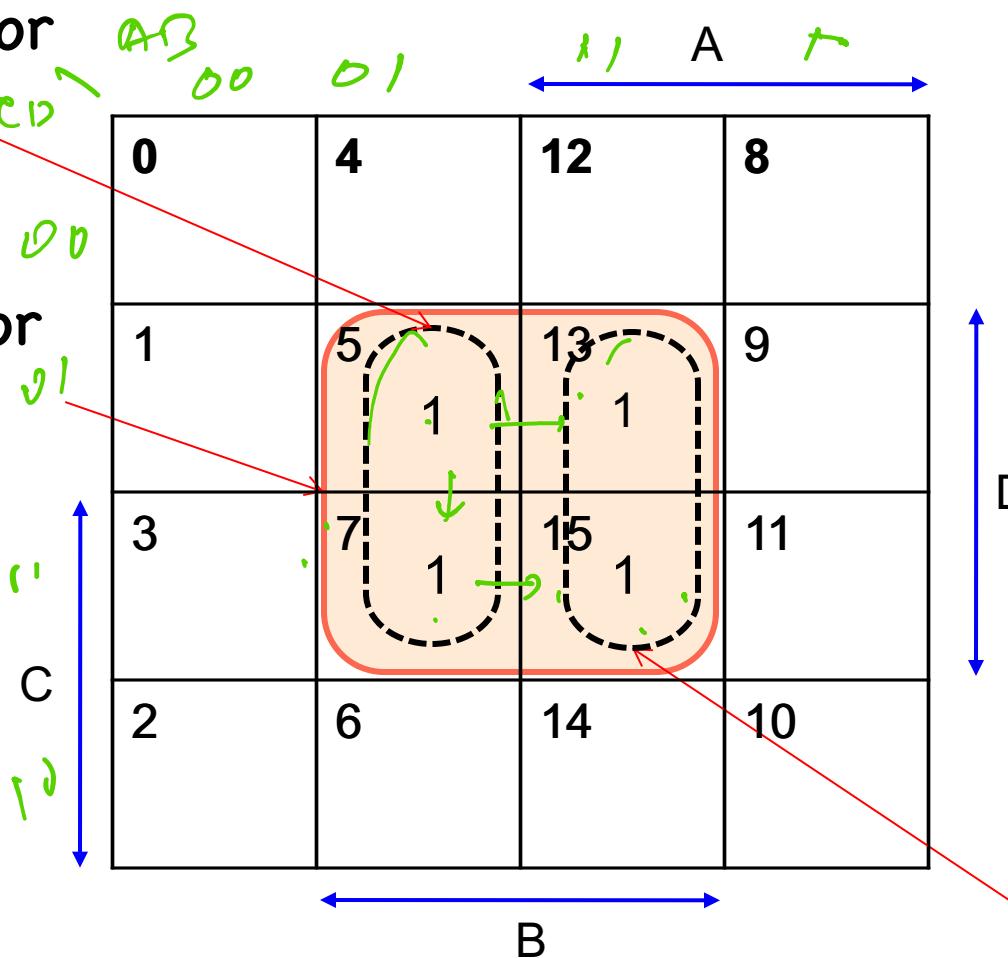
# Growing Cubes, Reducing Products

1-implicant or  
1-cube

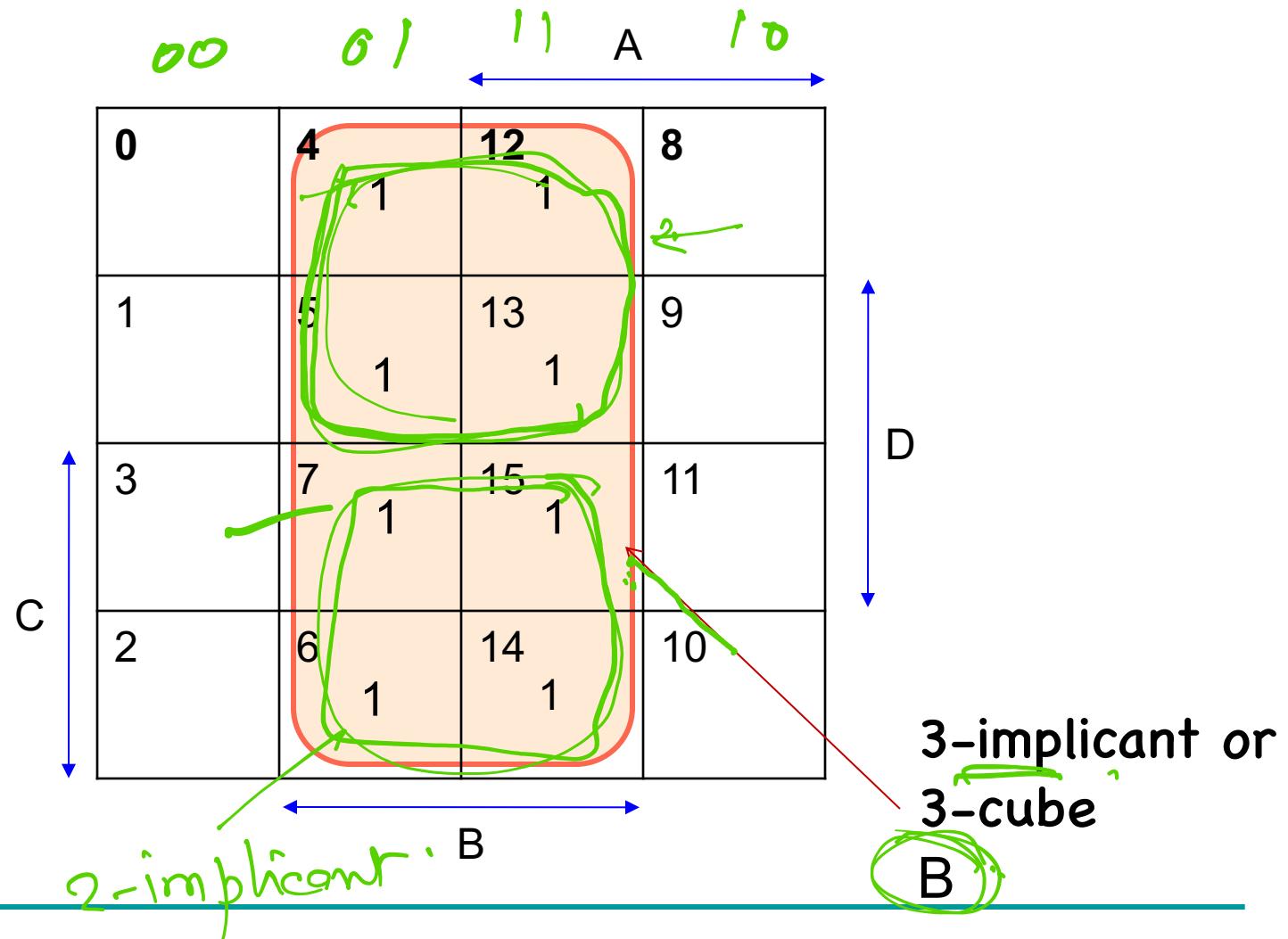
$\bar{A}B D$

2-implicant or  
2-cube

$BD$



# Largest Cubes or Smallest Products



# Implication and Covering

---

- A larger cube **covers** a smaller cube if all minterms of the smaller cube are included in the larger cube.
- A smaller cube implies (or subsumes) a larger cube if all minterms of the smaller cube are included in the larger cube.



# Implicants of a Function

- Minterms, products, cubes that imply the function.  $\sum(3, 4, 5, 6, 7, 8, 13, 15)$

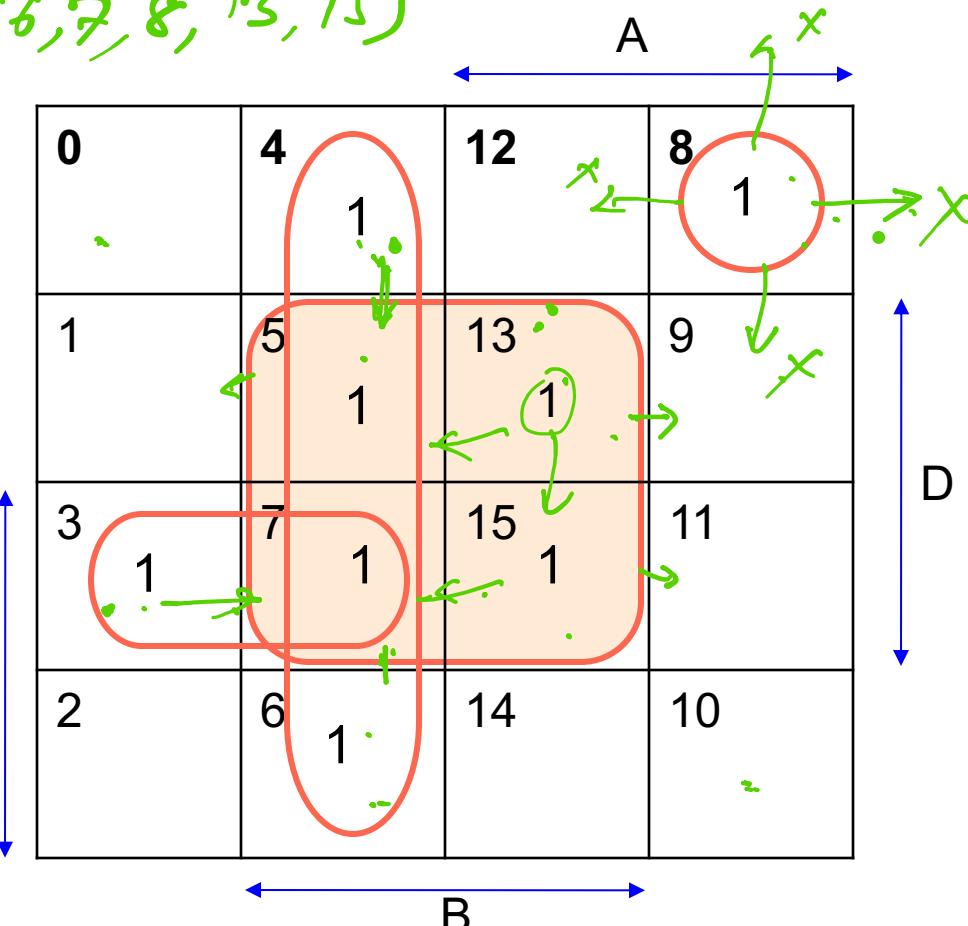
$$F = \overline{AB} + \overline{BD} + \overline{ACD} + \overline{ABC\bar{D}}$$

✓ ✓ ✓ ✓

Prime Implicant

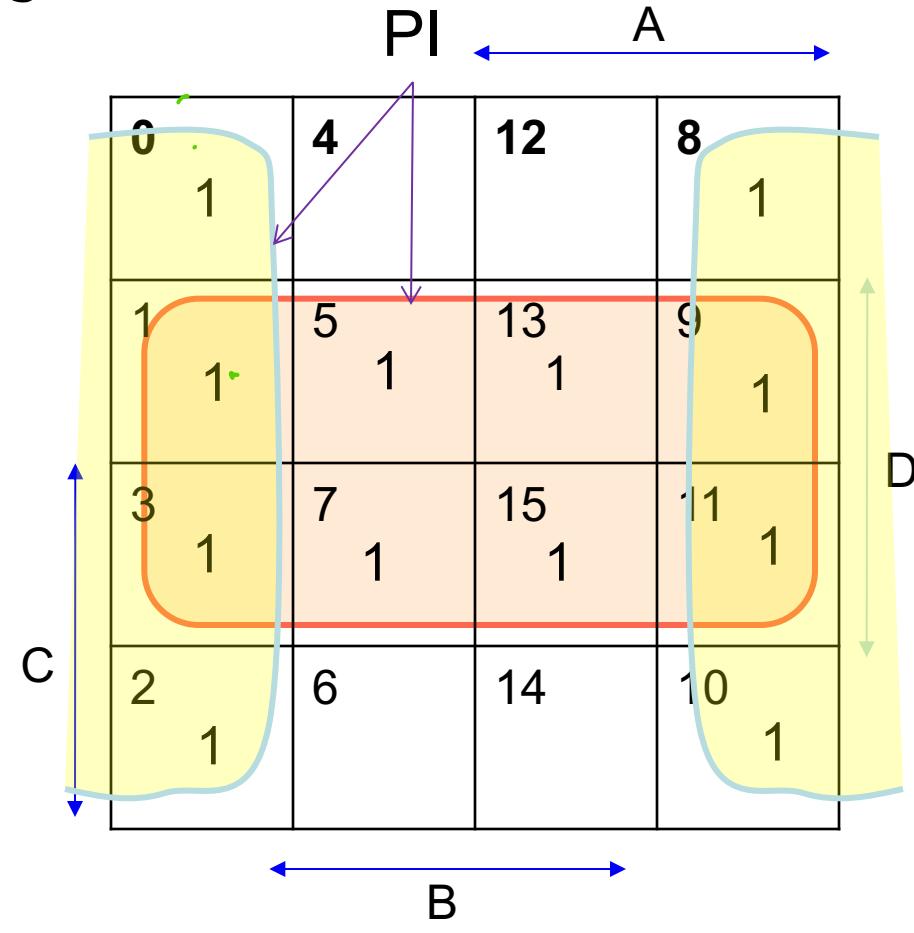
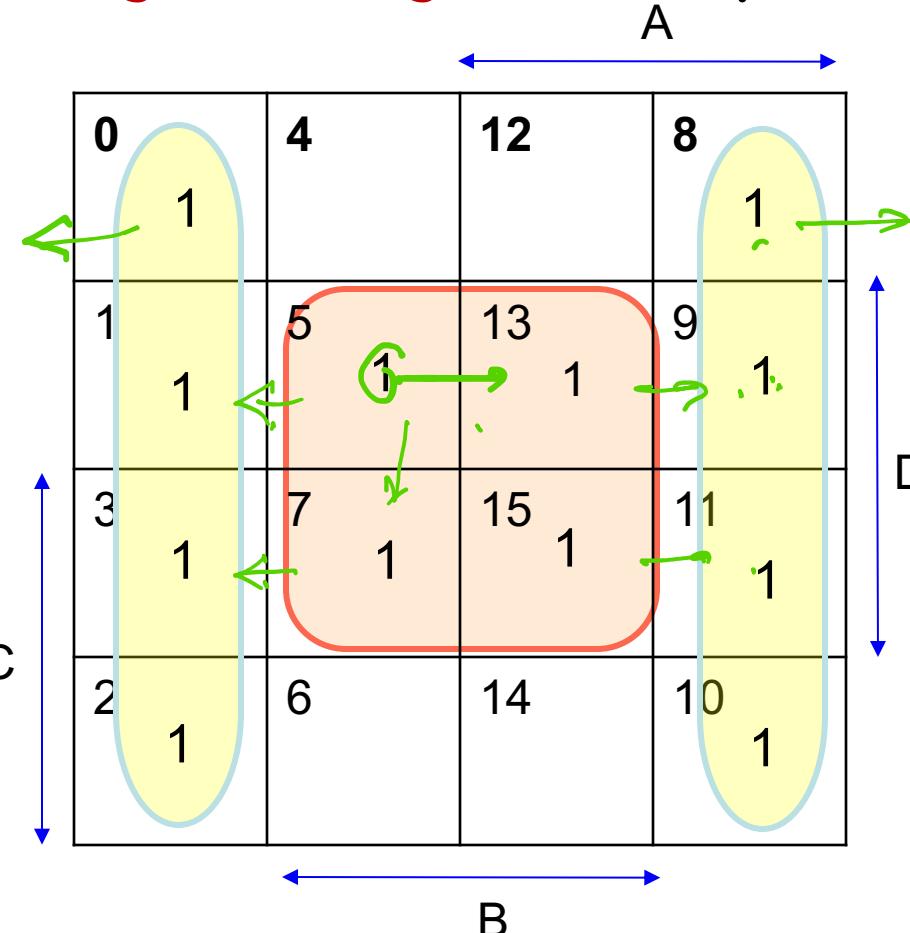
U

minimize  
literals in a  
product term.



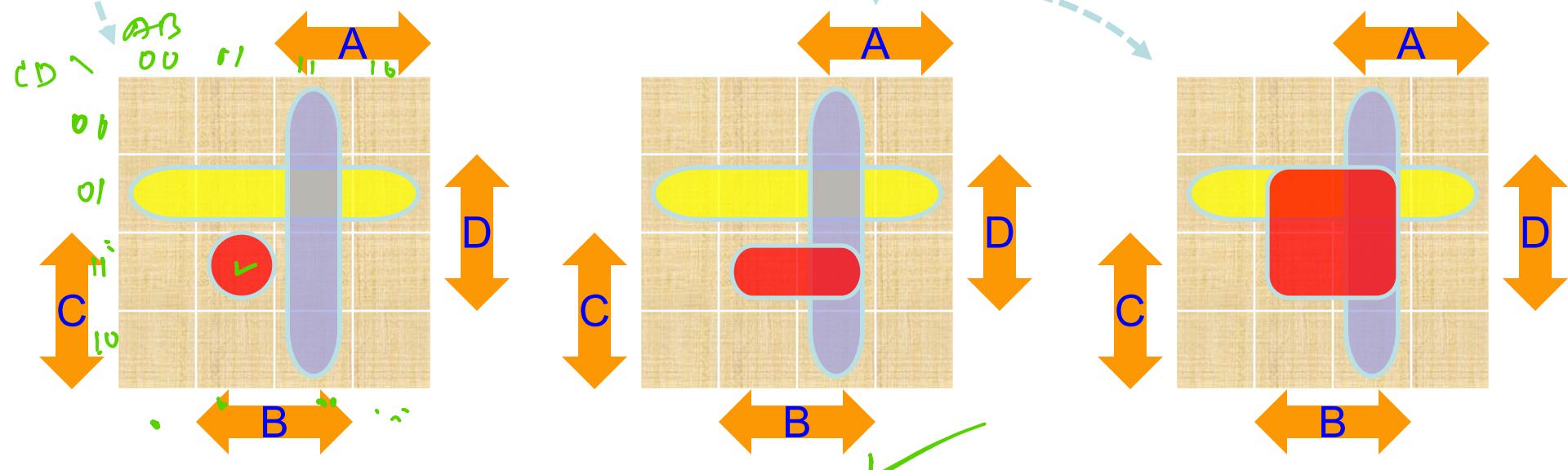
# Prime Implicant (PI)

- A cube or **implicant** of a function **that cannot grow larger by expanding into other cubes.**



# Growing Implicants to PI

- $F = \overline{AB} + \overline{CD} + \overline{ABCD}$   $\checkmark$   
 $= AB + \overline{ABCD} + BCD + \overline{CD}$   
 $= AB + BCD + \overline{CD}$   
 $= AB + BCD + \overline{CD} + BD$   
 $= AB + \overline{CD} + BD$
- $AB + \overline{ABCD} + B \cdot \overline{BCD}$   $\checkmark$   
*initial implicants*  
*consensus th.*  
*absorption th.*  
*consensus th.*  
*absorption th.*



PI

product terms

Sum

$f =$

will it be a minimum

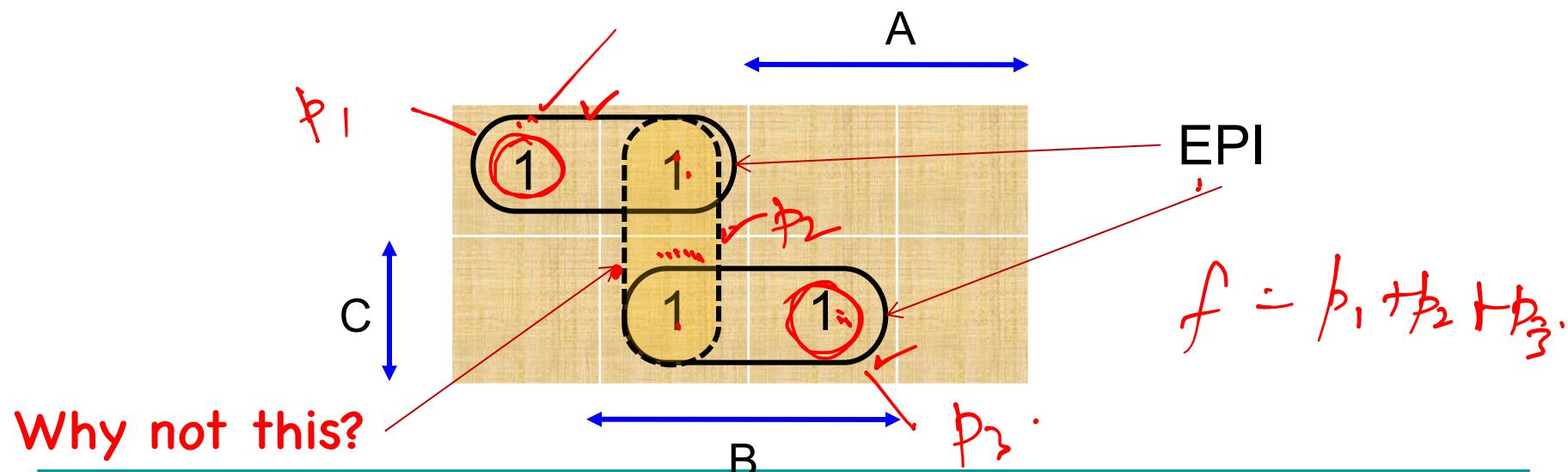
  



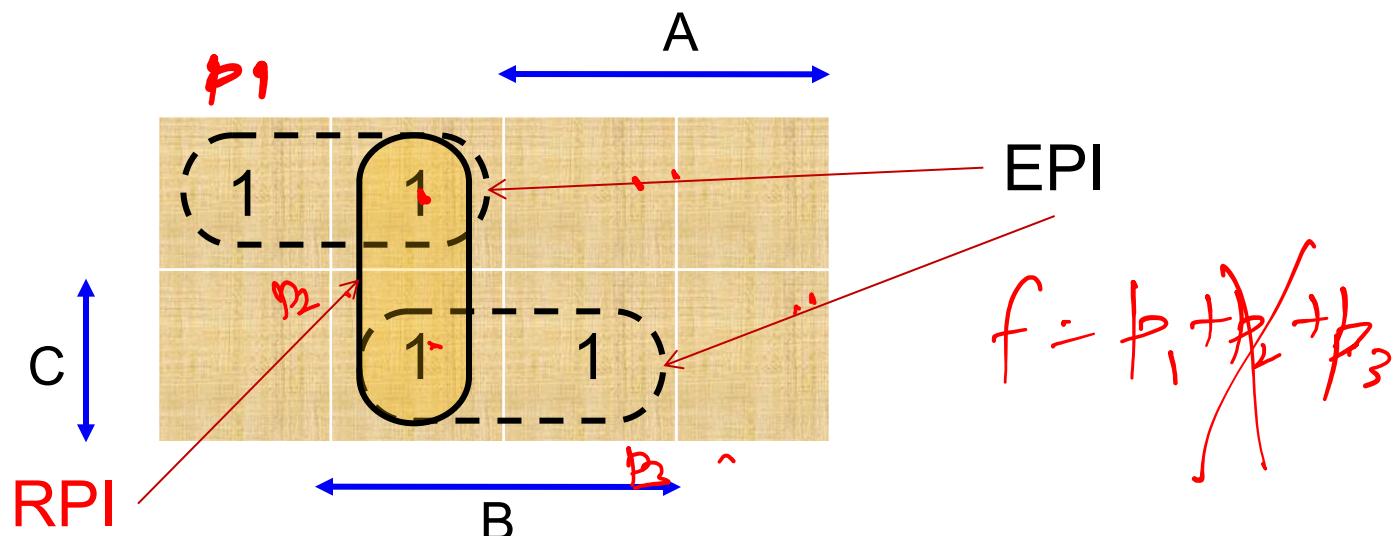
# Essential Prime Implicant (EPI)

- If among the minterms subsuming a prime implicant (PI), there is **at least one minterm that is covered by this and only this PI**, then the PI is called an essential prime implicant (EPI). ✓
- Also called essential prime cube (EPC).



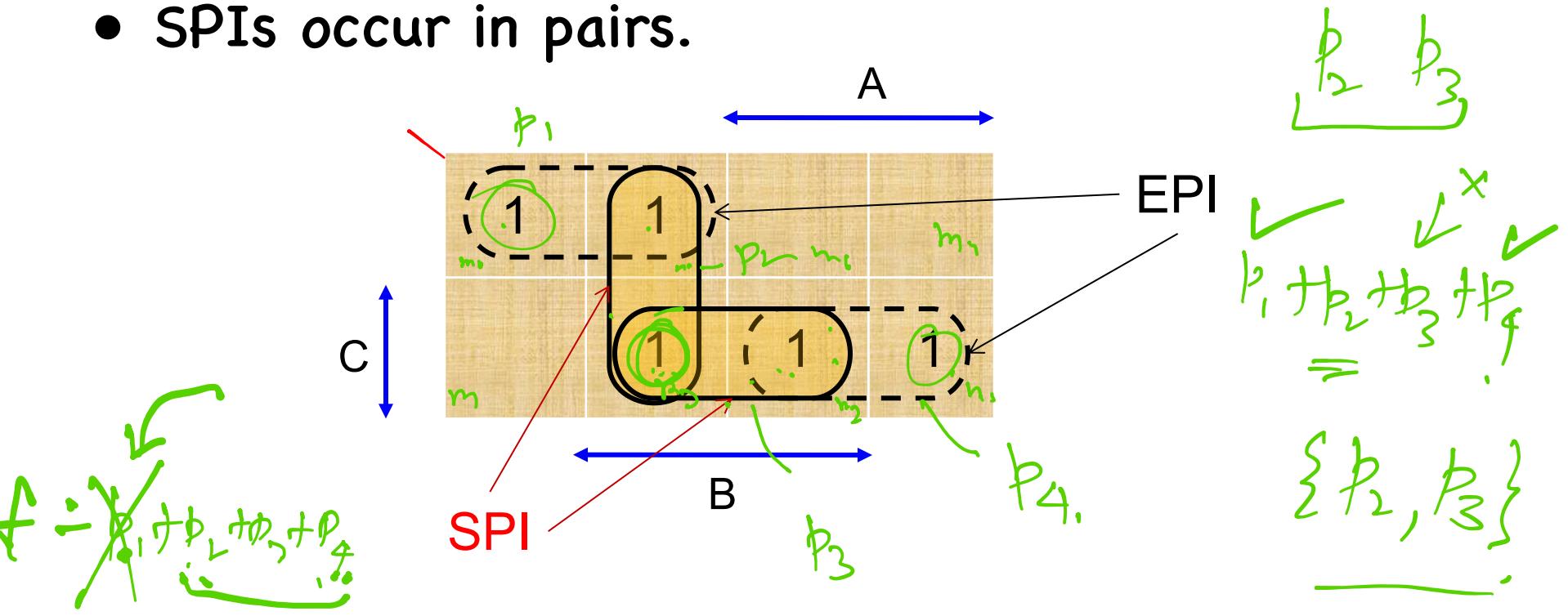
# Redundant Prime Implicant (RPI)

- If each minterm subsuming a prime implicant (PI) is also covered by other essential prime implicants, then that PI is called a redundant prime implicant (RPI).
- Also called redundant prime cube (RPC).



# Selective Prime Implicant (SPI)

- A prime implicant (PI) that is neither EPI nor RPI is called a selective prime implicant (SPI).
- Also called selective prime cube (SPC).
- SPIs occur in pairs.



# Minimum Sum of Products (MSOP)

---

- Identify all prime implicants (PI) by letting minterms and implicants grow.
- Construct MSOP with PI only :

➤ Cover all minterms

➤ Use only essential prime implicants (EPI)

➤ Use no redundant prime implicant (RPI)

➤ Use cheaper selective prime implicants (SPI)

Select min. no. of PI & ~~total~~



# Selection of Selective PI (SPI)

↙  
min number with min  
no. of literals.



# Identifying EPI

---

- Find all prime implicants.
  - From prime implicant SOP, remove a PI. ]
  - Apply **consensus theorem** to the remaining SOP.
  - If the removed PI is generated, then it is either an RPI or an SPI. :
  - If the removed PI is not generated, then it is an EPI .
- 



# Example

- PI SOP:  $F = \underline{AD} + \underline{\bar{A}\bar{C}} + \underline{\bar{C}D}$

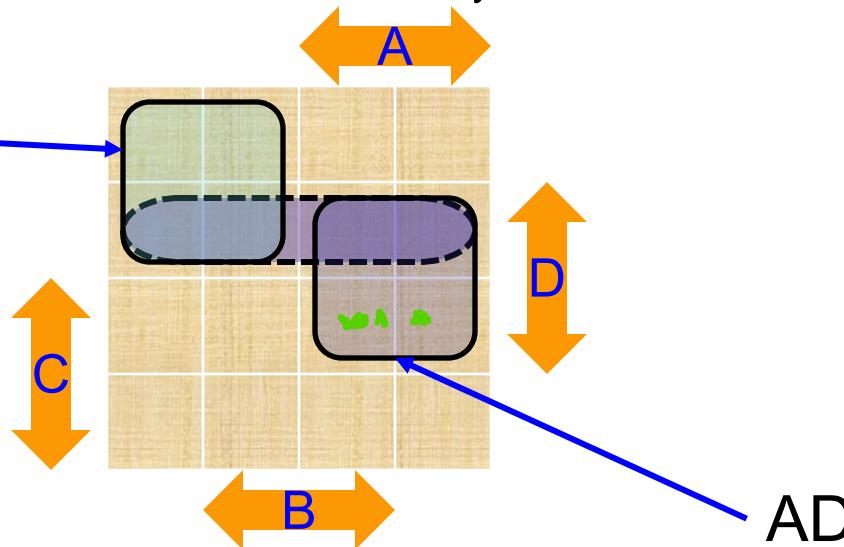
- Is AD an EPI?

$$F - \{AD\} = \underline{\bar{A}\bar{C}} + \underline{\bar{C}D}, \text{ no new PI can be generated}$$

Hence, AD is an EPI. Similarly, A  $\bar{C}$  is an EPI.



$\bar{A}\bar{C}$



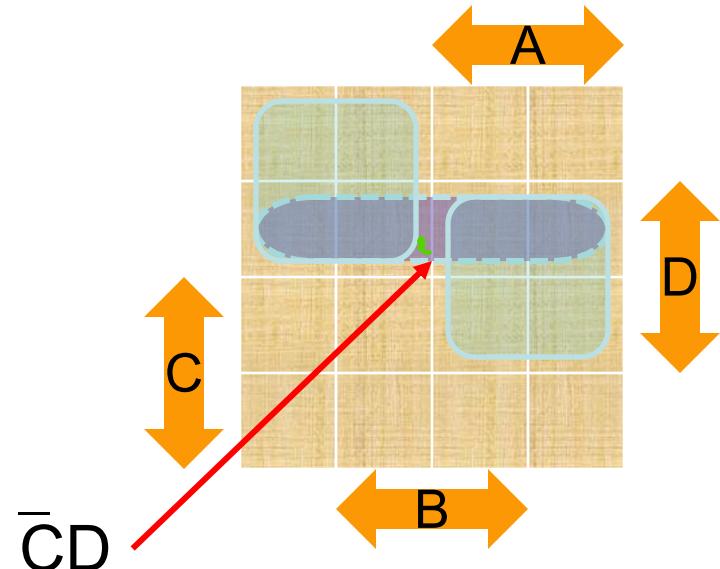
# Example (Cont.)

- PI SOP:  $F = AD + \bar{A} \bar{C} + \bar{C}D$
- Is  $\bar{C}D$  an EPI?

$$\begin{aligned} F - \{\bar{C}D\} &= AD + \bar{A} \bar{C} \\ &= AD + \bar{A} \bar{C} + \bar{C}D \quad (\text{Consensus theorem}) \end{aligned}$$

Hence  $\bar{C}D$  is not an EPI  
(it is an RPI) ✓

Minimum SOP:  
 $F = AD + \bar{A} \bar{C}$



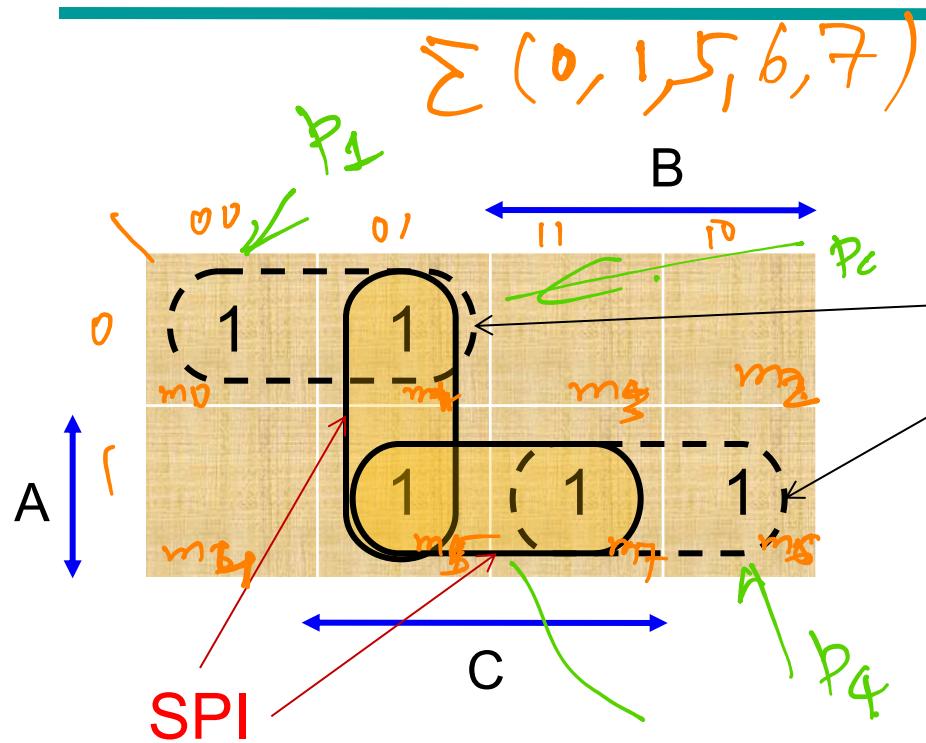
# Finding MSOP

---

1. Start with minterm or cube SOP representation of Boolean function.
2. Find all prime implicants (PI). EPI
3. Include all EPI's in MSOP.]
4. Find the set of uncovered minterms, {UC}.
5. MSOP is minimum if {UC} is empty. *DONE.*
6. For a minterm in {UC}, include the largest PI from remaining PI's (non-EPI's) in MSOP.]
7. Go to step 4.



# Selection of SPI: Patrick's Method



$$f = p_1 + p_2 + p_3 + p_4$$

$$m_0 = p_1 \quad \checkmark$$

$$m_1 = p_1 + p_2 \quad \checkmark$$

$$m_5 = p_2 + p_3 \quad \checkmark$$

$$m_6 = p_3 + p_4 \quad \checkmark$$

$$m_7 = p_4 \quad \text{either}$$

$$p_2 = 1$$

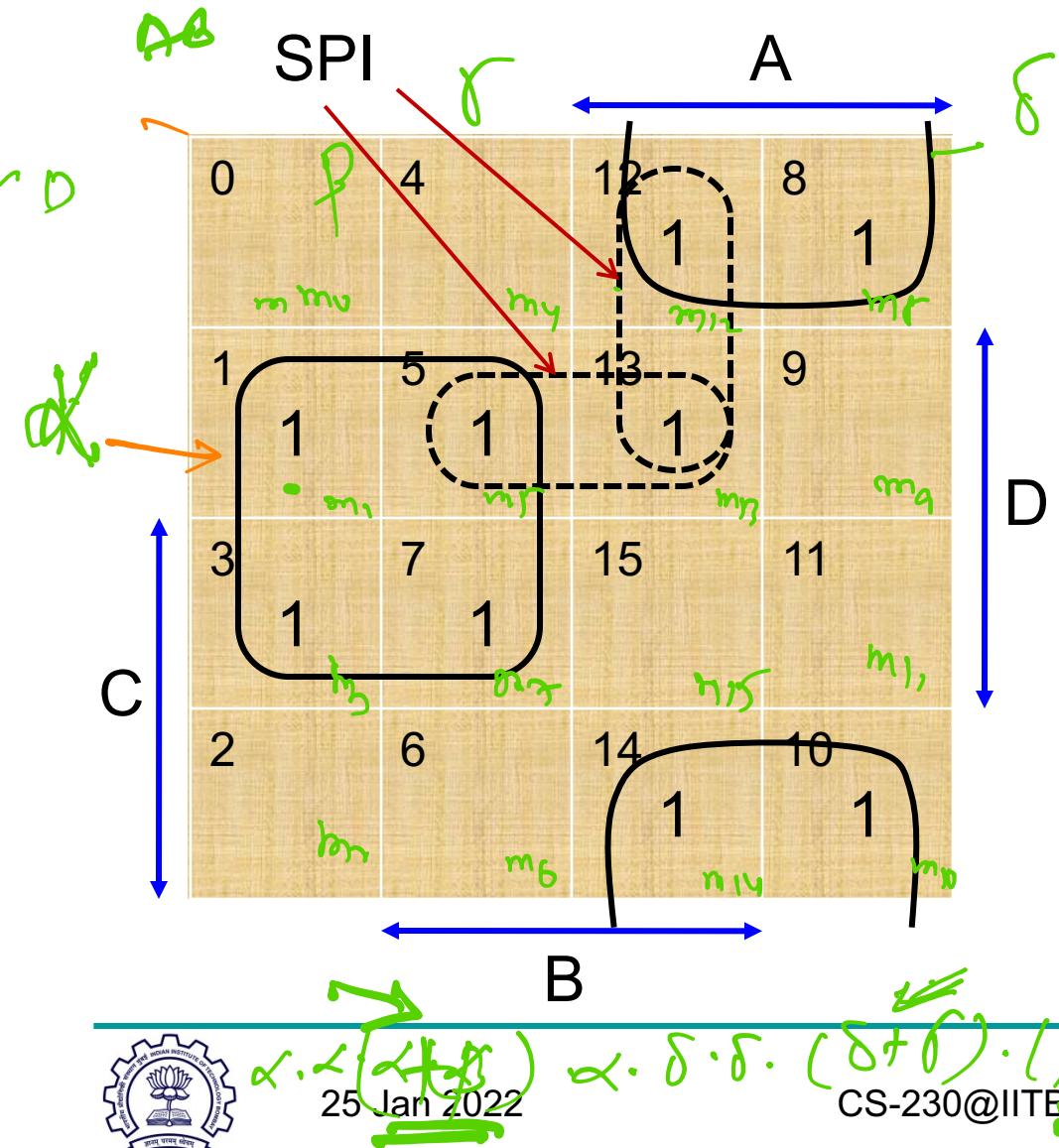
$$p_3 = 1$$

$$p_1 \cdot (p_1 + p_2) \cdot (p_1 + p_3) \cdot (p_1 + p_4) \cdot p_4 = 1$$

$$p_1 = 1 \quad p_4 = 1$$

$$\boxed{p_2 + p_3 = 1}$$

# Example: $F = \sum m(1, 3, 5, 7, 8, 10, 12, 13, 14)$



MSOP:

$$F = \overline{A}D + A\overline{D} + AB\overline{C}$$

$$m_1 = \alpha$$

$$m_3 = \alpha$$

$$m_5 = \alpha + \beta$$

$$m_7 = \alpha$$

$$m_{10} = \delta$$

$$m_{12} = \beta + \gamma$$

$$m_{13} = \beta + \gamma$$

$$m_{14} = \delta$$

$$\beta + \gamma = 1$$

$$\alpha = 1$$

$$\delta = 1$$

# Thank You



# Logic Optimization

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 10 (31 January 2022)

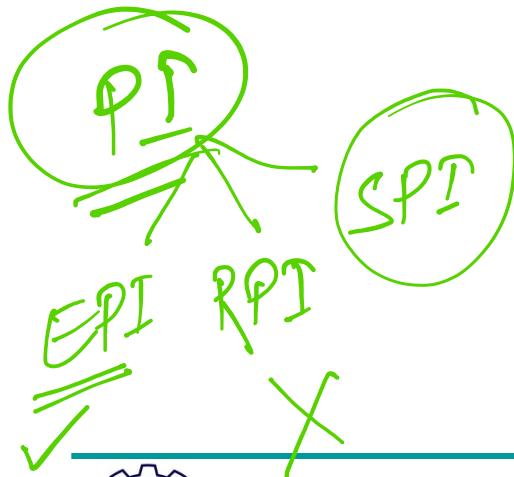
**CADSL**

## Logic & Minimization

Objective → [ Cost ✓  
Delay ]

1, 2, 3, 4. Variable .

		cd	00	01	11	10
		00	$m_0$	$m_1$	$m_3$	$m_2$
a, b		01	$m_4$	$m_5$	$m_7$	$m_6$
		11	$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
		10	$m_8$	$m_9$	$m_{11}$	$m_{10}$



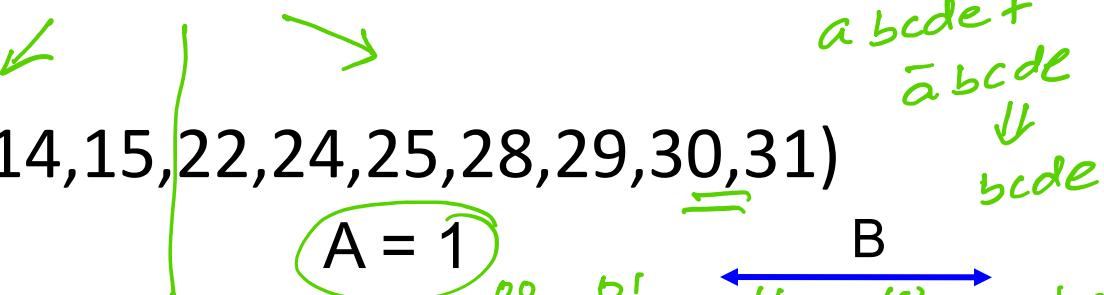
# Five-Variable Function

- $F(A, B, C, D, E)$

$$= \sum m(0, 1, 4, 5, 6, 13, 14, 15, 22, 24, 25, 28, 29, 30, 31)$$

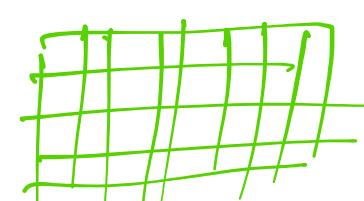
$A = 0$

Karnaugh map for  $A = 0$  (row 0). The columns are labeled B (0, 1, 2, 3) and the rows are labeled D (0, 1, 2, 3). The minterms are marked as 1 in cells (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3).



$\bar{a}bce$

E



Karnaugh map for  $A = 1$  (row 1). The columns are labeled B (0, 1, 2, 3) and the rows are labeled D (0, 1, 2, 3). The minterms are marked as 1 in cells (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3).

E

$a b c d e + \bar{a} b c d e$   
 $\downarrow$   
 $b c d e$

$$F = \bar{A} \bar{B} \bar{D} + A B \bar{D} + B C E + C D \bar{E}$$

$\underline{abce}$

$\cdot \underline{bce}$

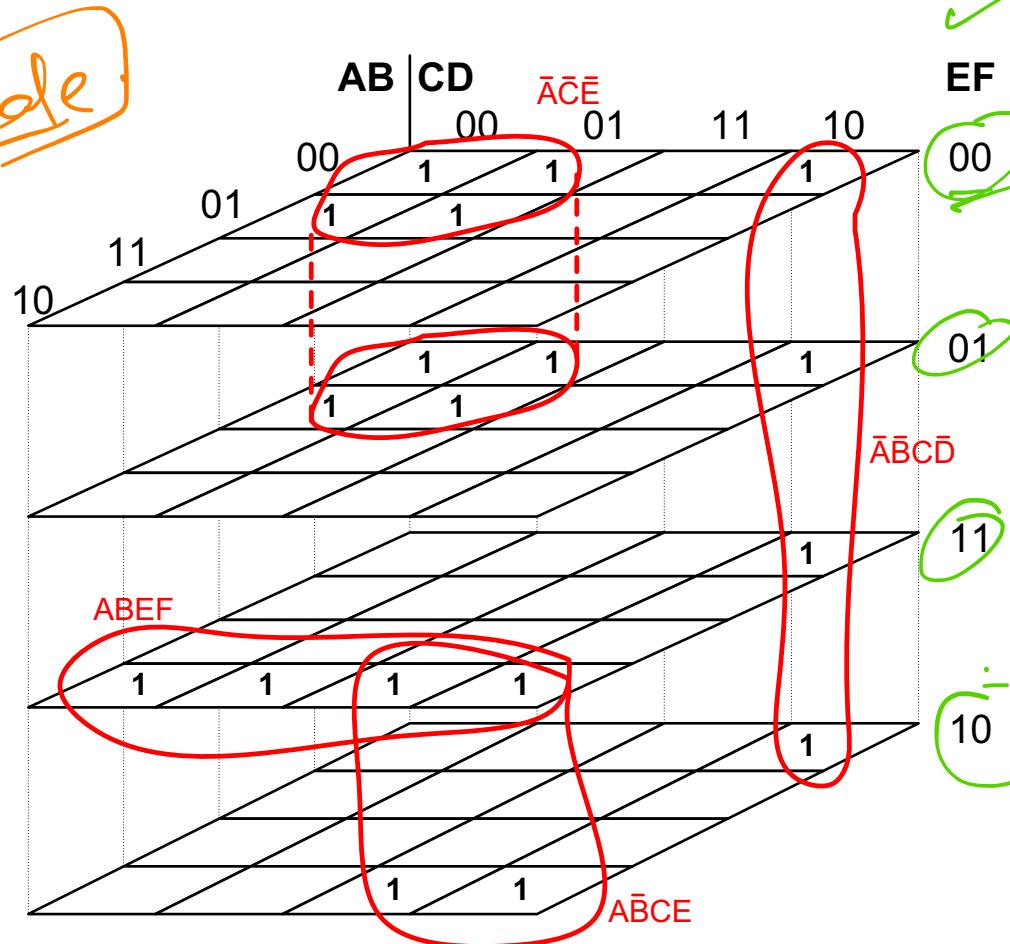


# Why Not More Than Five Variables?

- Too hard to visualize in 3+ dimensions
- Systematic approach:  
“Tabular Methods”
  - Used in computer programs
- Why K-Maps at all?
  - Faster for quick optimizations
  - Understand Boolean logic and HW design
  - Design with simplification in mind

$f(e,f,a,s,c,d)$

Six-Variable K-Map



64

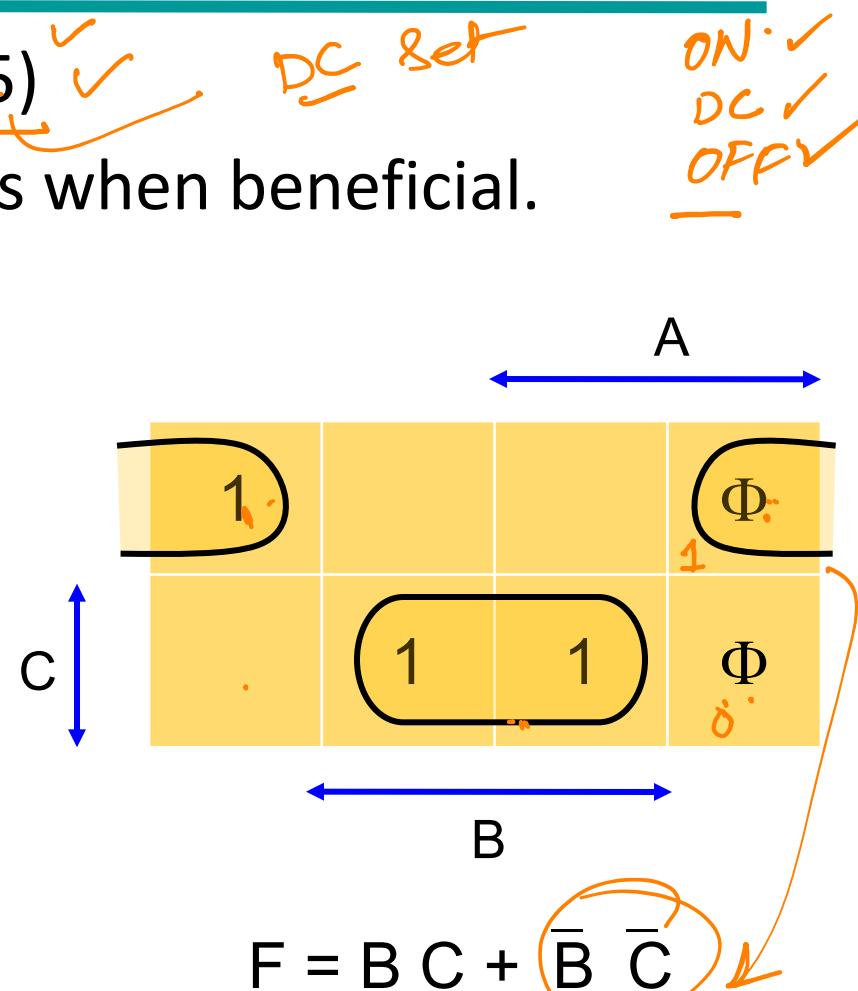
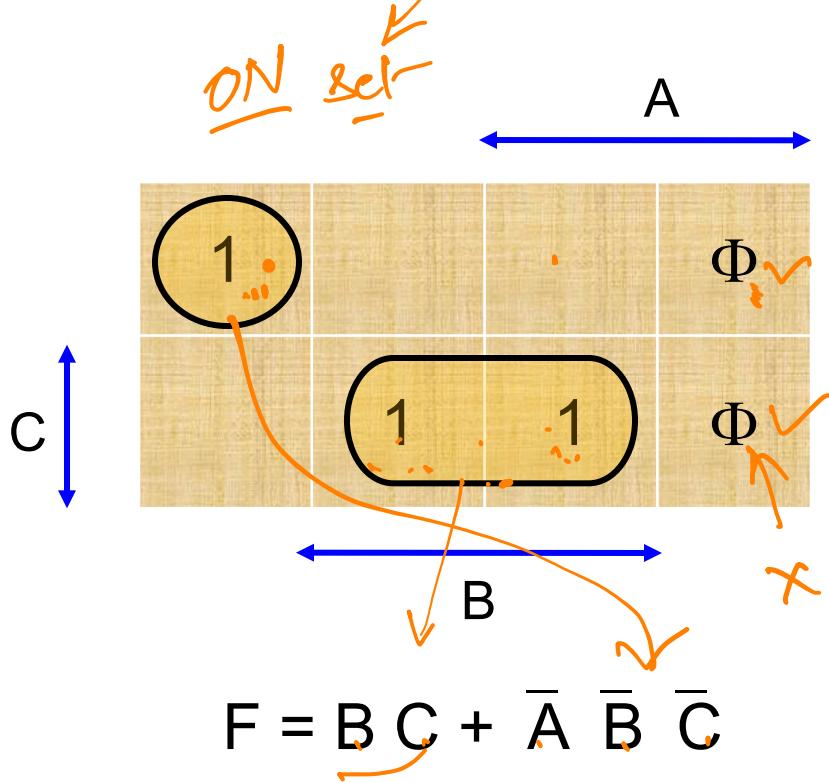
16, 16, 16, 16



CADSL

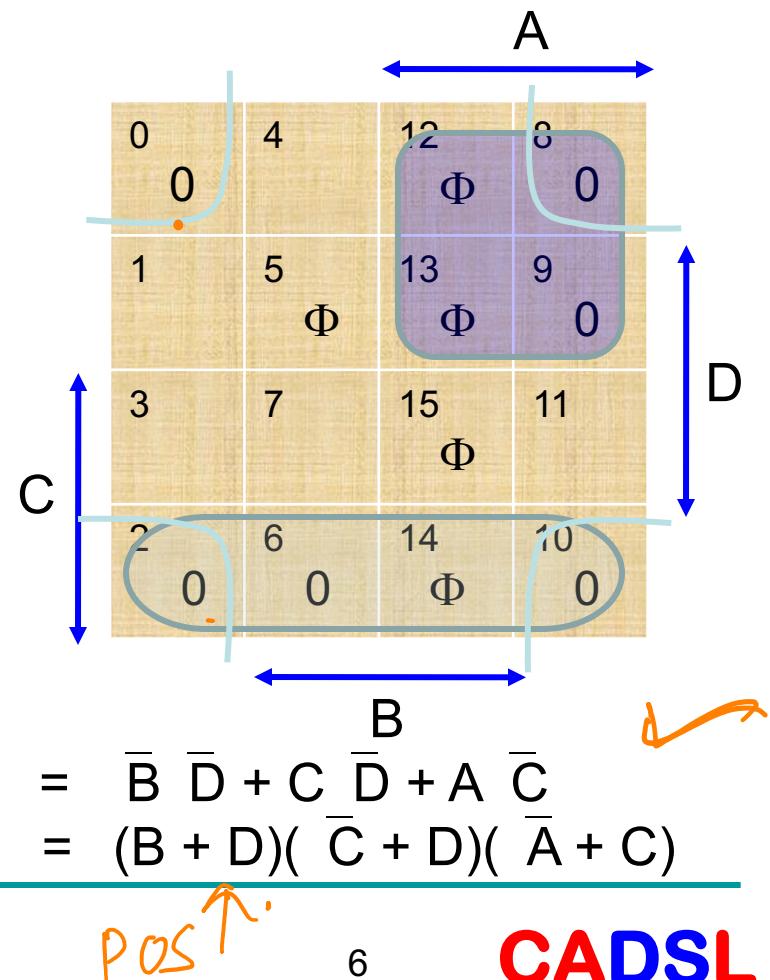
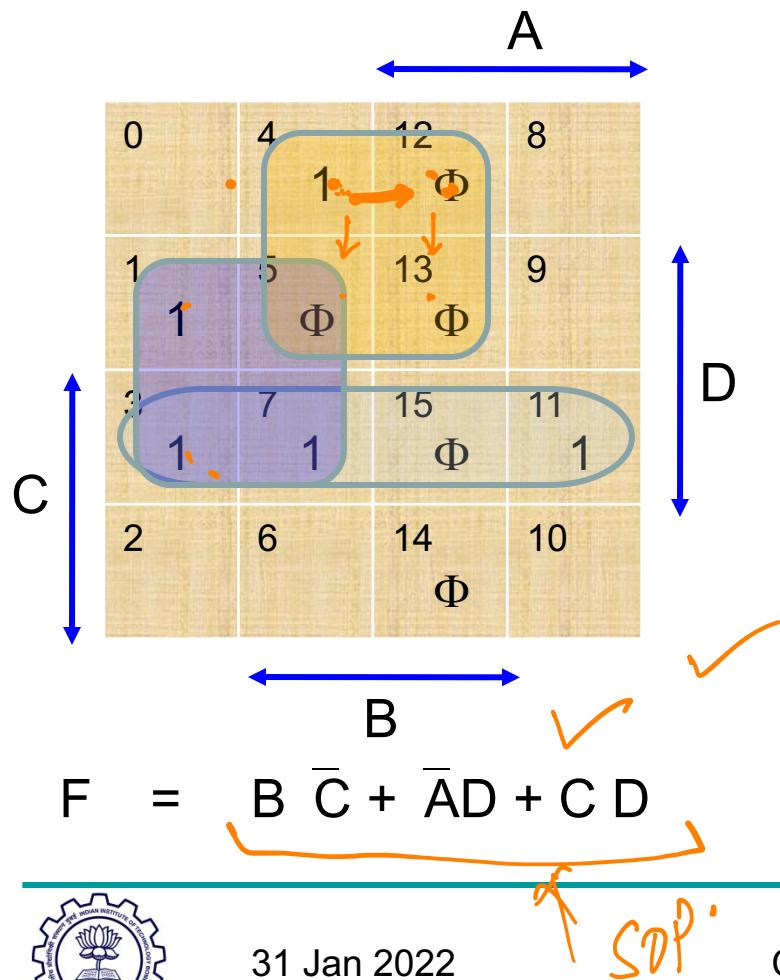
# Functions with Don't Care Minterms

- $F(A,B,C) = \sum m(0,3,7) + d(4,5)$  ✓ DC set
- Include don't care minterms when beneficial.



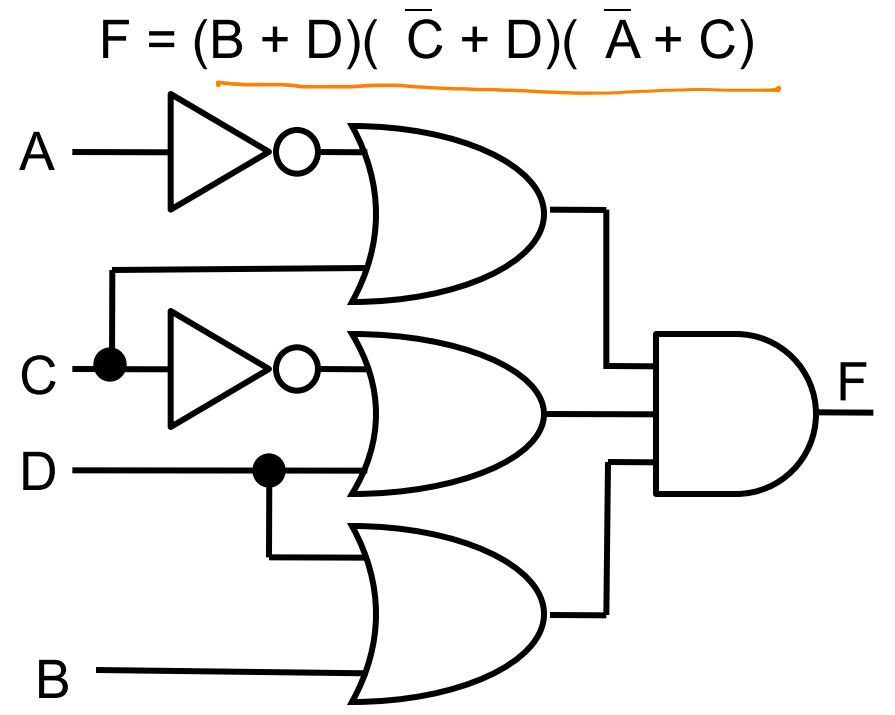
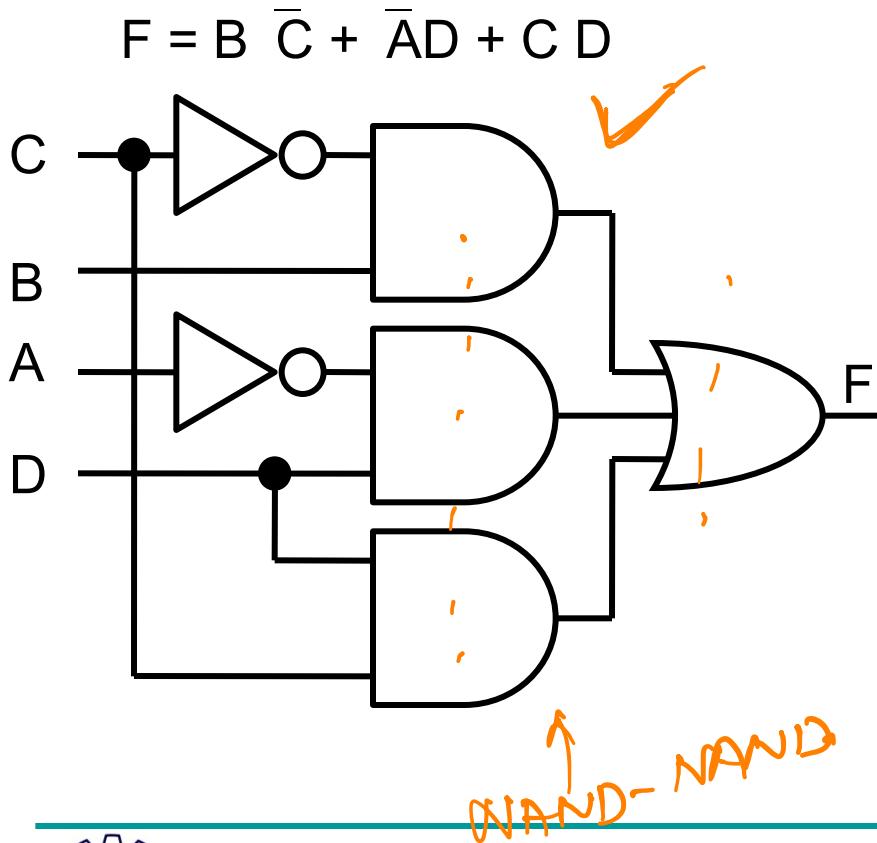
# Minimized SOP and POS

- $$\begin{aligned} F(A,B,C,D) &= \sum m(1,3,4,7,11) + d(5,12,13,14,15) \\ &= \prod M(0,2,6,8,9,10) \ D(5,12,13,14,15) \end{aligned}$$



# SOP and POS Circuits

- $F(A,B,C,D) = \sum m(1,3,4,7,11) + d(5,12,13,14,15)$   
 $= \prod M(0,2,6,8,9,10) D(5,12,13,14,15)$



# Multiple- Output Minimization



# Multiple-Output Minimization

Inputs				Outputs	
A	B	C	D	F1	F2
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	1

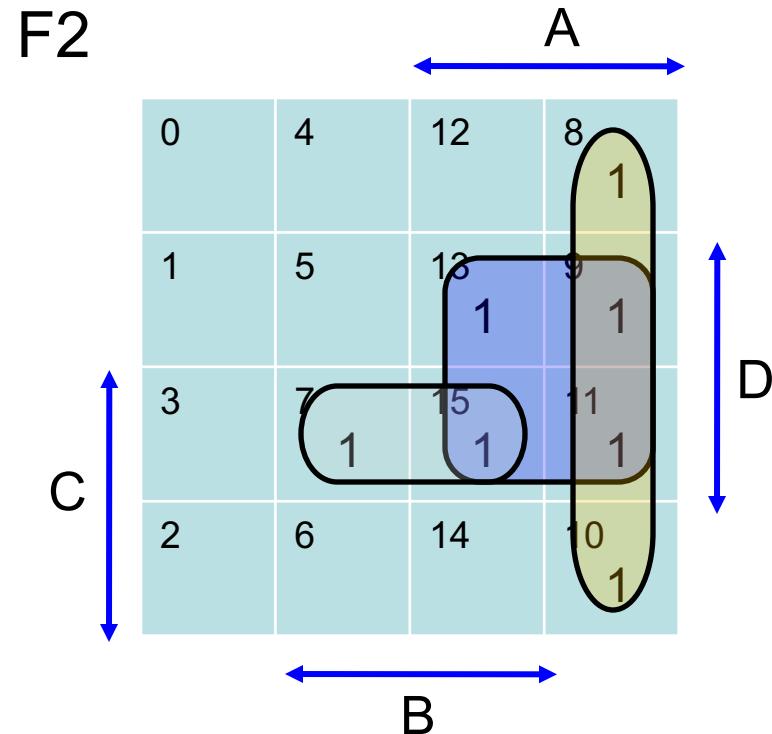
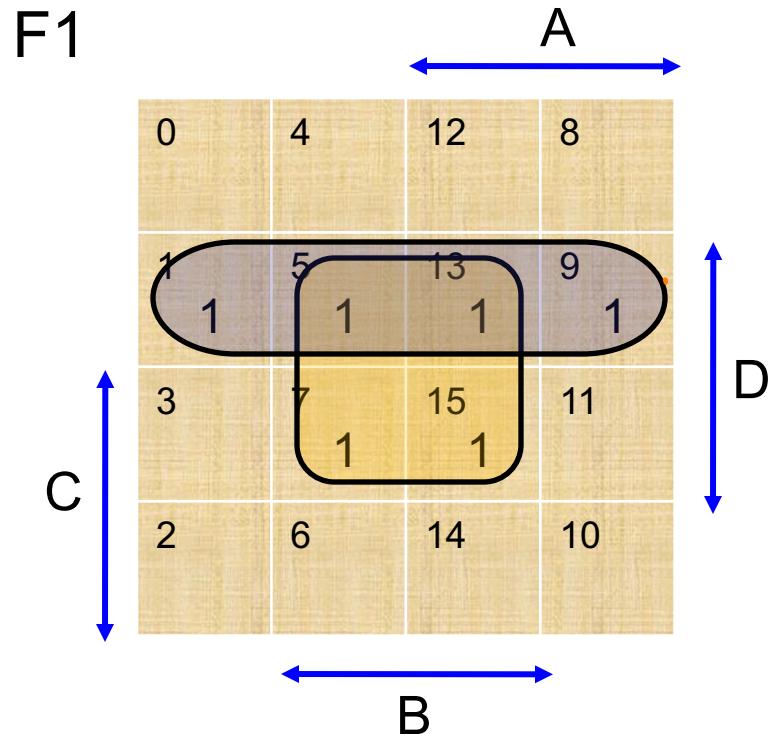
$f_1(A, B, C, D)$

$f_2(A, B, C, D)$



# Individual Output Minimization

Need five products.



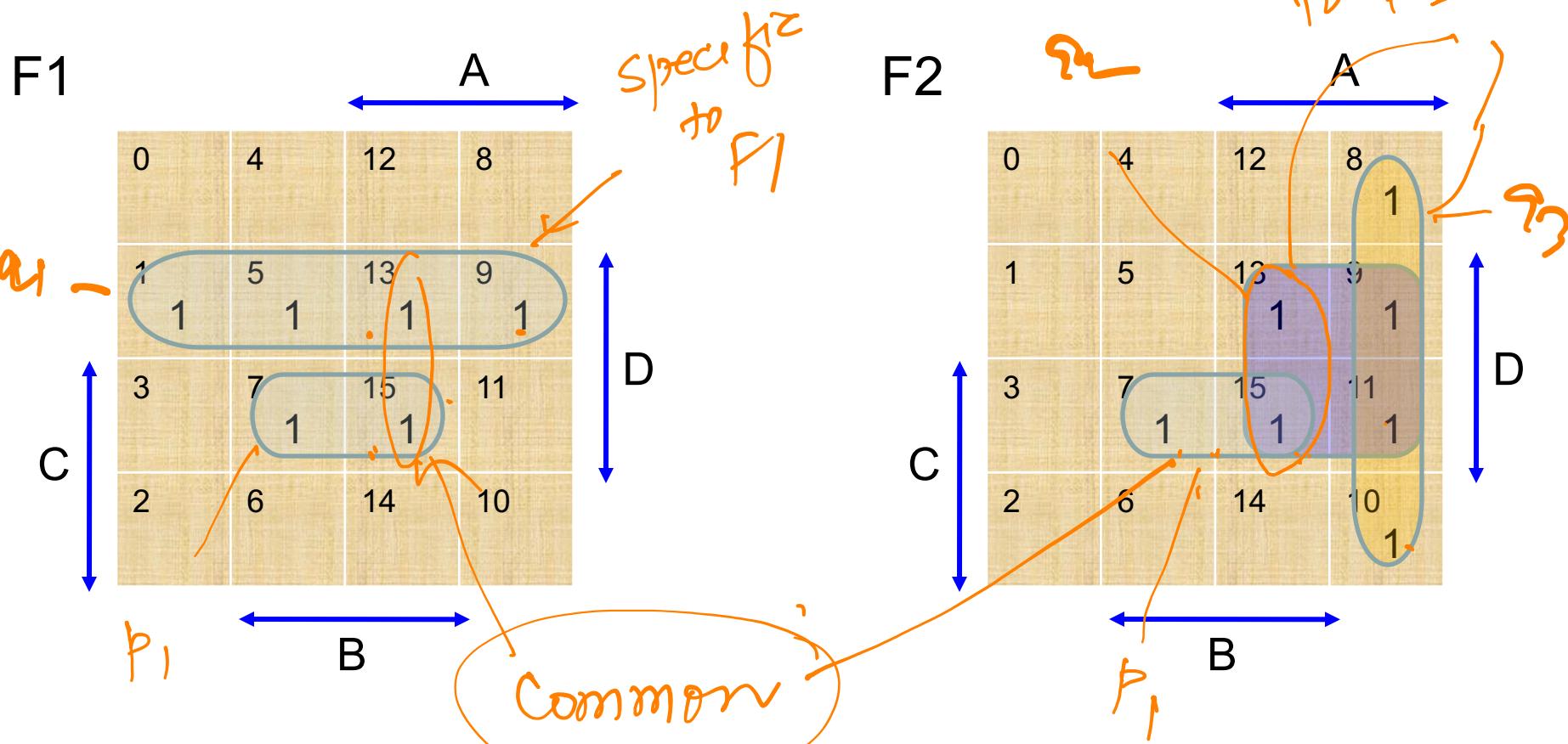
# Global Minimization



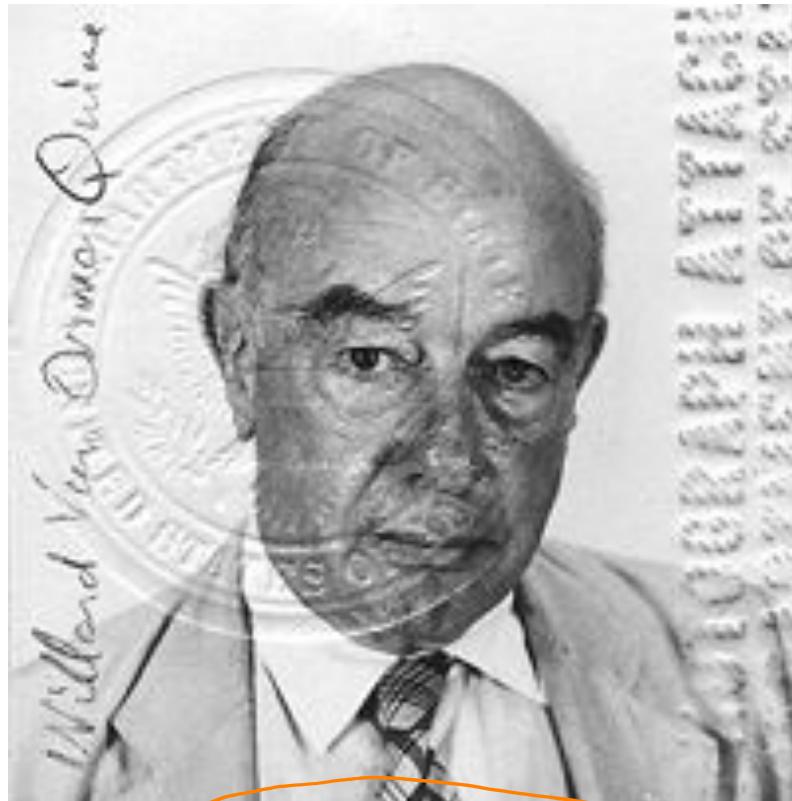
Need four products.



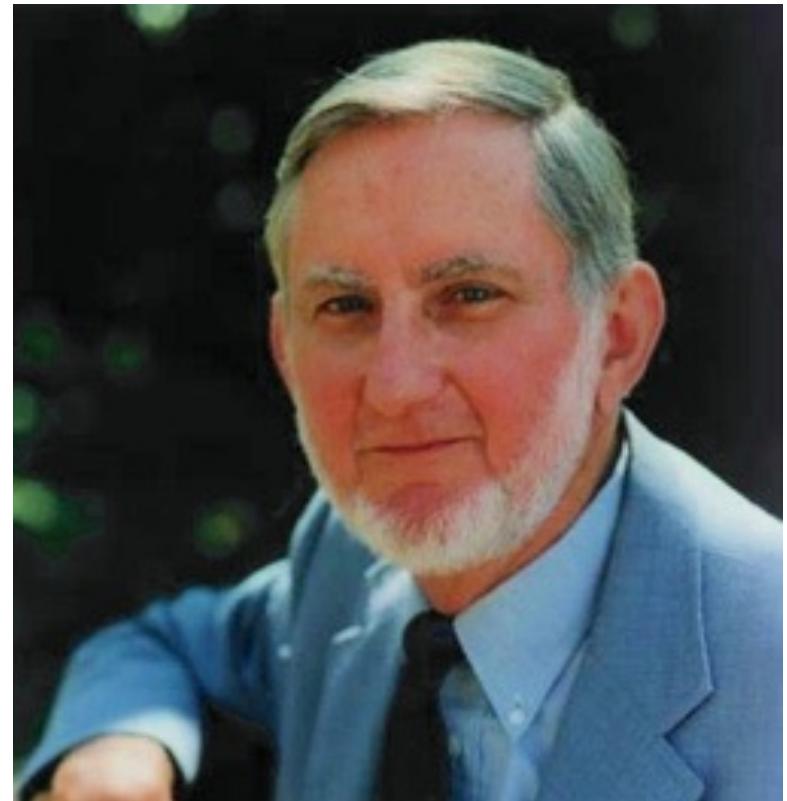
5 variables  
6 variables



# Quine-McCluskey



Willard V. O. Quine  
1908 – 2000



Edward J. McCluskey  
1929 -- 2016

# Quine-McCluskey Tabular Minimization Method

---

- W. V. Quine, “**The Problem of Simplifying Truth Functions**,” *American Mathematical Monthly*, vol. 59, no. 10, pp. 521-531, October 1952. ✓
- E. J. McCluskey, “**Minimization of Boolean Functions**,” *Bell System Technical Journal*, vol. 35, no. 11, pp. 1417-1444, November 1956.



# Q-M Tabular Minimization

- Minimizes functions with many variables.
- Begin with minterms:
  - Step 1: **Tabulate minterms** in groups of increasing number of true variables.
  - Step 2: Conduct **linear searches** to identify all prime implicants (PI).
  - Step 3: Tabulate PI's vs. minterms to identify EPI's.
  - Step 4: Tabulate non-essential PI's vs. minterms not covered by EPI's. *Select minimum* number of PI's to cover all minterms.
- MSOP contains all **EPI's** and *selected* non-EPI's.

$$\begin{matrix} abc \\ ab\bar{c} \end{matrix} \} \Rightarrow \underline{\underline{ab}}$$

$$\begin{matrix} abc \\ a\bar{b}c \end{matrix} \Rightarrow \underline{ac}$$
$$\begin{matrix} abc \\ \bar{a}bc \end{matrix} \} \underline{bc}$$

EPT



$$F(A,B,C,D) = \sum m(2,4,6,8,9,10,12,13,15)$$

- Q-M Step 1: Group minterms with 1 true variable, 2 true variables, etc.

0010  
 0010 -2  
 0100 -4  
 0000 -8  
 0110

Minterm	ABCD	Groups
2 ✓	0010	1: single 1
4	0100	
8	1000	2: two 1's
6 ✓	0110	
9	1001	3: three 1's
10	1010	
12	1100	4: four 1's
13	1101	
15	1111	

G1  
 G2  
 ✓

]  
 (2, 6)  
 (2, 10)



# Q-M Step 2

---

- Find all implicants by combining minterms, and then combining products that differ in a single variable: For example,
  - 2 and 6, or  $\bar{A} \bar{B} C \bar{D}$  and  $\bar{A} B C \bar{D} \rightarrow \bar{A} C \bar{D}$ , written as 0 – 1 0.
- Try combining a minterm (or product) with all minterms (or products) listed below in the table.
- Include resulting products in the next list.
- If minterm (or product) does not combine with any other, mark it as PI. ✓
- Check the minterm (or product) and repeat for all other minterms (or products).



# Step 2 Executed on Example

List 1			List 2			List 3		
Minterm	ABCD	PI?	Minterms	ABCD	PI?	Minterms	ABCD	PI?
2	0010	X	2, 6	0-10	PI_2	8,9,12,13	1-0-	PI_1
4	0100	X	2,10	-010	PI_3			
8	1000	X	4,6	01-0	PI_4			
6	0110	X	4,12	-100	PI_5			
9	1001	X	8,9	100-	X			
10	1010	X	8,10	10-0	PI_6			
12	1100	X	8,12	1-00	X			
13	1101	X	9,13	1-01	X			
15	1111	X	12,13	110-	X			
			13,15	11-1	PI_7			



EPL

SPL



# Step 3: Identify EPI's

Covered by EPI →				x	x			x	x	x
Minterms →	2	4	6	8	9	10	12	13	15	
PI_1 is EPI				x	x		x	x		
PI_2	x		x	.						
PI_3	x					x				
PI_4		x	x							
PI_5		x						x		
PI_6				x		x				
PI_7 is EPI								x	x	

PI\_1 → PI\_7

EPI



# Step 4: Cover Remaining Minterms

Remaining minterms →	2	4	6	10
$x_2$	PI_2	x	x	
$x_3$	PI_3	x		x
$x_4$	PI_4		x ✓	x ✓
$x_5$	PI_5		x ✓	
$x_6$	PI_6			x

Integer linear program (ILP), available from MATLAB and other sources: Define integer {0,1} variables,  $x_k = 1$ , select PI\_k;  $x_k = 0$ , do not select PI\_k.

Minimize  $\sum_k x_k$ , subject to constraints:

$$\begin{aligned}x_2 + x_3 &\geq 1 \\x_4 + x_5 &\geq 1 \\x_2 + x_4 &\geq 1 \\x_3 + x_6 &\geq 1\end{aligned}$$

$$\min(x_2 + x_3 + x_4 + x_5 + x_6)$$

A solution is  $x_3 = x_4 = 1$ ,  $x_2 = x_5 = x_6 = 0$ , or select PI\_3, PI\_4



# Linear Programming (LP)

---

- A mathematical optimization method for problems where some “cost” depends on a large number of variables.
- An easy to understand introduction is:
  - S. I. Gass, *An Illustrated Guide to Linear Programming*, New York: Dover Publications, 1970.
- Very useful tool for a variety of engineering design problems.
- Available in software packages like MATLAB.



# Step 4: Cover Remaining Minterms

Remaining minterms →	2	4	6	10
$x_2$	PI_2	x		x
$\bar{x}_3$	PI_3	x		x
$x_4$	PI_4		x	x
$\bar{x}_5$	PI_5		x	
$x_6$	PI_6			x

Patrick's Method

Noticed.

$$(x_2 + \bar{x}_3) \cdot (\bar{x}_4 + x_5) \cdot (\underline{x_2} + \underline{x_4}) \cdot (\underline{x_3} + \underline{x_6}) = 1$$

SOP expression

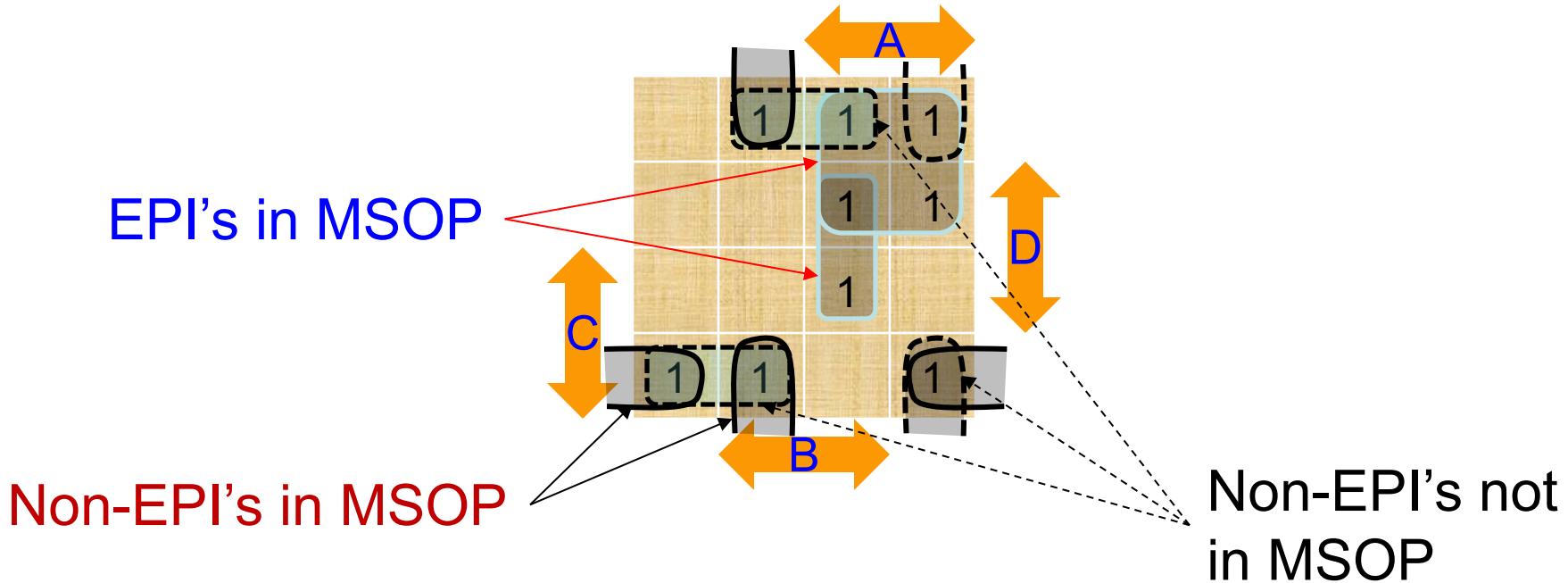
POS  
 $x_2 = x_3 = x_4 = x_5 = x_6 = 1$

$$(x_2 x_4 + x_2 \bar{x}_5 + \bar{x}_3 x_4 + x_3 \bar{x}_5)$$

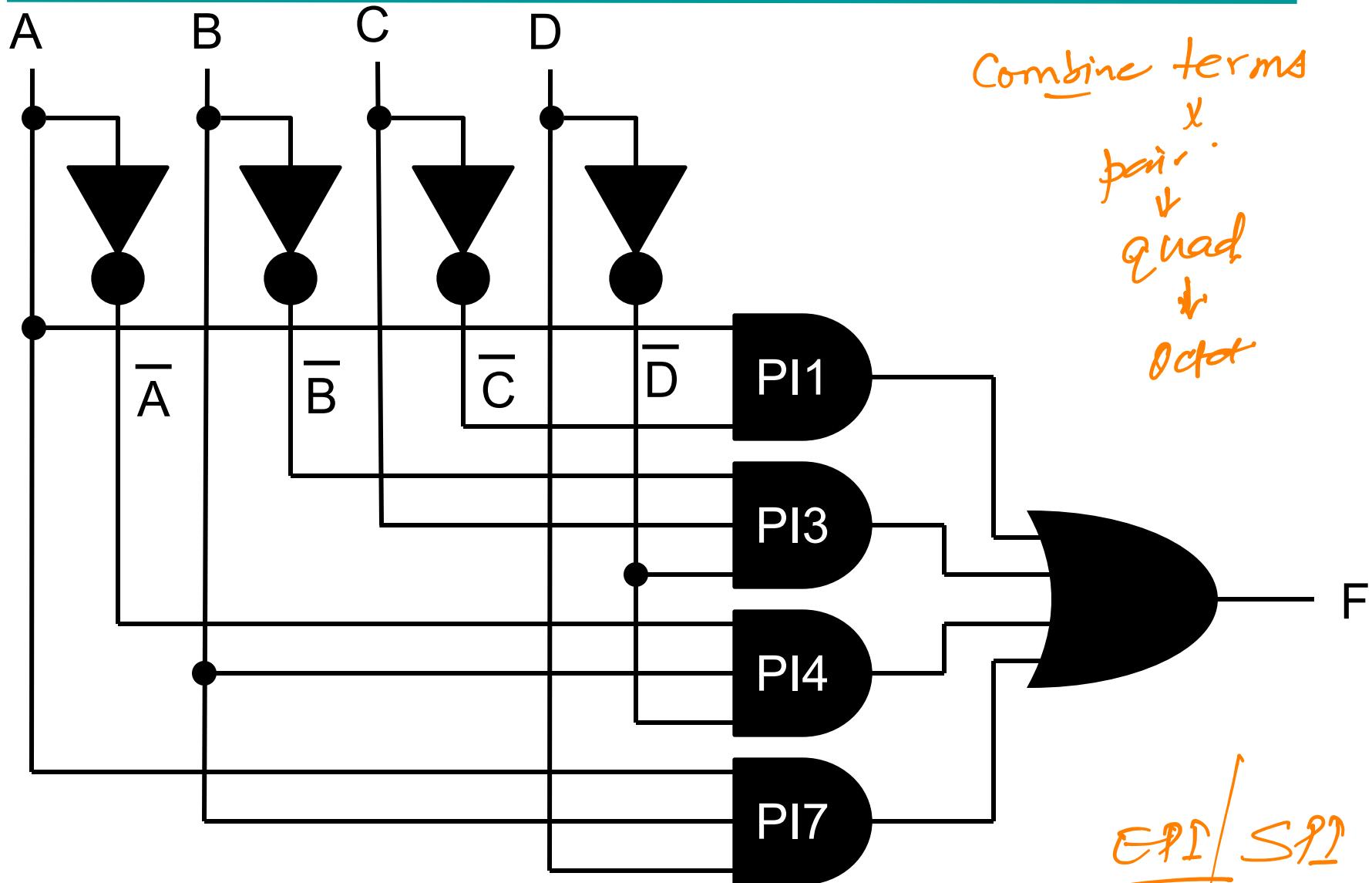


# Q-M MSOP Solution and Verification

- $F(A,B,C,D) = PI_1 + PI_3 + PI_4 + PI_7$   
=  $1\bar{0}\bar{1} + \bar{0}10 + 01\bar{0} + 11\bar{1}$   
=  $A\bar{C} + \bar{B}C\bar{D} + \bar{A}B\bar{D} + ABD$
- See Karnaugh map.



# Minimized Circuit



# Function with Don't Cares

$$F(A,B,C,D) = \sum m(4,6,8,9,10,12,13) + \sum d(2, 15)$$

- Q-M Step 1: Group “all” minterms with 1 true variable, 2 true variables, etc.

Minterm	ABCD	Groups
2	0010	1: single 1
4	0100	
8	1000	
6	0110	2: two 1's
9	1001	
10	1010	
12	1100	
13	1101	3: three 1's
15	1111	4: four 1's



# Step 2: Same As Before on “All” Minterms

---

List 1			List 2			List 3		
Minterm	ABCD	PI?	Minterms	ABCD	PI?	Minterms	ABCD	PI?
2	0010	X	2, 6	0-10	PI2	8,9,12,13	1-0-	PI1
4	0100	X	2,10	-010	PI3			
8	1000	X	4,6	01-0	PI4			
6	0110	X	4,12	-100	PI5			
9	1001	X	8,9	100-	X			
10	1010	X	8,10	10-0	PI6			
12	1100	X	8,12	1-00	X			
13	1101	X	9,13	1-01	X			
15	1111	X	12,13	110-	X			
			13,15	11-1	PI7			



# Step 3: Identify EPI's Ignoring Don't Cares

Covered by EPI →			x	x		x	x
Minterms →	4	6	8	9	10	12	13
PI1 is EPI			x	x		x	x
PI2		x					
PI3					x		
PI4	x	x					
PI5	x					x	
PI6			x		x		
PI7							x



# Step 4: Cover Remaining Minterms

Remaining minterms →	4	6	10
PI_2		x	
PI_3			x
PI_4	x	x	
PI_5	x		
PI_6			x

Integer linear program (ILP), available from Matlab and other sources: Define integer {0,1} variables,  $x_k = 1$ , select PI\_k;  $x_k = 0$ , do not select PI\_k.

Minimize  $\sum_k x_k$ , subject to constraints:

$$x_4 + x_5 \geq 1$$

$$x_2 + x_4 \geq 1$$

$$x_3 + x_6 \geq 1$$

A solution is  $x_3 = x_4 = 1$ ,  $x_2 = x_5 = x_6 = 0$ , or select PI\_3, PI\_4



# Step 4: Cover Remaining Minterms

---

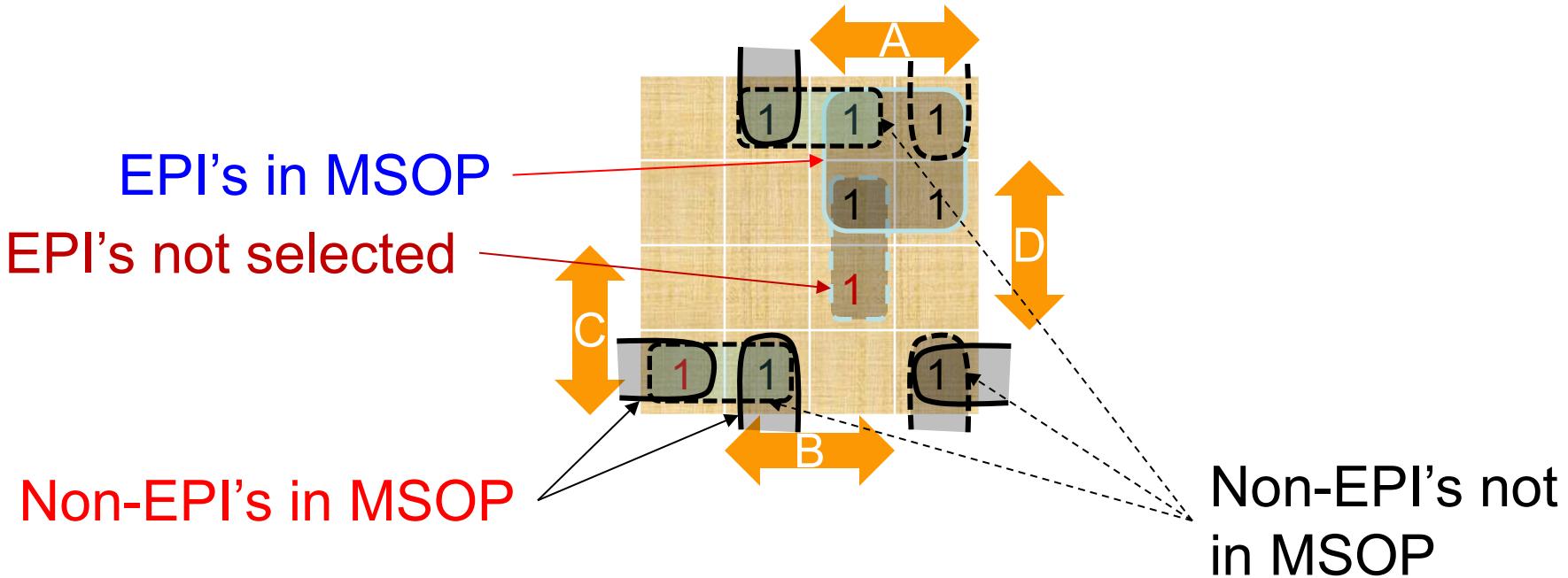
Remaining minterms →	4	6	10
PI_2		x	
PI_3			x
PI_4	x	x	
PI_5	x		
PI_6			x

Patrick's Method

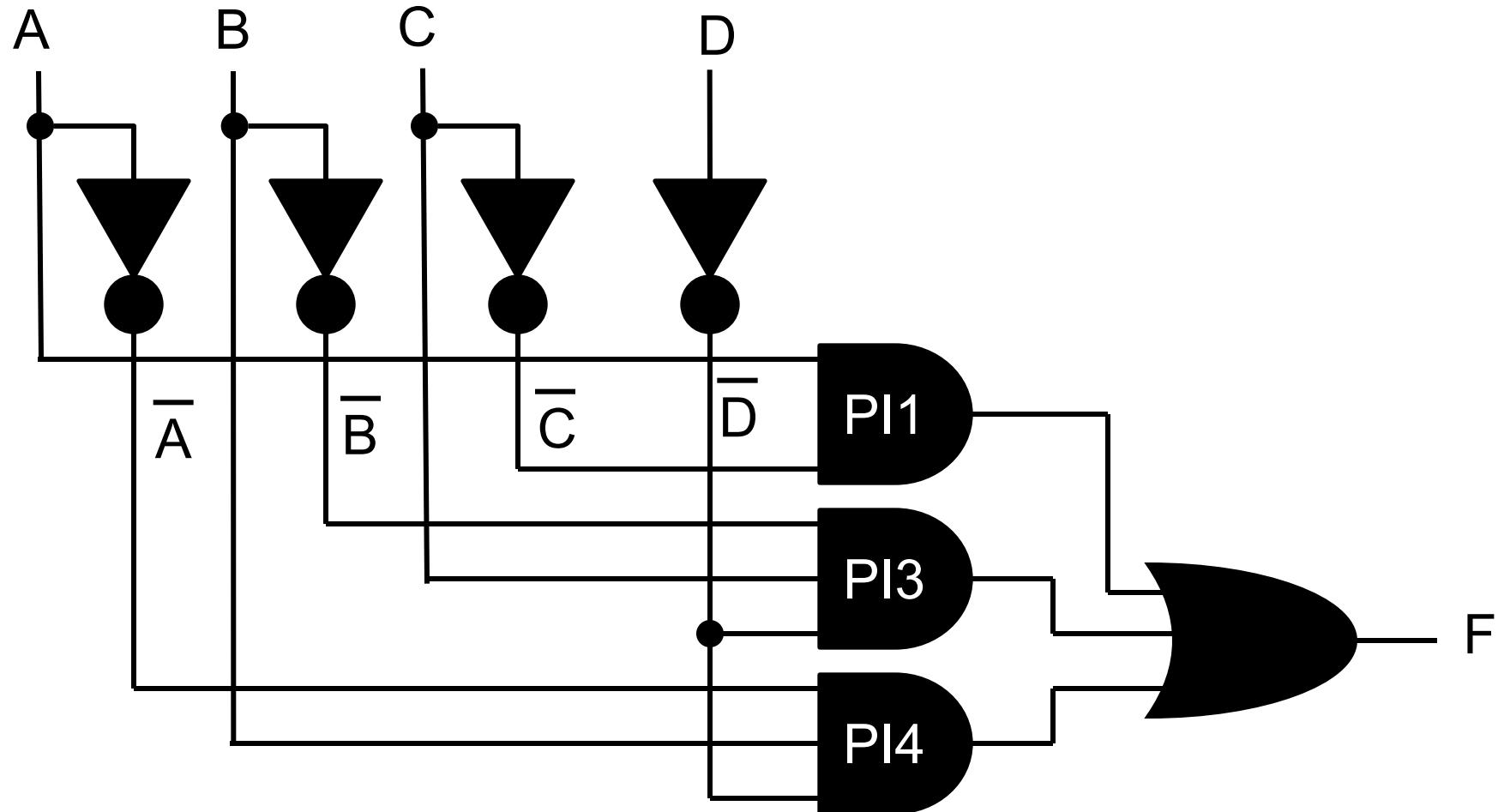


# Q-M MSOP Solution and Verification

- $F(A,B,C,D) = \text{PI\_1} + \text{PI\_3} + \text{PI\_4}$   
=  $1\text{-}0\text{-} + -010 + 01\text{-}0$   
=  $A \bar{C} + \bar{B} C \bar{D} + \bar{A} B \bar{D}$
- See Karnaugh map.



# Minimized Circuit



# QM Minimizer on the Web

---

- <http://quinemccluskey.com/>



# Thank You



# Arithmetic Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee{iitb.ac.in}](mailto:viren@cse, ee{iitb.ac.in})

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 12 (01 February 2022)

**CADSL**

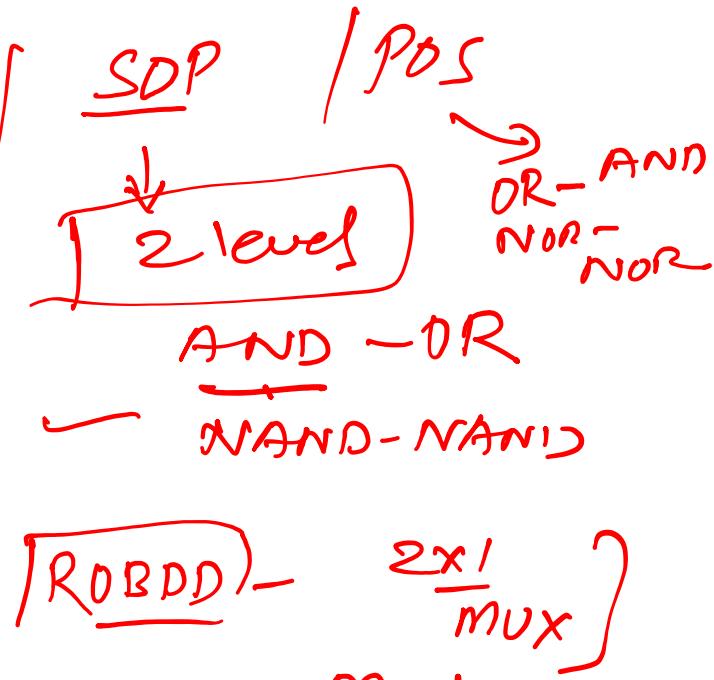
Logical expression



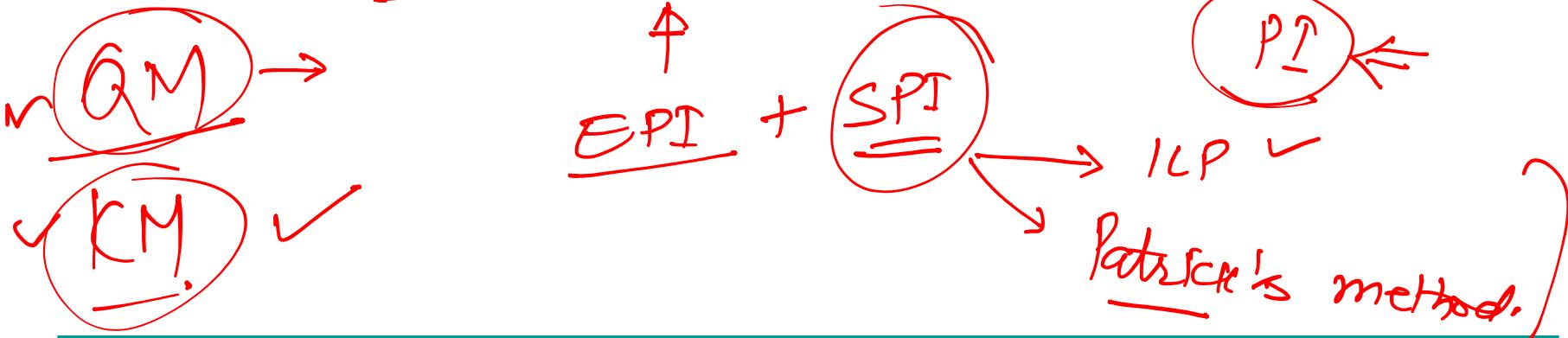
Minimization

COST  
# switches

DELAY



$\min \left( \begin{array}{c} \# \text{ no. of product terms} \\ \hline \end{array} \right)$  Multi-level  
exp.



# Multiple-Level Optimization

---



- Multiple-level circuits are circuits that have more than two level (plus input and/or output inverters)
- For a given function, multiple-level circuits can have reduced gate input cost compared to two-level (SOP and POS) circuits
- Multiple-level optimization is performed by applying transformations to circuits represented by equations while evaluating cost



# Transformations

---

- Factoring - finding a factored form from SOP or POS expression  $a + \overline{a}b = a + b$
- ✓ – Algebraic - No use of axioms specific to Boolean algebra such as complements or idempotence
- ✓ – Boolean - Uses axioms unique to Boolean algebra
- Decomposition - expression of a function as a set of new functions



# Transformations (continued)

---

- Substitution of G into F - expression function F as a function of G and some or all of its original variables
- Extraction - decomposition applied to multiple functions simultaneously



# Transformation Examples

## Algebraic Factoring

$$F = \bar{A} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot C + A \cdot C \cdot \bar{D}$$

– Factoring:

$$F = \bar{A} \cdot (\bar{C} \cdot \bar{D} + B \cdot \bar{C}) + A \cdot (B \cdot C + C \cdot \bar{D})$$

– Factoring again:

$$F = \bar{A} \cdot \bar{C} \cdot (B + \bar{D}) + A \cdot C \cdot (B + \bar{D})$$

– Factoring again:

$$F = (\bar{A} \cdot \bar{C} + A \cdot C) \cdot (B + \bar{D})$$

12+4

G = 16 ✓

G = 16 ✓

G = 12

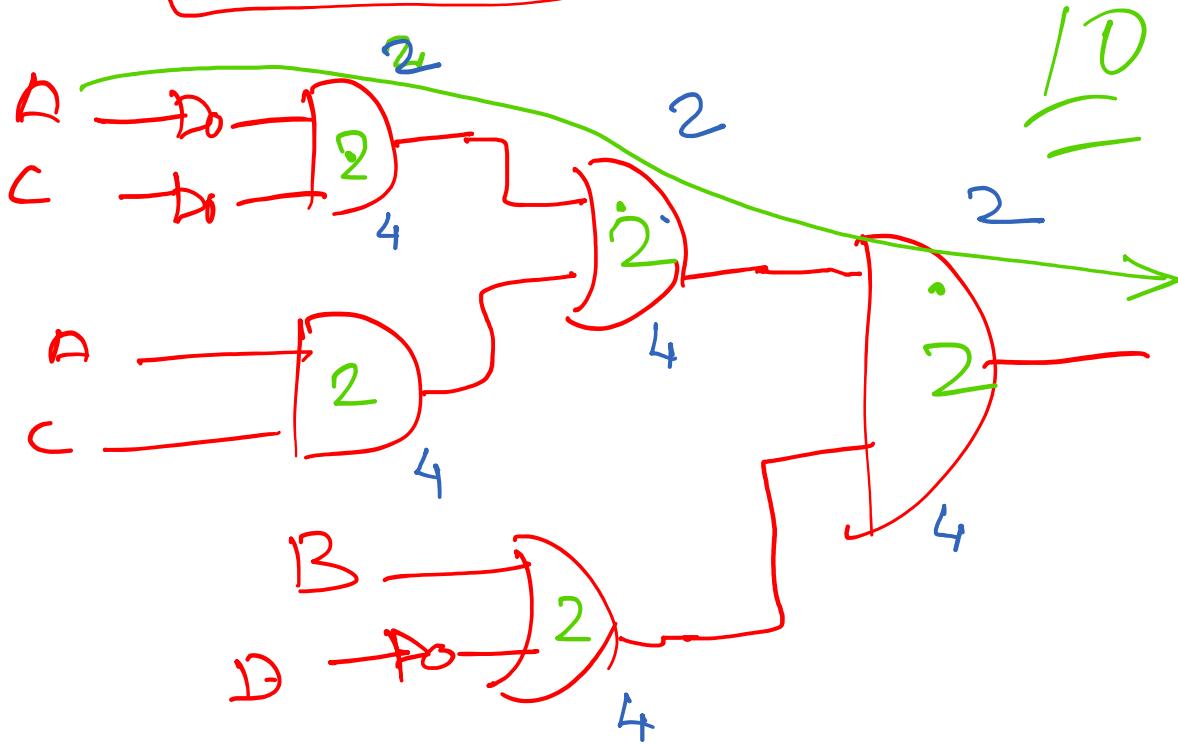
8+2+2

G = 10

6+2+2 ✓



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



162

Cost

.....



# Transformation Examples

- Decomposition

$$(A'C + AC) LB2D)$$

–  $F = A'C'D' + A'BC' + ABC + ACD'$   $G = 16$

– The terms  $A'C' + AC$  and  $B + D'$  can be defined as new functions  $H$  and  $E$  respectively, decomposing

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

$$F = H E, H = A'C' + AC, E = B + D' \quad G = 10$$

- This series of transformations has reduced  $G$  from 16 to 10, a substantial savings.
- The resulting circuit has three levels plus input  $\bar{B}$  inverters.



# Transformation Examples

- Substitution of E into F

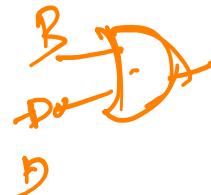
- Returning to F just before the final factoring step:

$$F = \bar{A} \bar{C} (B + \bar{D}) + AC (B + \bar{D}) \quad G = 12$$

- Defining  $E = B + \bar{D}$ , and substituting in F:

$$F = \bar{A} \bar{C} E + AC E \quad G = 10$$

- This substitution has resulted in the same cost as the decomposition



# Transformation Examples



01 Feb 2022

CS-230@IITB

10

**CADSL**

# Summary

---

- Multi-level Optimization ✓

- Transformations

- Factoring - find a factored form from SOP or POS expression
- Decomposition - express a function as a set of new functions
- Substitution - express function F as a function of G and some or all of its original variables



# Arithmetic

## Circuits:

### Adders

TJH KPF

2 input ~  
Adder  
↓  
HA

3 input ~  
Adder  
↓  
FA  
=



# Half-Adder Adds two Bits

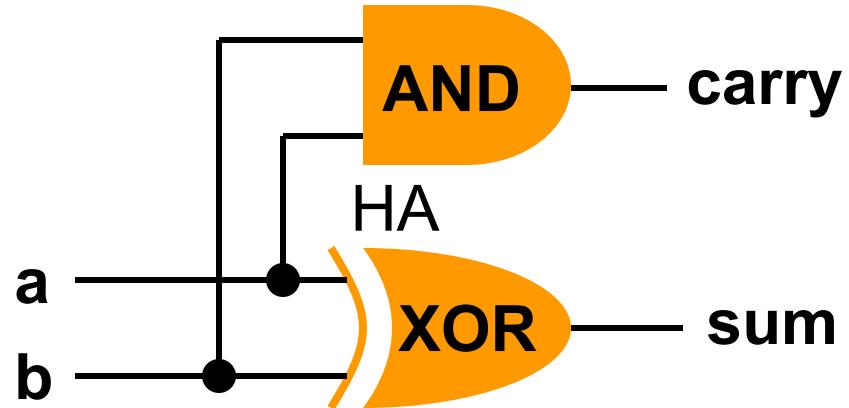
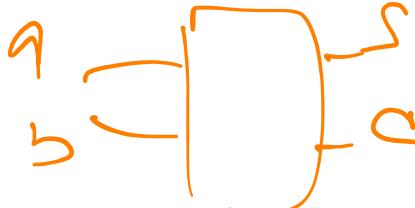
*“half” because it has no carry input*

---

- Adding two bits:

a	b	$a + b$
0	0	00
0	1	01
1	0	01
1	1	10

carry      sum



# Full Adder: Include Carry Input

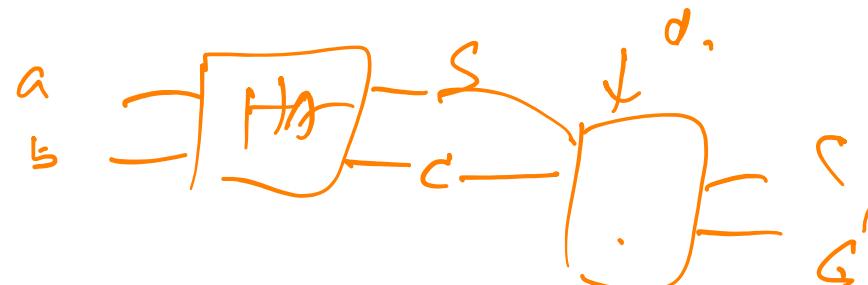
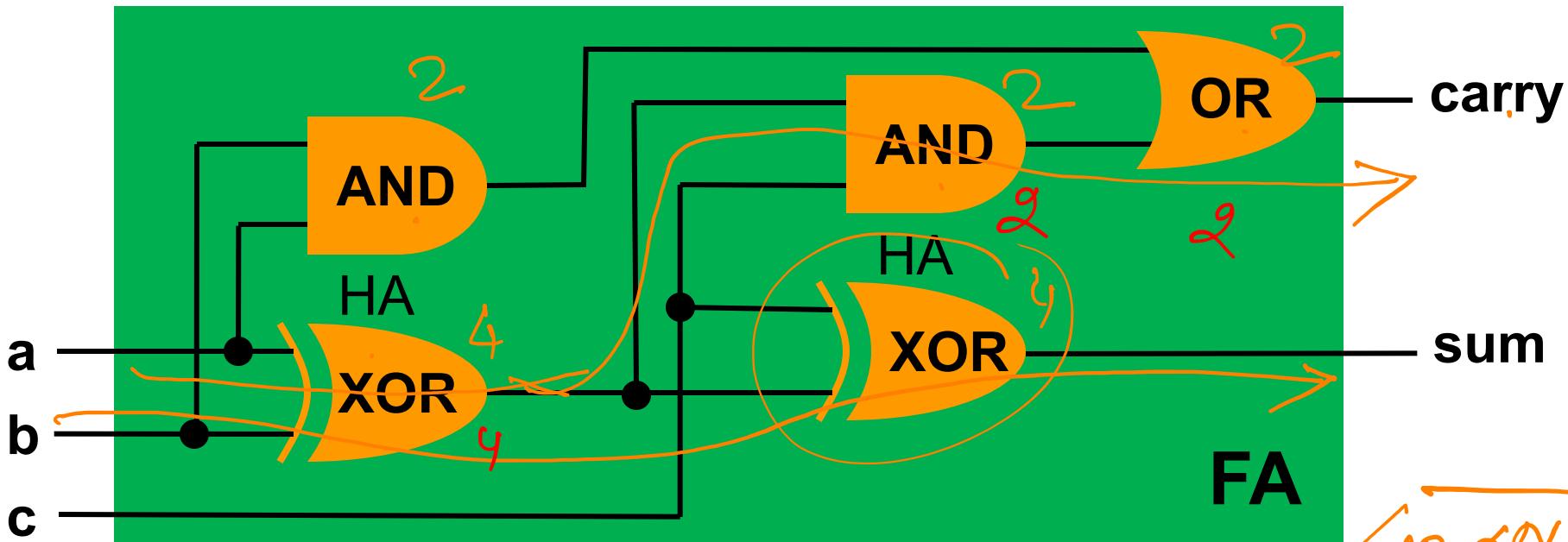
a	b	c	$s = a + b + c$	
			Decimal value	Binary value
0	0	0	0	00
0	0	1	1	01
0	1	0	1	01
0	1	1	2	10
1	0	0	1	01
1	0	1	2	10
1	1	0	2	10
1	1	1	3	11



# Full-Adder Adds Three Bits

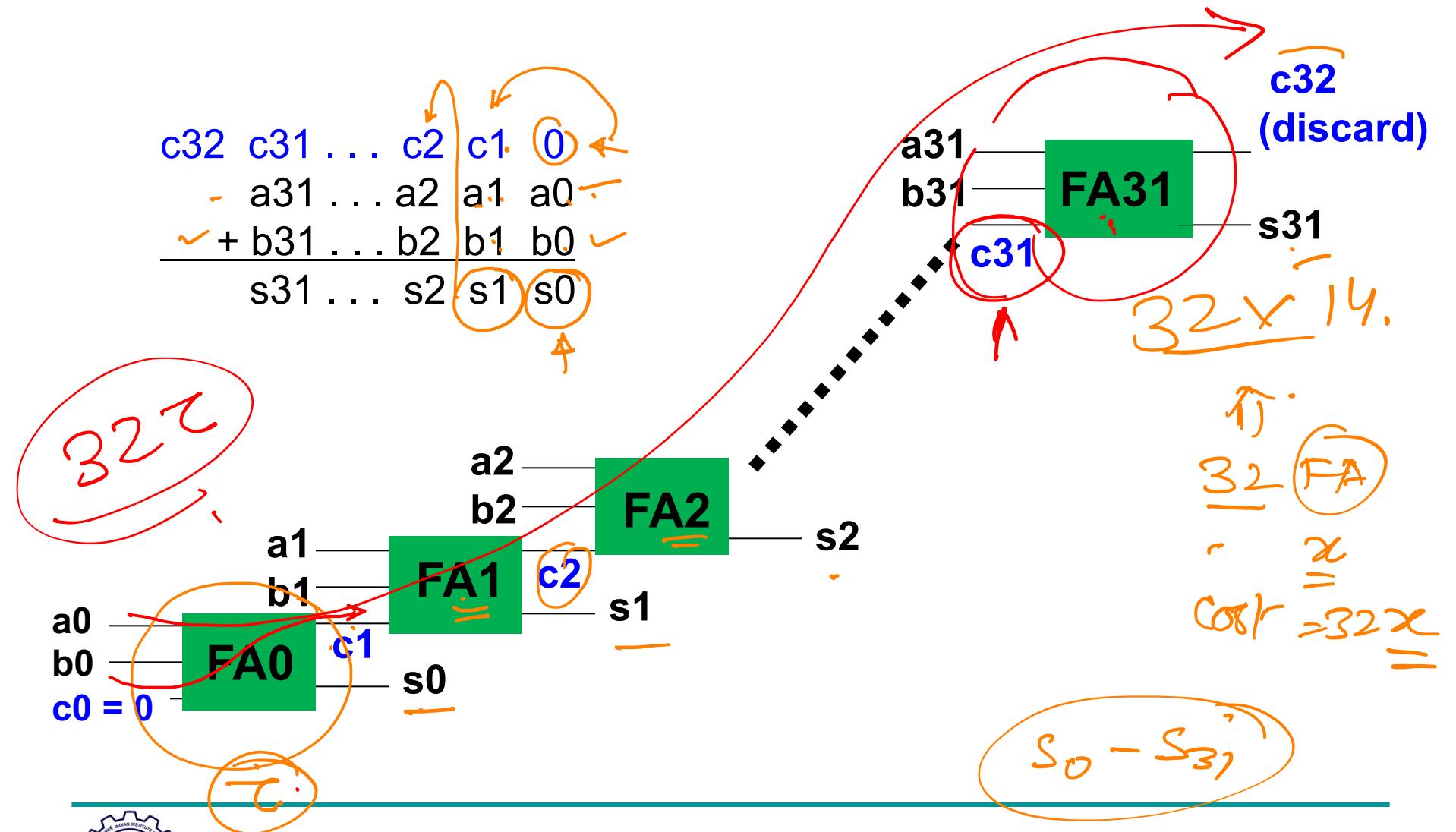
$$4+2+2 = 8$$

14



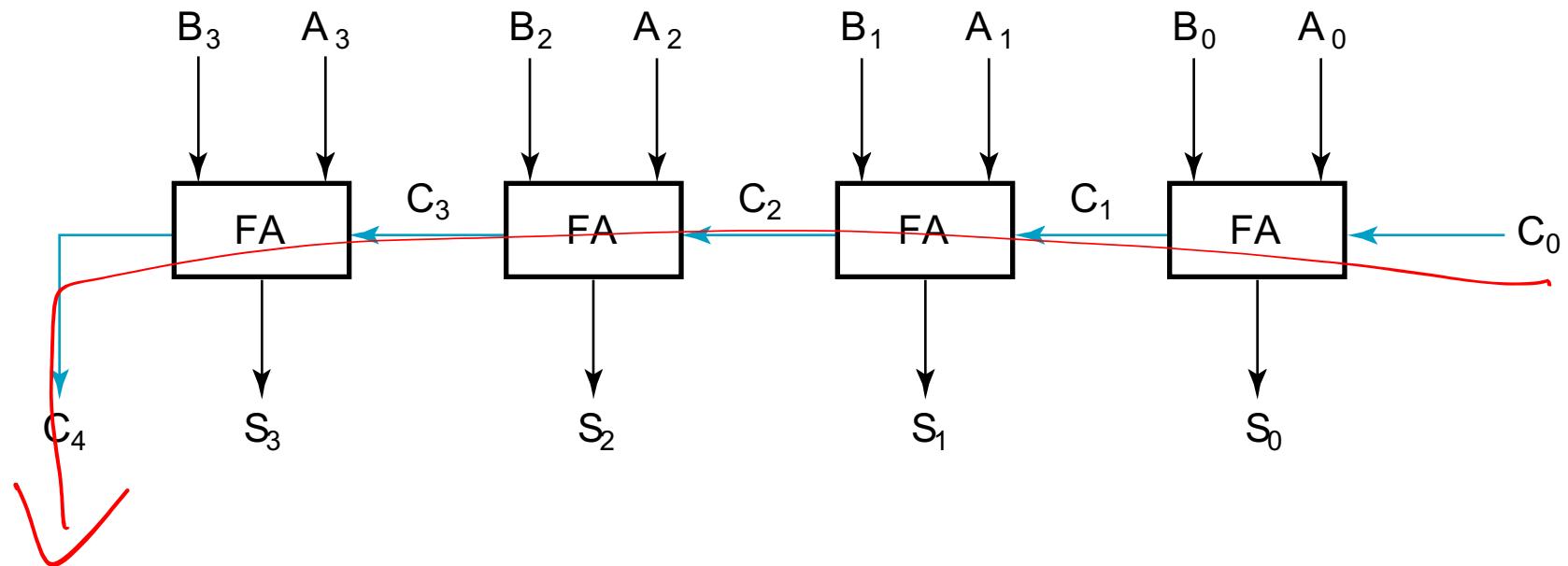
OR  $\propto N$   
AND  $\propto N$   
XOR  $\propto 2N$

# 32-bit Ripple-Carry Adder



# 4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



# How Fast is Ripple-Carry Adder?

---

- Longest delay path (critical path) runs from  $(a_0, b_0)$  to  $\text{sum31/C32}$

        
        
        
        
3.2 x 10<sup>-8</sup>

- Suppose delay of full-adder is 100ps

$$= 3.2 \times 10^{-8}$$

- Critical path delay = 3,200ps

- Must use more efficient ways to handle carry

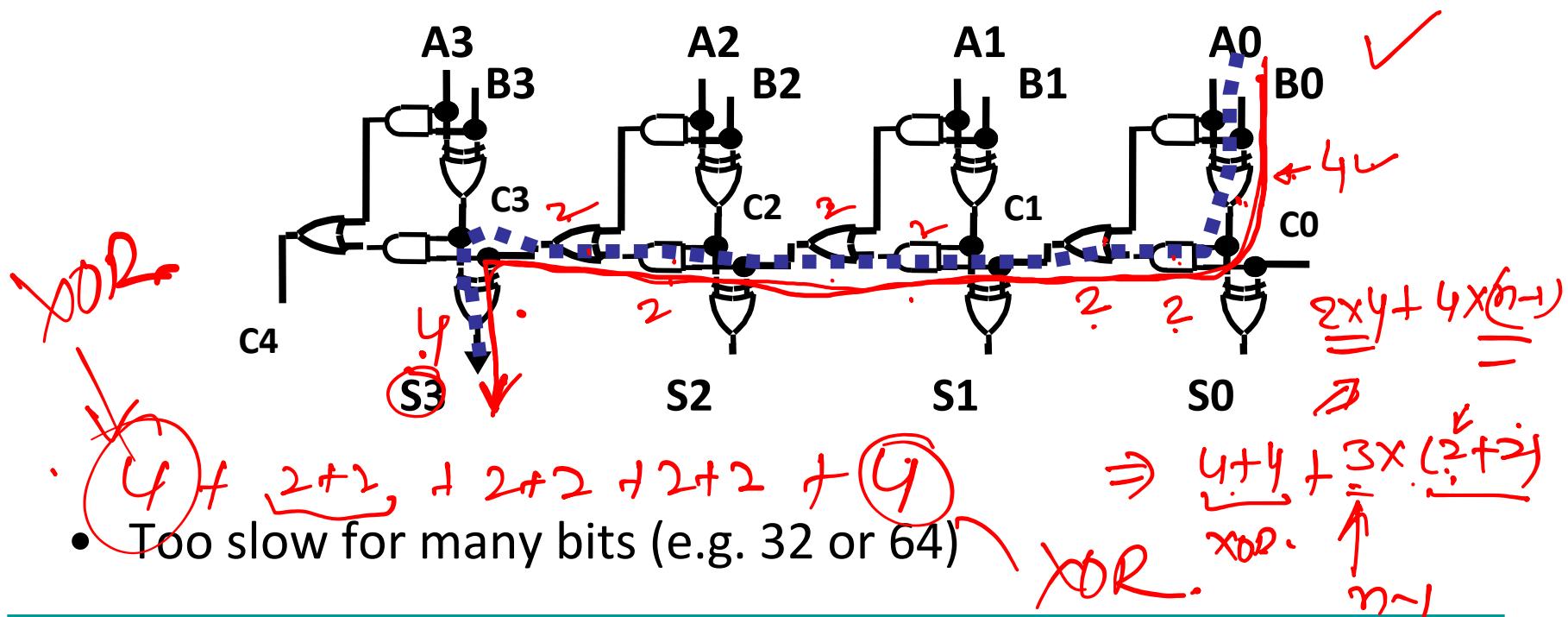


# Carry Propagation & Delay

OR · A N  
ANDAN

- Propagation delay:
    - Carry must ripple from LSB to MSB.
  - The gate-level propagation path for a 4-bit ripple carry adder of the last example:

$$\text{XOR} \propto 2N'$$



# Carry Propagation & Delay

$$\begin{aligned} & \text{32 bits} \\ & = 4 \times 1 + 6 \times 3 \quad \text{---} \\ & = 8 + 12 \quad = 132 \quad \checkmark \end{aligned}$$

10 ps

$$\begin{aligned} & = 1320 \text{ ps} \quad = 1.32 \text{ ns} \\ & = \frac{1}{1.32 \times 10^9} = \end{aligned}$$



# Carry Propagation & Delay

---



01 Feb 2022

CS-230@IITB

21

**CADSL**

# Ripple Carry Adder

- Easy to create
  - Good hierarchy
  - Tileable structures
- Small hardware

- Slow!

- Design is limited by the delays in propagating carry through all of the bitwise additions
- The output bit at position m is not valid until after the carry out of the m-1 position is ready

32 FA      18 FA

$$= \lceil \frac{8 + 4 \times (n-1)}{2} \rceil \times n$$



# Thank You



# Arithmetic Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee{iitb.ac.in}](mailto:viren@cse, ee{iitb.ac.in})

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 13 (03 February 2022)

**CADSL**

# Arithmetic

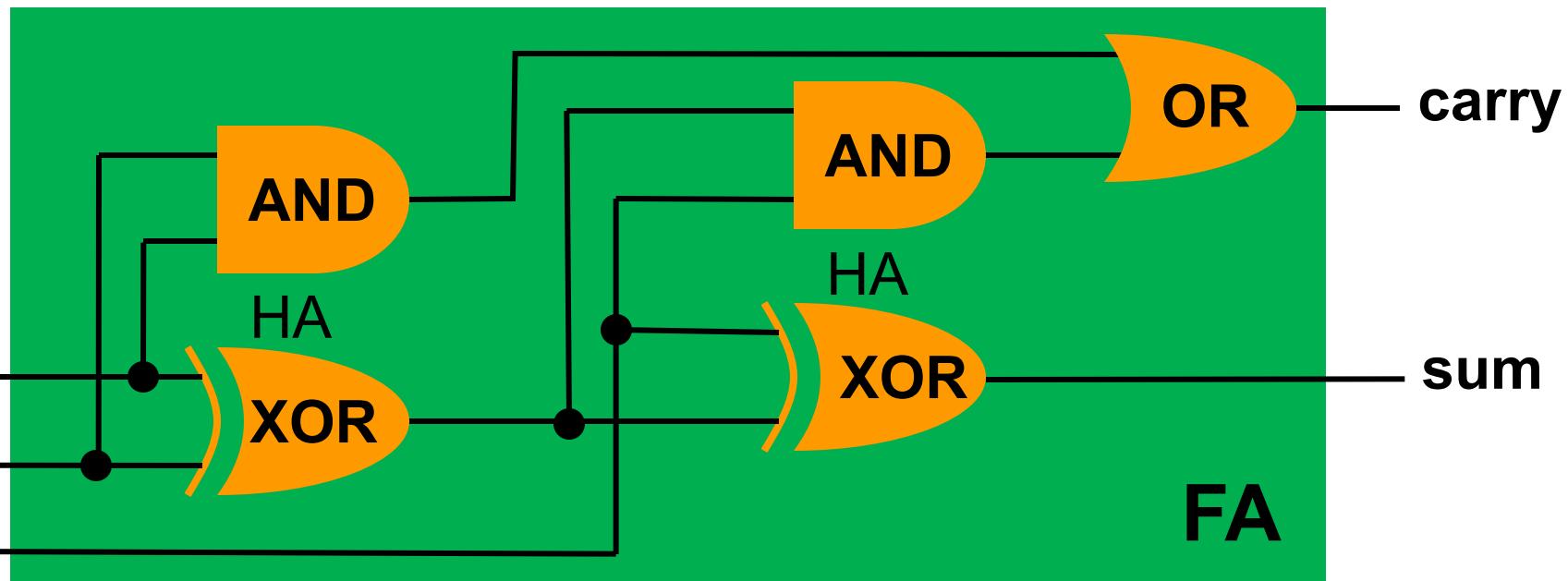
## Circuits:

(Adders)



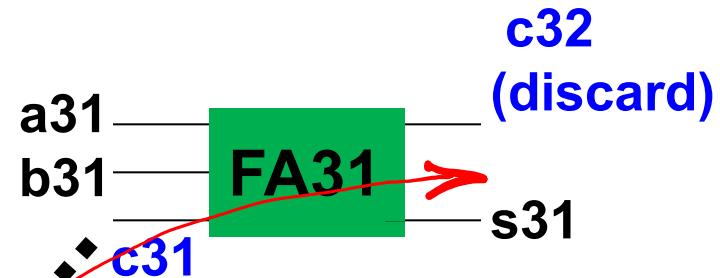
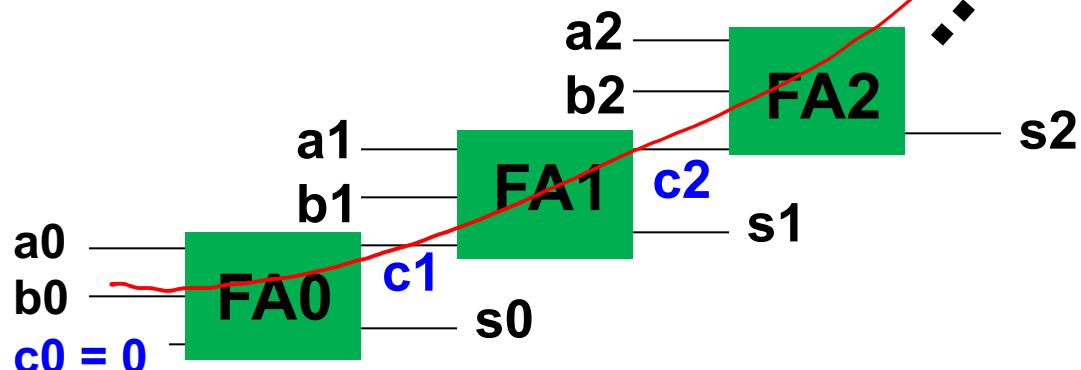
# Full-Adder Adds Three Bits

---



# 32-bit Ripple-Carry Adder

$$\begin{array}{r} c_{32} \ c_{31} \dots \ c_2 \ c_1 \ 0 \\ a_{31} \dots \ a_2 \ a_1 \ a_0 \\ + b_{31} \dots \ b_2 \ b_1 \ b_0 \\ \hline s_{31} \dots \ s_2 \ s_1 \ s_0 \end{array}$$



cost  $\propto n$   
 $\propto n \cdot (\text{cost of FA})$

delay  
 $\Rightarrow \text{delay} \propto n$

Cut down carry propagation path



# Ripple Carry Adder

---

- Easy to create
  - Good hierarchy
  - Tileable structures
- Small hardware
- Slow!
  - Design is limited by the delays in propagating carry through all of the bitwise additions
  - The output bit at position  $m$  is not valid until after the carry out of the  $m-1$  position is ready

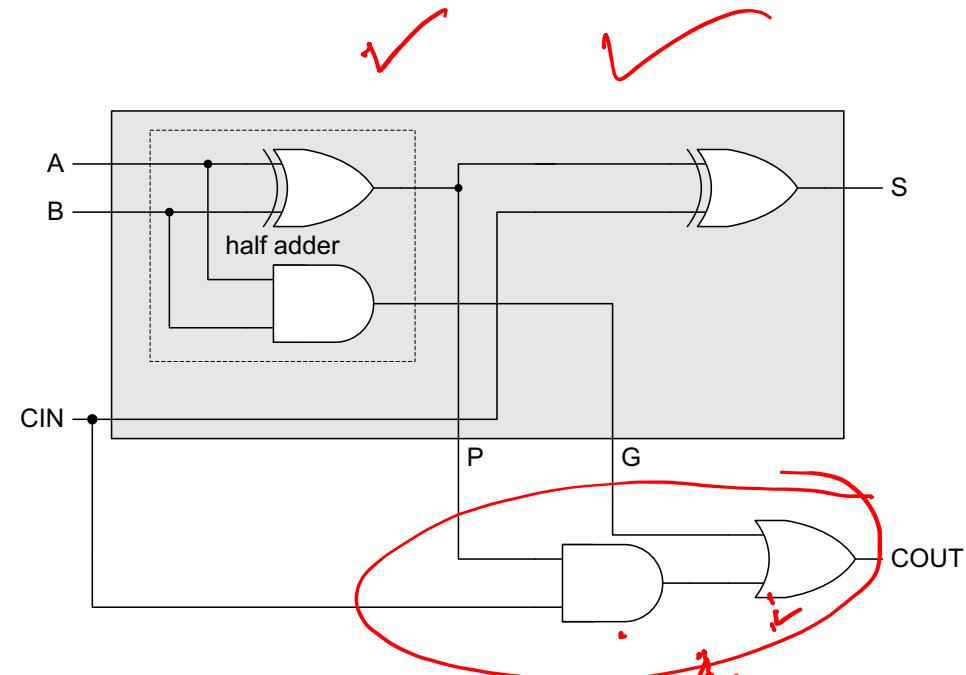
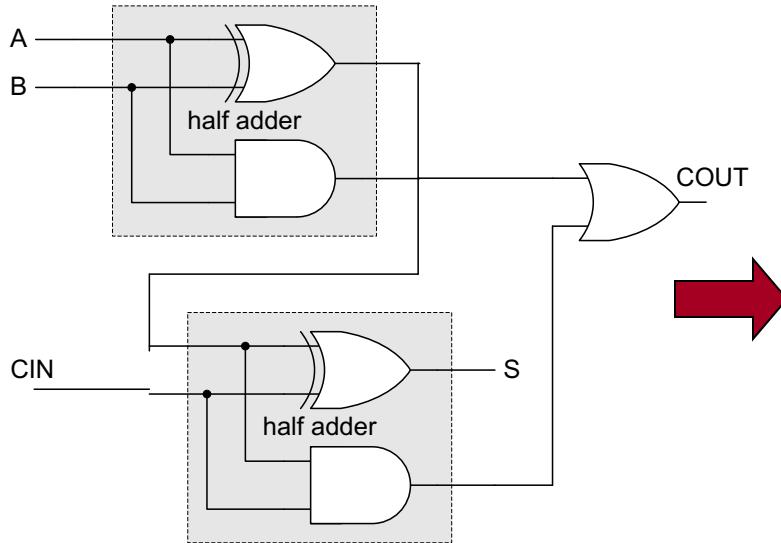


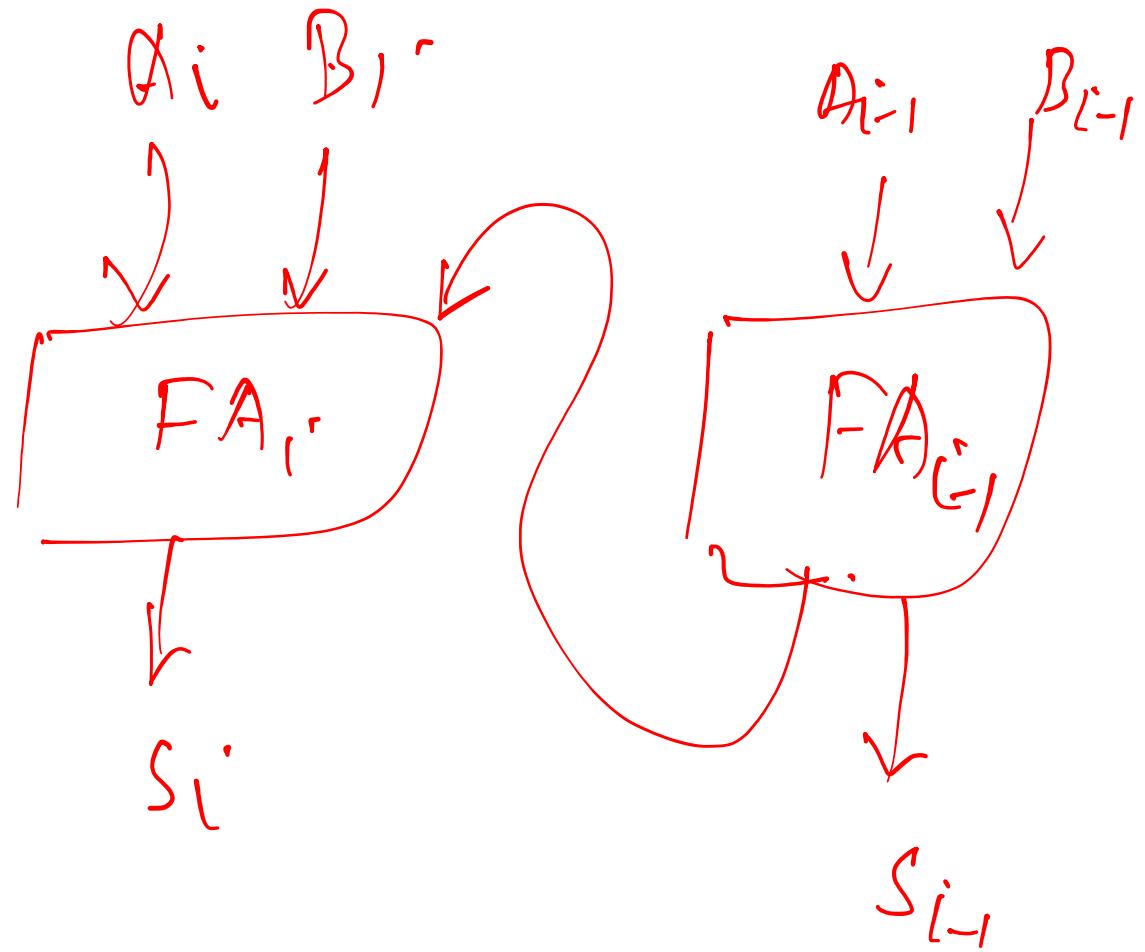
# FAST ADDERS



# Faster Addition

- For big adders, the carry-chain is very long
- Separate the carry chain and sum logic
- Partial Full Adders
  - Contain only the sum part of a FA

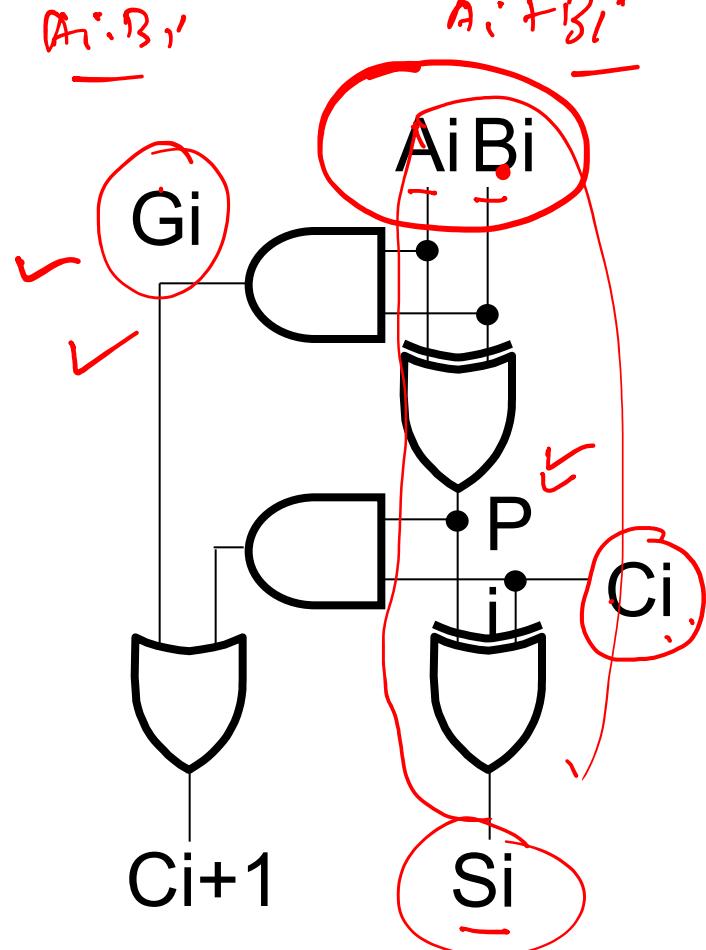




# Carry Lookahead

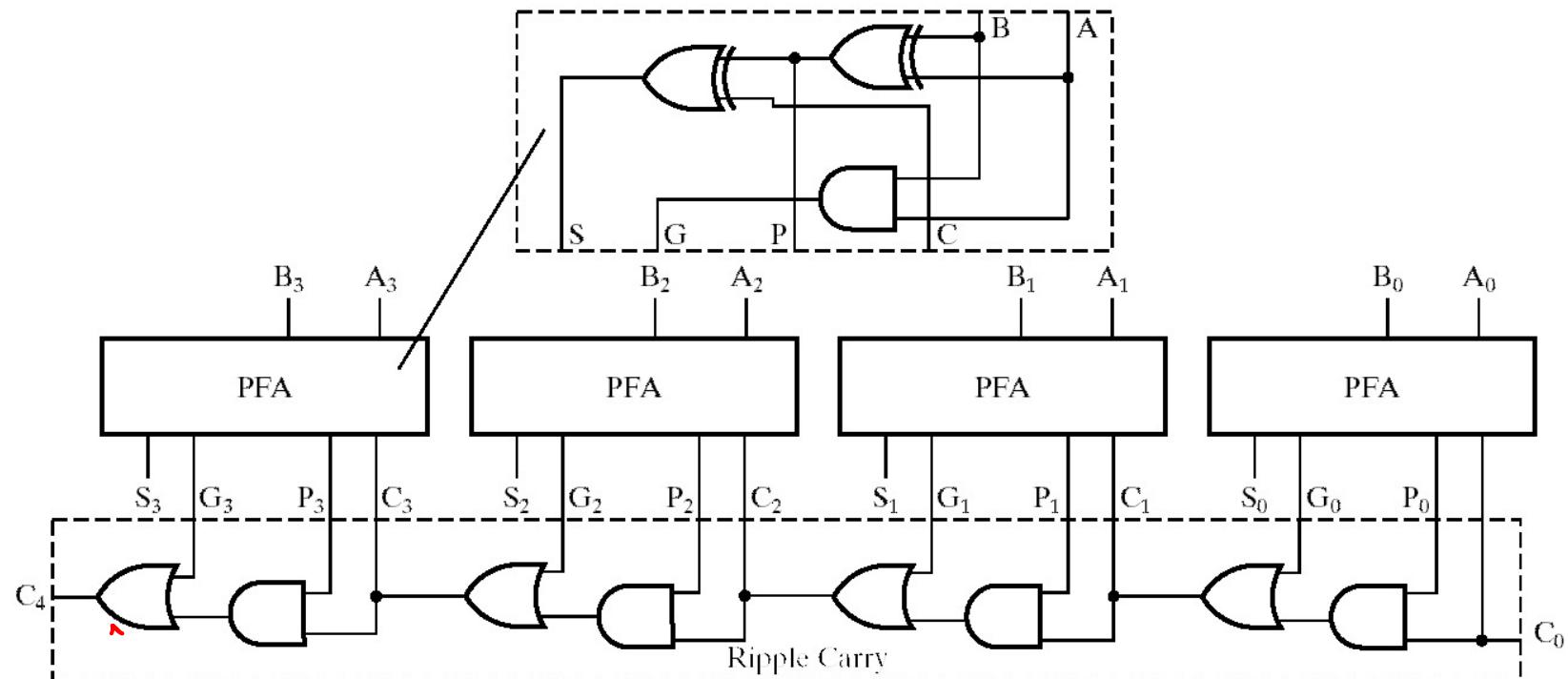
A:  $DB_i'$

- Given Stage  $i$  from a Full Adder, we know that there will be a carry generated when  $A_i = B_i = "1"$ , whether or not there is a carry-in.
- Alternately, there will be a carry propagated if the “half-sum” is “1” and a carry-in,  $C_i$  occurs.
- These two signal conditions are called generate, denoted as  $G_i$ , and propagate, denoted as  $P_i$  respectively and are identified in the circuit:



# Reassembling RCA Using PFAs

- Can create ripple-carry adder with PFAs
  - G: **Generates** a carry at this position
  - P: **Propagates** a carry through this position



# Carry Lookahead (continued)

- In the ripple carry adder:
  - $G_i$ ,  $P_i$ , and  $S_i$  are local to each cell of the adder
  - $C_i$  is also local each cell
- In the carry lookahead adder, in order to reduce the length of the carry chain,  $C_i$  is changed to a more global function spanning multiple cells
- Defining the equations for the Full Adder in term of the  $P_i$  and  $G_i$ :

$$P_i = \underline{A_i} \oplus \underline{B_i}$$
$$S_i = P_i \oplus C_i$$

$$G_i = \underline{\underline{A_i} \underline{\underline{B_i}}}$$
$$C_{i+1} = \underbrace{G_i}_{\text{---}} + \underbrace{P_i}_{\text{---}} \underbrace{C_i}_{\text{---}}$$

$$A_i \oplus B_i \oplus C_i$$



# Carry Lookahead Development

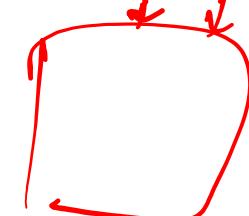
- Flatten equations for carry using  $G_i$  and  $P_i$  terms for less significant bits

- Beginning at the cell 0 with carry in  $C_0$ :

$$C_1 = \underline{G_0} + \underline{P_0} \underline{C_0}$$

$$Q_0 = \dot{A}_0 \cdot B_0.$$

$$\underline{P_0} = \dot{A}_0 \oplus B_0.$$



$$C_2 = G_1 + P_1 C_1 \quad C_1 = G_1 + P_1(G_0 + P_0 C_0)$$

$$= G_1 + P_1 G_0 + P_1 P_0 C_0$$



# Carry Lookahead Development

$$C_3 = G_2 + P_2 \quad C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0 C_0)$$

$$= \textcircled{G_2} + P_2 \cancel{G_1} + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

4z

✓  $C_4 = G_3 + P_3 \quad C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1$

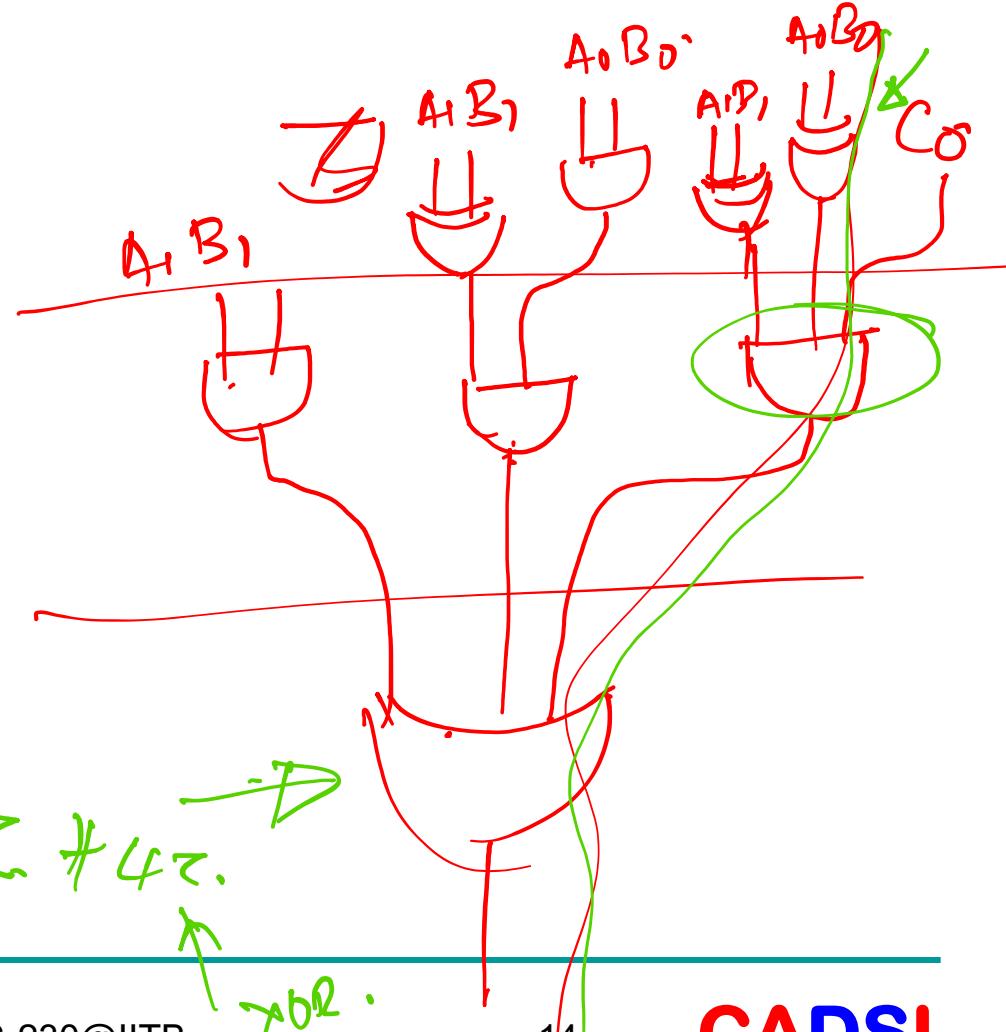
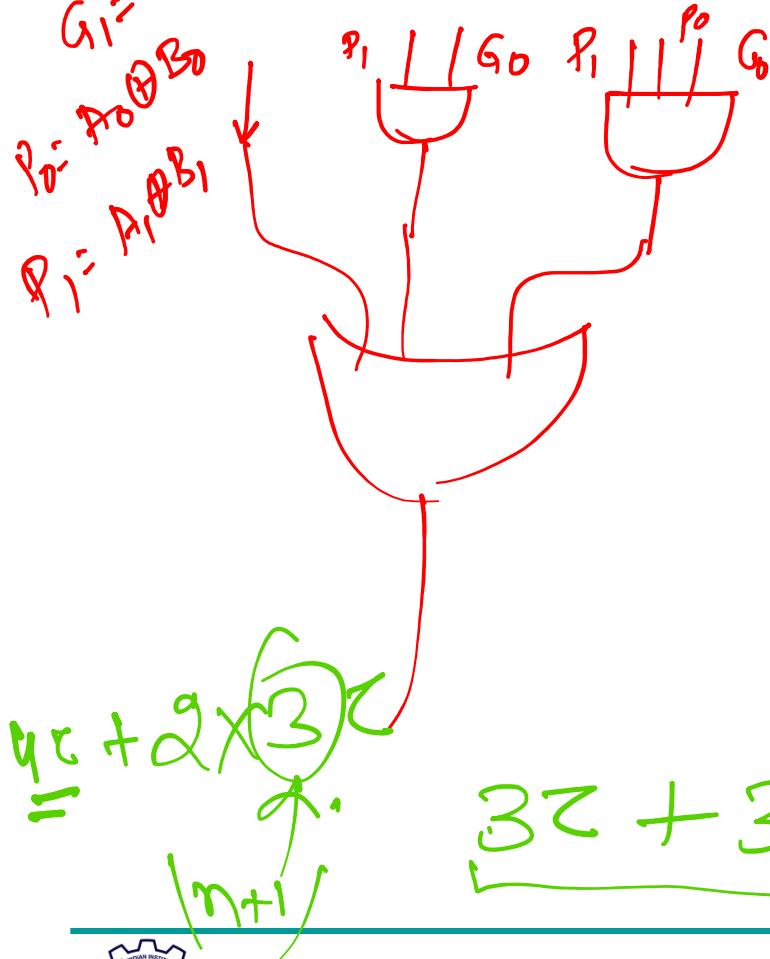
$$+ P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$



# Carry Lookahead Adder

$$\begin{aligned}G_0 &= A_0 \bar{B}_0 \\G_1 &= A_1 \bar{B}_1 \\P_0 &= P_0 \oplus B_0 \\P_1 &= A_0 \oplus B_1\end{aligned}$$

$$G_1 + P_1 G_0 + P_1 P_0 C_0$$



# Carry Lookahead Adder

delay  $\propto n$

$4\tau + (n+1)\tau$

$= 5\tau + (n)\tau$

$\Rightarrow$  CMOS

BJT

TTL

delay is independent of input.

constant delay.



# Carry Look-Ahead Adder

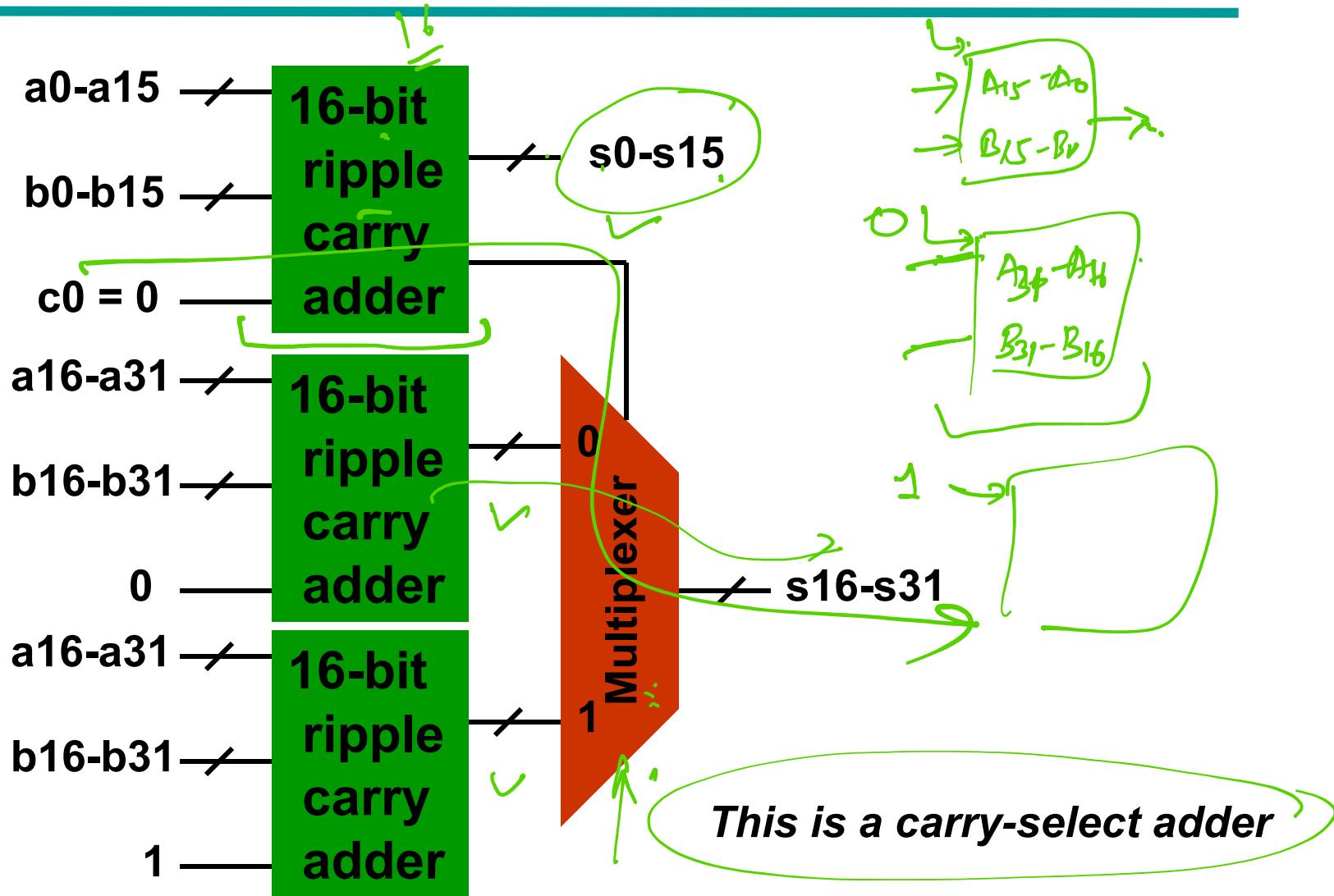
---

- As usual, can trade area/power for speed
  - Multi-level logic (ripple carry) reduces area
  - Flattening carry logic increases speed
- Sometimes called CLA

Break carry chain



# Speeding Up the Adder



# Fast Adders

---

- In general, any output of a 32-bit adder can be evaluated as a logic expression in terms of all 65 inputs.  

- Number of levels of logic can be reduced to  $\log_2 N$  for N-bit adder. Ripple-carry has N levels.
- More gates are needed, about  $\log_2 N$  times that of ripple-carry design.





# Prefix Computation

# Key Architectures for Carry Calculation

---

- 1960: J. Sklansky adder
- 1973: Kogge-Stone adder ✓
- 1980: Ladner-Fisher adder
- 1982: Brent-Kung adder
- 1987: Han Carlson adder
- 1999: S. Knowles adder

Other parallel adder architectures:

- 1981: H. Ling adder
- 2001: Beaumont-Smith



# Binary Addition

---

- **Input:** two  $n$ -bit binary numbers  $a_{n-1} \dots a_1 a_0$  and  $b_{n-1} \dots b_1 b_0$ , one bit carry-in  $c_0$
- **Output:**  $n$ -bit sum  $s_{n-1} \dots s_1 s_0$  and one bit carry out  $c_n$
- Prefix Addition: Carry generation & propagation

Generate:  $g_i = a_i \cdot b_i$

Propagate:  $p_i = a_i \oplus b_i$

$$c_{i+1} = g_i + p_i \cdot c_i$$

$$s_i = c_i \oplus (a_i \oplus b_i)$$



# Binary Addition as a prefix sum problem.

$$\begin{array}{r} a_3 \quad a_2 \quad a_1 \quad a_0 \\ + b_3 \quad b_2 \quad b_1 \quad b_0 \\ \hline \end{array}$$

A stage  $i$  will generate a carry if  
 $g_i = a_i \cdot b_i$   
and propagate a carry if  
 $p_i = \text{XOR}(a_i, b_i)$   
Hence for stage  $i$ :  
 $c_i = g_i + p_i c_{i-1}$

With:

Where:

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$
$$(G_0, P_0) = (g_0, p_0)$$

$\checkmark$   $(g_x, p_x) \circ (g_y, p_y) = (g_x \# p_x \cdot g_y, p_x \cdot p_y)$

We have:

$$(G_1, P_1) = (g_1, p_1)$$

$$(G_2, P_2) = (g_2, p_2) \circ (G_1, P_1) = (g_2 \# p_2 \cdot g_1, p_2 \cdot p_1)$$

$$\begin{aligned} (G_3, P_3) &= (g_3, p_3) \circ (G_2, P_2) = (g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1), p_3 \cdot p_2 \cdot p_1) \\ &= (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1, p_3 \cdot p_2 \cdot p_1) \end{aligned}$$

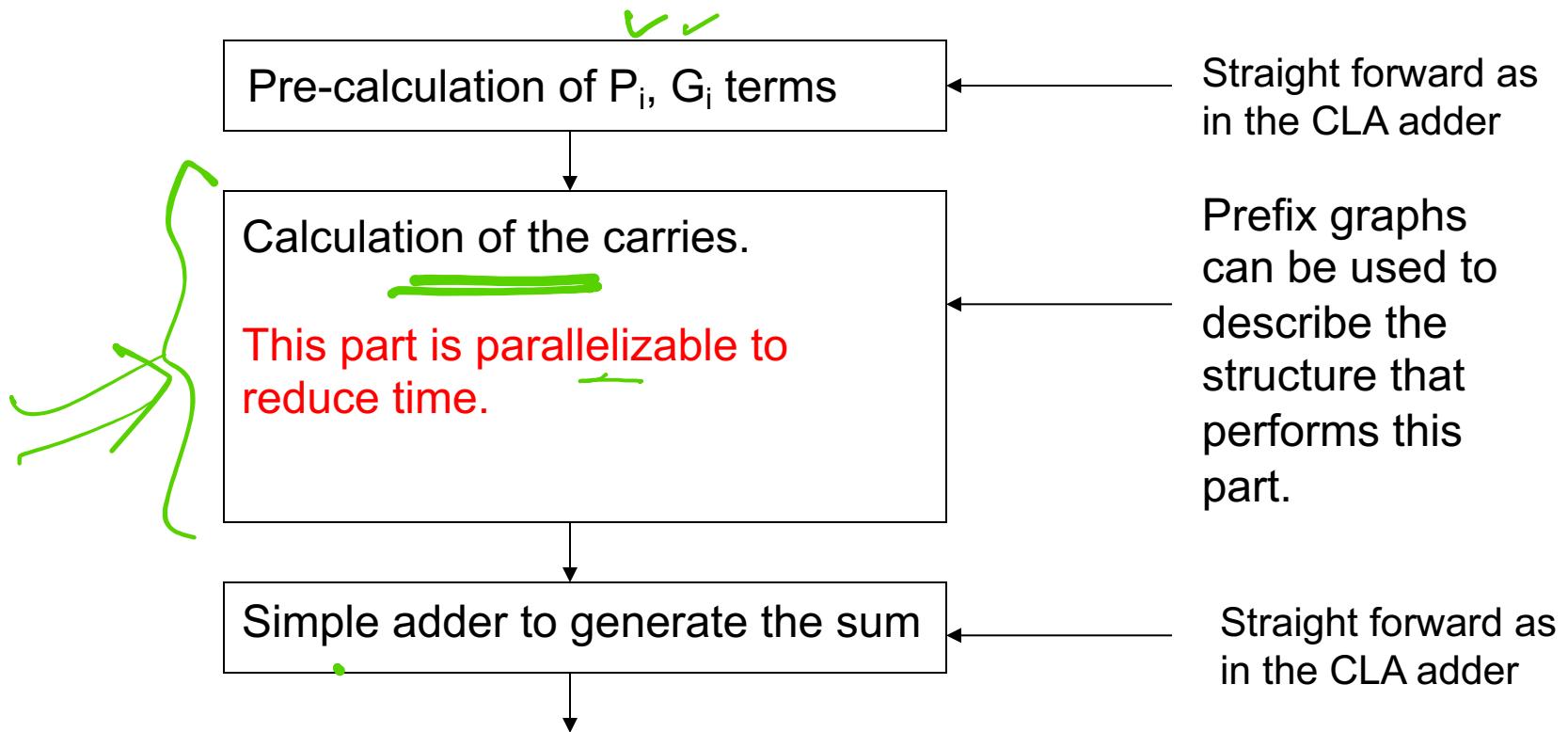
etc ...

... The familiar  
carry bit generating  
equations for stage  $i$   
in a CLA adder.



# Parallel Prefix Adders

- The parallel prefix adder employs the 3-stage structure of the CLA adder. The improvement is in the carry generation stage which is the most intensive one:



# Prefix Addition – Formulation

Pre-processing:

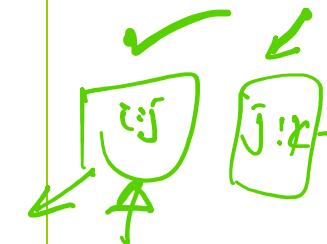
$$g_i = a_i b_i \quad p_i = a_i \oplus b_i$$

Prefix Computation:

$$\begin{aligned} G_{[i:k]} &= G_{[i:j]} + P_{[i:j]} G_{[j+1:k]} \\ P_{[i:k]} &= P_{[i:j]} P_{[j+1:k]} \end{aligned}$$

Post-processing:

$$\begin{aligned} c_{i+1} &= G_{[i:0]} + P_{[i:0]} \cdot c_0 \\ s_i &= p_i \oplus c_i \end{aligned}$$



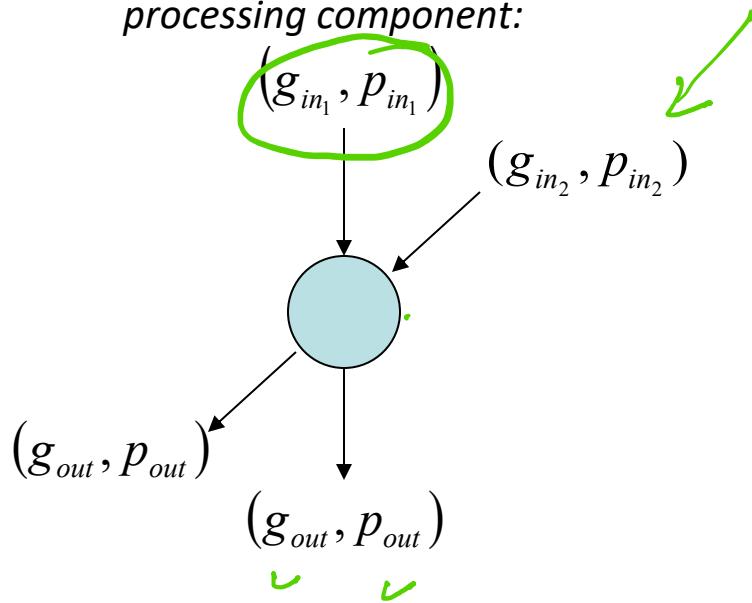
generated  
for preo  
group



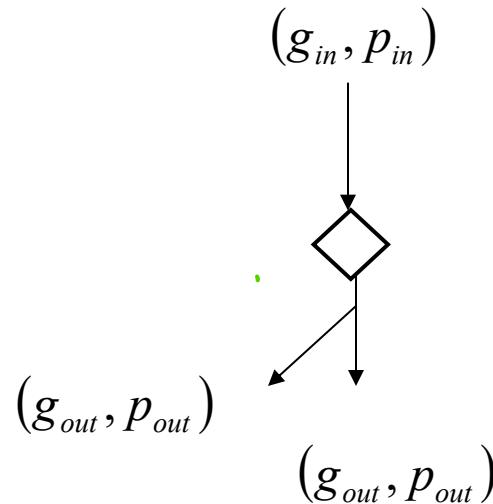
# Computation of Carries – Prefix Graphs

The components usually seen in a prefix graph are the following:

*processing component:*



*buffer component:*

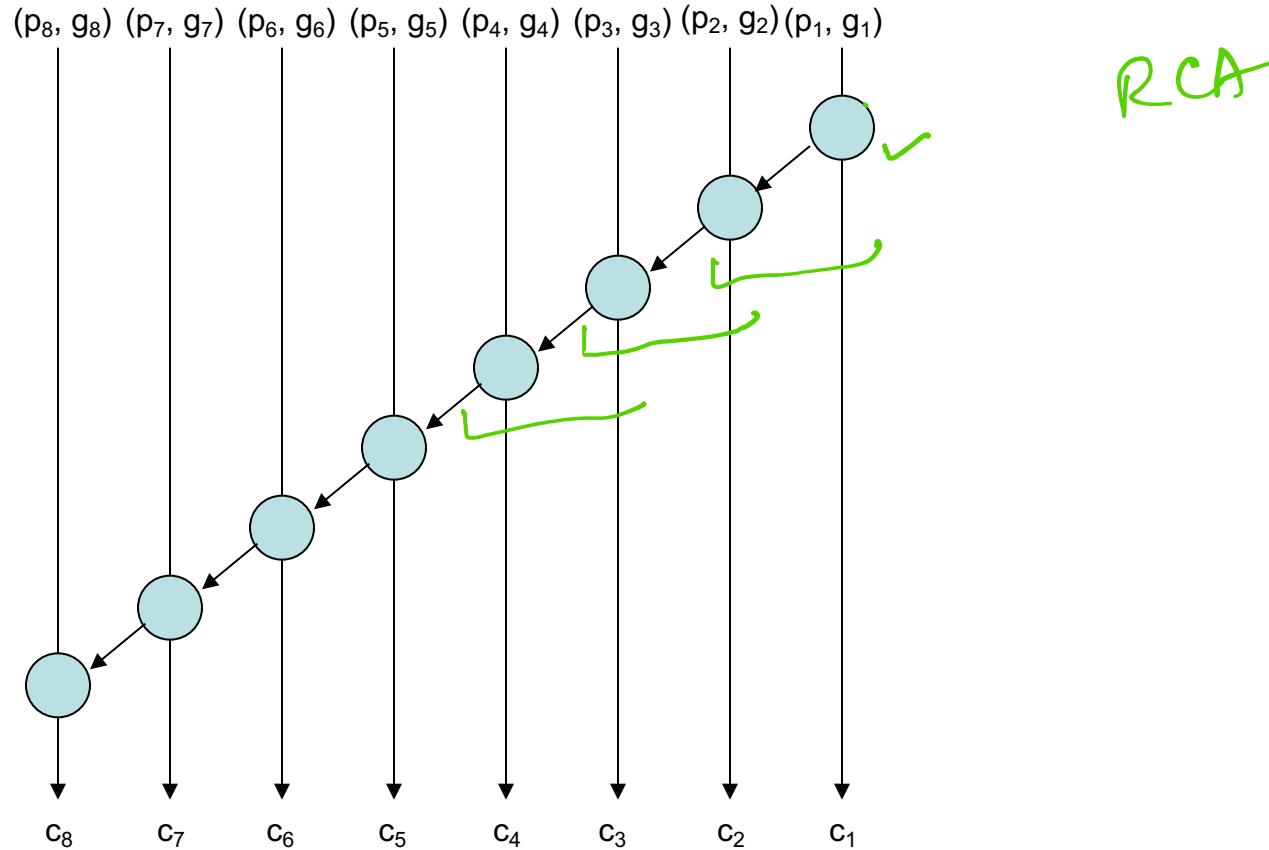


$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2})$$

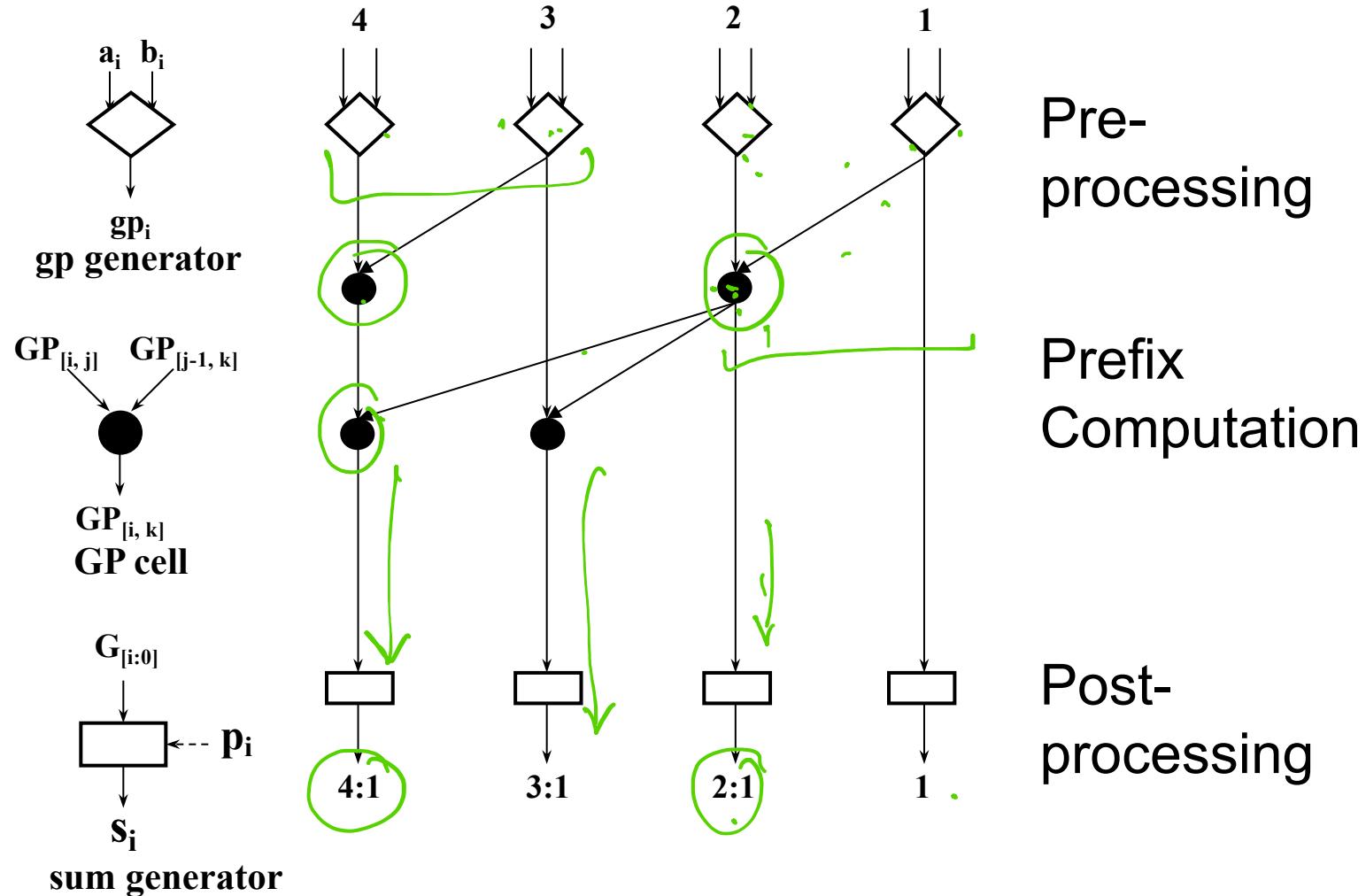
$$(g_{out}, p_{out}) = (g_{in}, p_{in})$$

# Prefix graphs for representation of Prefix addition ✓

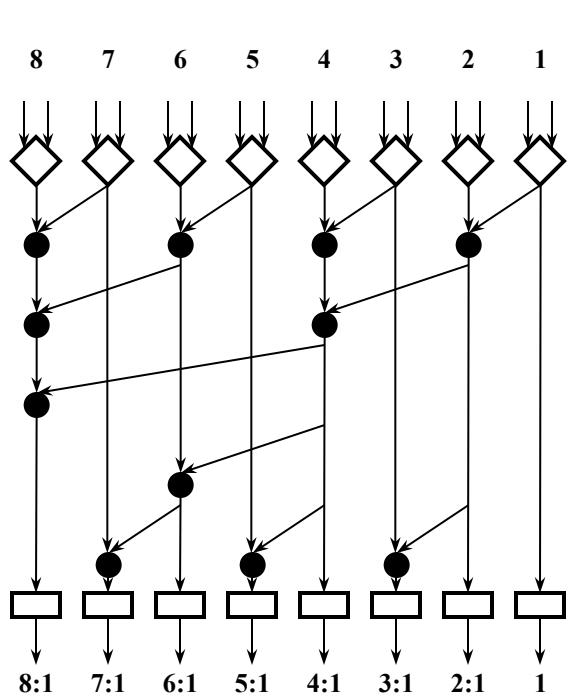
- Serial adder carry generation represented by prefix graphs



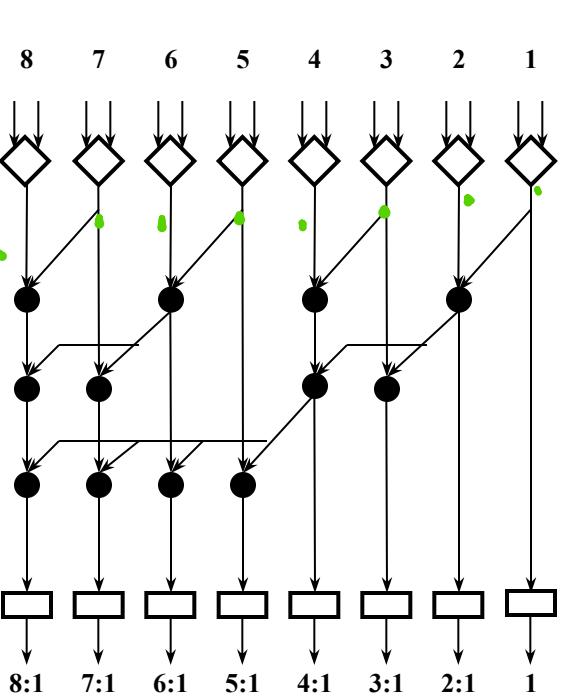
# Prefix Adder – Prefix Structure Graph



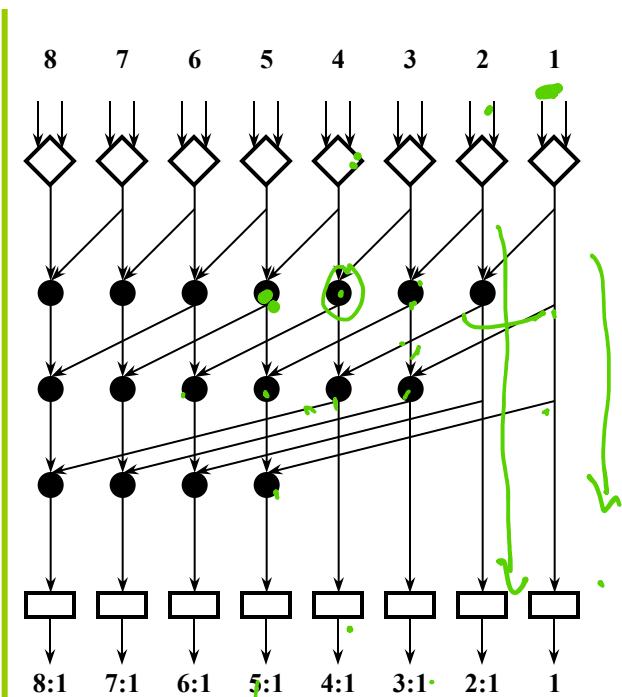
# Classical Prefix Adders



Brent-Kung:  
Logical levels:  $2\log_2 n - 1$   
Max fanouts: 2



Sklansky:  
Logical levels:  $\log_2 n$   
Max fanouts:  $n/2$

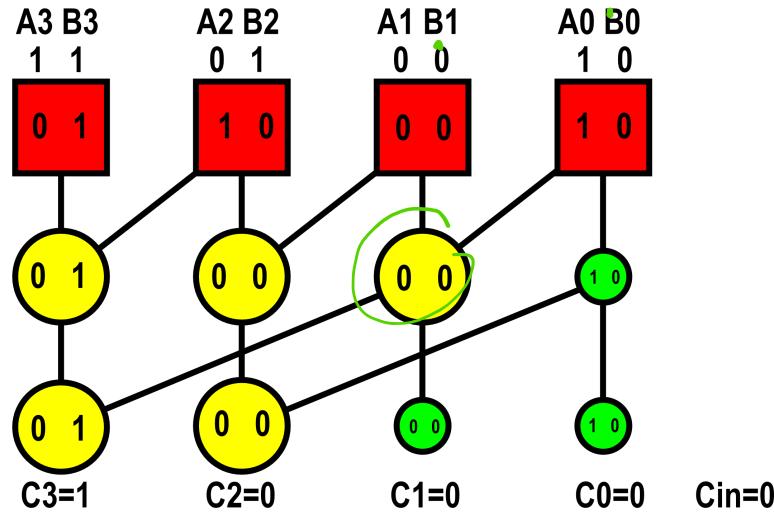


Kogge-Stone:  
Logical levels:  $\log_2 n$   
Max fanouts: 2



# Kogge Stone Adder

$$A = 1001 \quad B = 1100 \quad \text{Sum} = 10101$$



Legend:

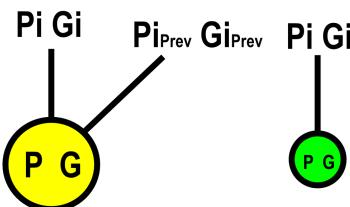


$$P = A_i \text{ xor } B_i$$

$$G = A_i \text{ and } B_i \quad G = (P_i \text{ and } G_{i-1}) \text{ or } G_i$$

$$C_i = G_i$$

$$S_i = P_i \text{ xor } C_{i-1}$$

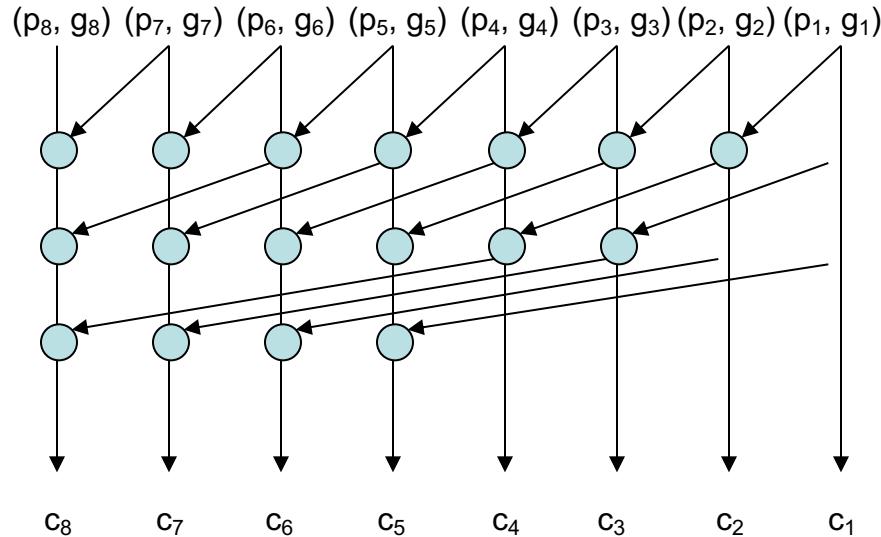


$$P = P_i$$

$$G = G_i$$

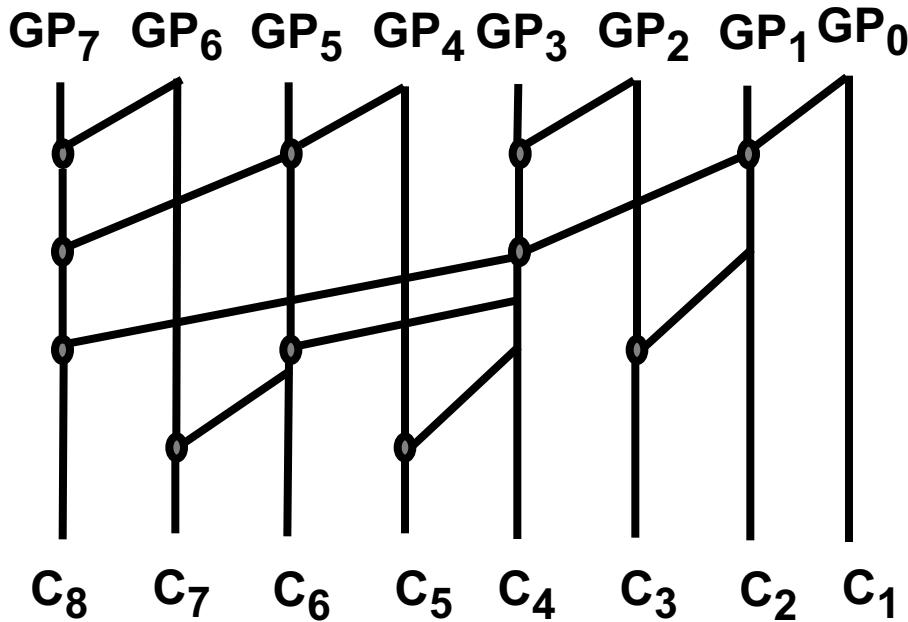


# 1973: Kogge-Stone adder



- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

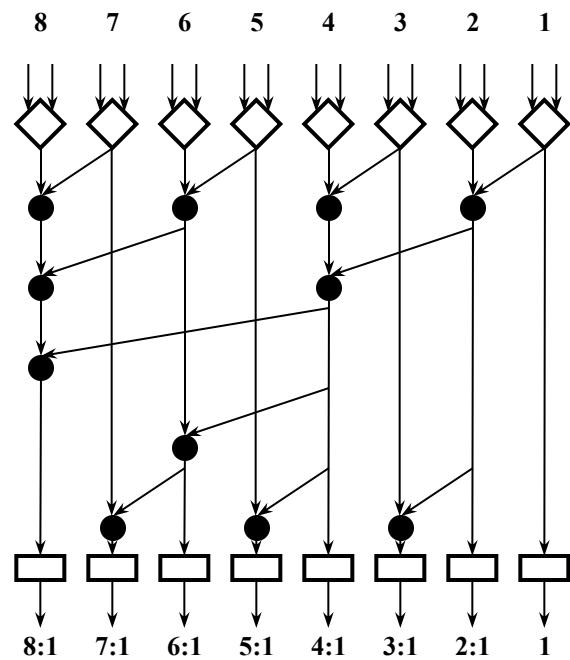
# 1982: Brent-Kung (BK) Adder



Brent and Kung, “A regular layout for parallel adders”, In IEEE transaction for Computers, 1982



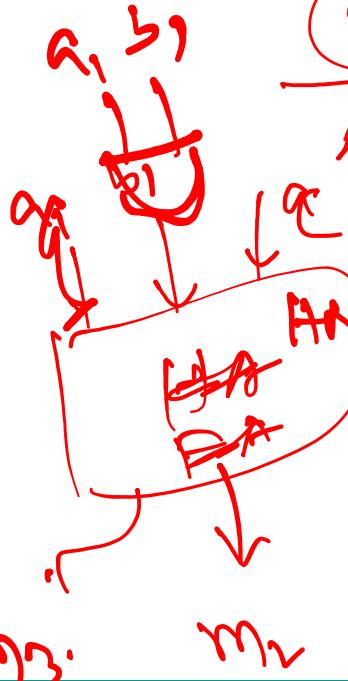
# Brent-Kung (BK) Adder



multiplier

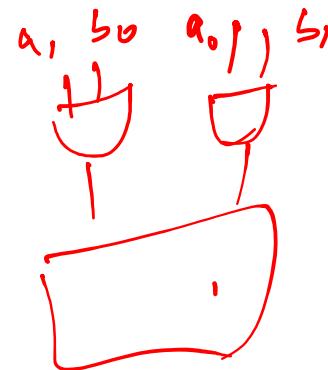
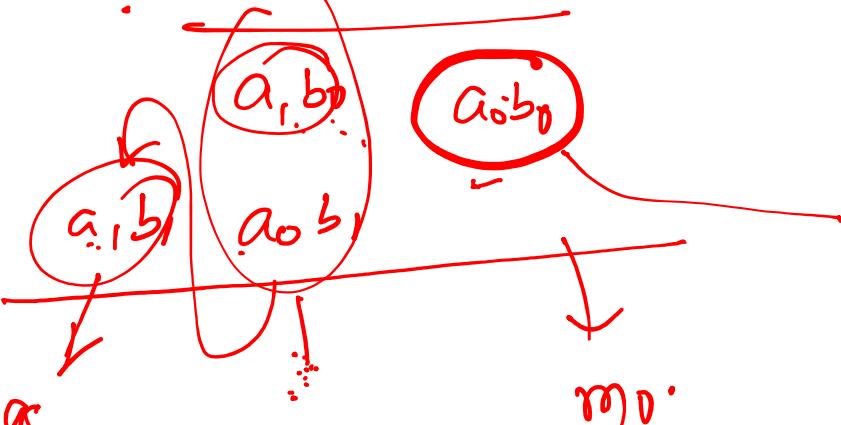
$A_i$





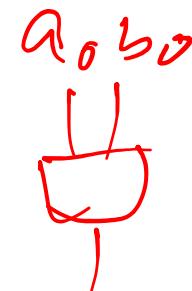
$$A = a_1 \oplus a_0$$

$$B: \quad b_1 \quad b_0$$



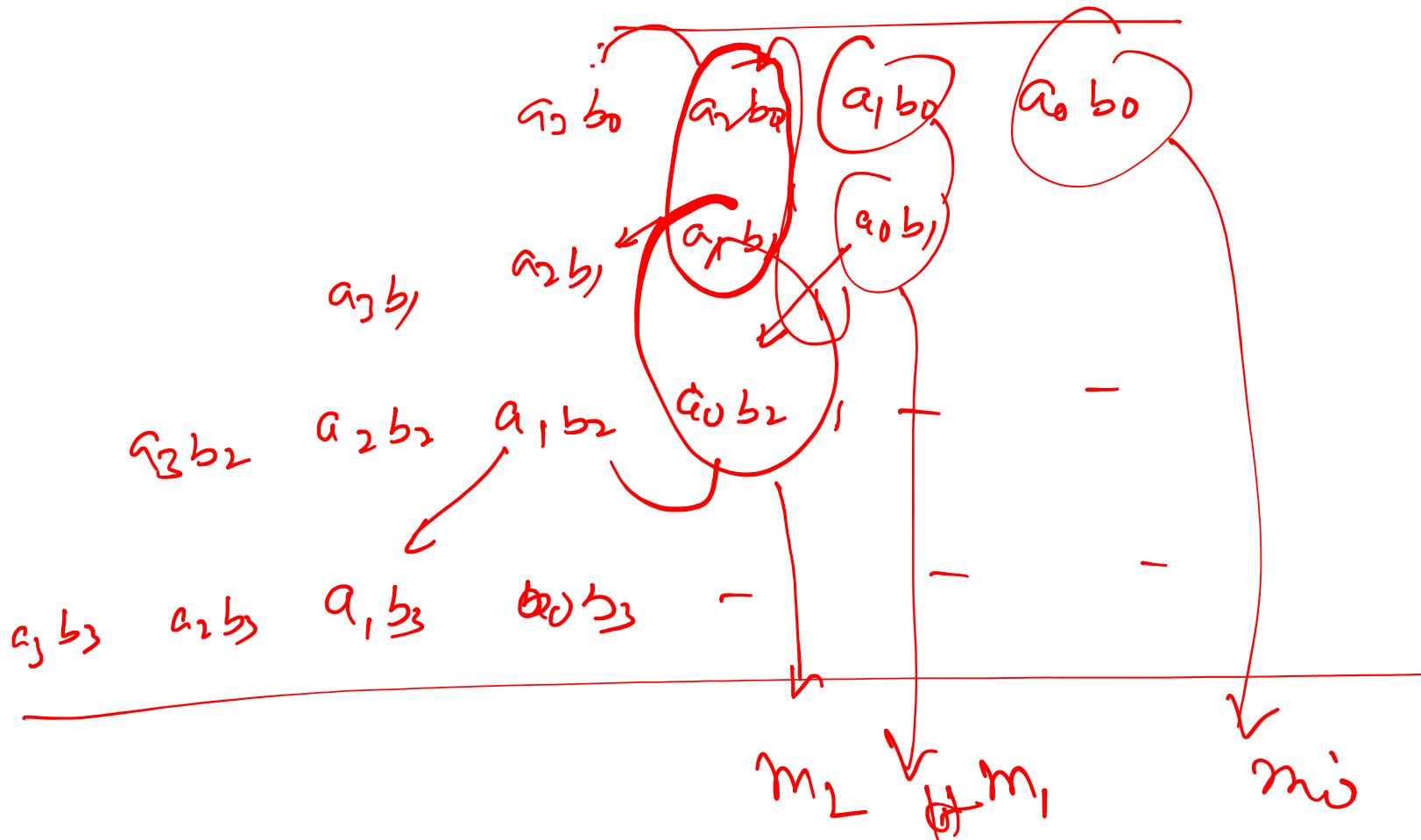
19

$$a_1 \bar{b}_1 + b_1 \bar{a}_1 \\ (a_0 \bar{b}_1)_f$$



$a_3 \ a_2 \ a_1 \ a_0$

$b_3 \ b_2 \ b_1 \ b_0$



Carry  
Save  
Doch.

$$\begin{array}{r} 0 \\ 278 \\ 151 \\ \hline 429 \end{array} \rightarrow \begin{array}{r} 278 \\ 151 \\ \hline 329 \end{array}$$
$$\begin{array}{r} 329 \\ 100 \leftarrow \\ \hline 429 \rightleftarrows \end{array}$$
$$\begin{array}{r} 278 \\ 151 \\ \hline 0.10 \leftarrow \end{array}$$

Array Multiplic



# Thank You



# Sequential Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 14 (07 February 2022)

**CADSL**

# Arithmetic

$C_1$  ✓  
 $C_2$   
 $C_3 = g_3 + p_1 \cdot C_2$   
 $C_4$

~~Multiplexers~~

## Circuits:

### Adders

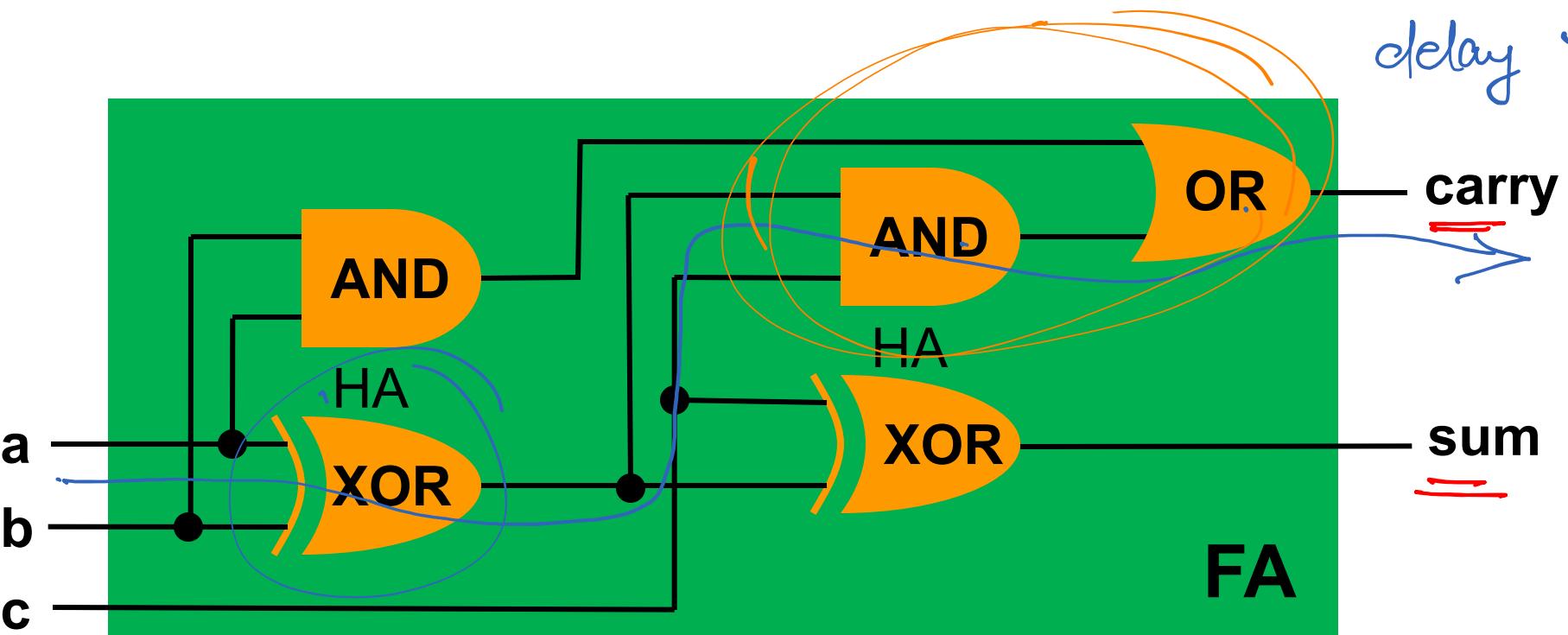
Adders  
✓ Area / Power  
RCA  
Performance  
CLA →  $\alpha n$

CSA - carry select

$\alpha k \log n$  ← Prefixor-Adder



# Full-Adder Adds Three Bits

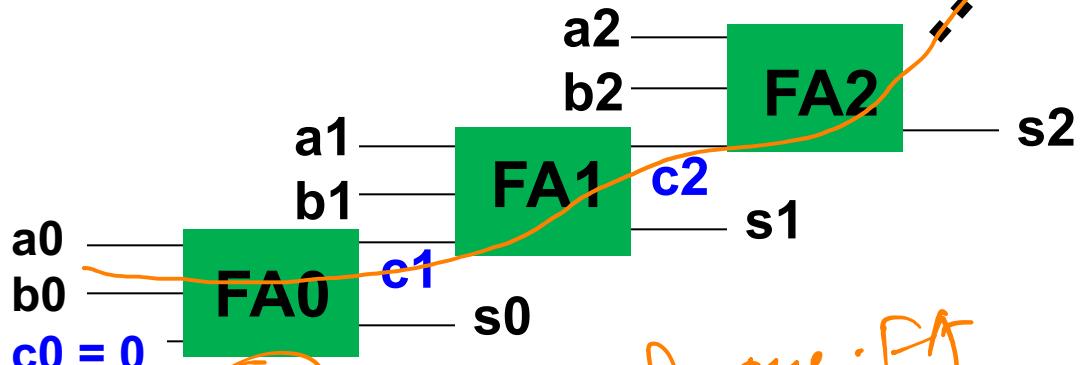


$$\begin{array}{c} 2 \times \text{XOR} + 2 \times \text{AND} + 1 \times \text{OR} \\ \hline \text{PA} - \oplus \stackrel{A}{=} \end{array}$$

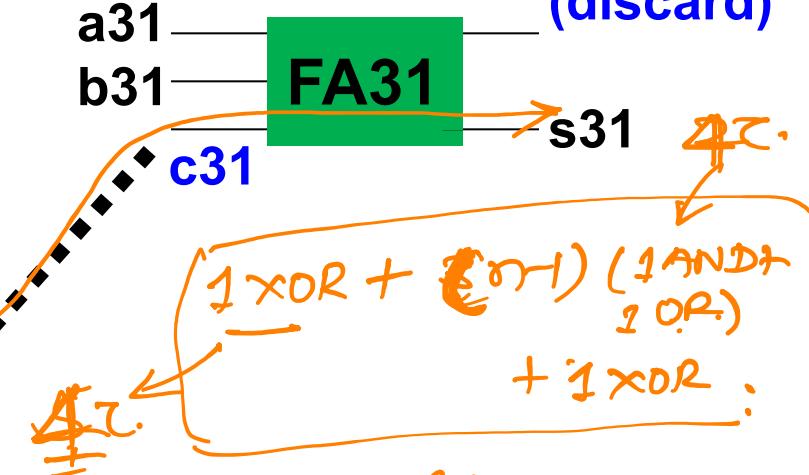
# 32-bit Ripple-Carry Adder

$c_{32}$   $s_{32}$

$$\begin{array}{r}
 c_{32} \ c_{31} \dots \ c_2 \ c_1 \ 0 \\
 a_{31} \dots \ a_2 \ a_1 \ a_0 \\
 + b_{31} \dots \ b_2 \ b_1 \ b_0 \\
 \hline
 s_{31} \dots \ s_2 \ s_1 \ s_0
 \end{array}$$



(1)  $\leftarrow$  delay  $\rightarrow$  one FA



$$4T + 4T + 4T(n-1)$$

$$= 8T + 4T + 4nT$$

$$= 6T + 4nT$$

$$\propto n$$

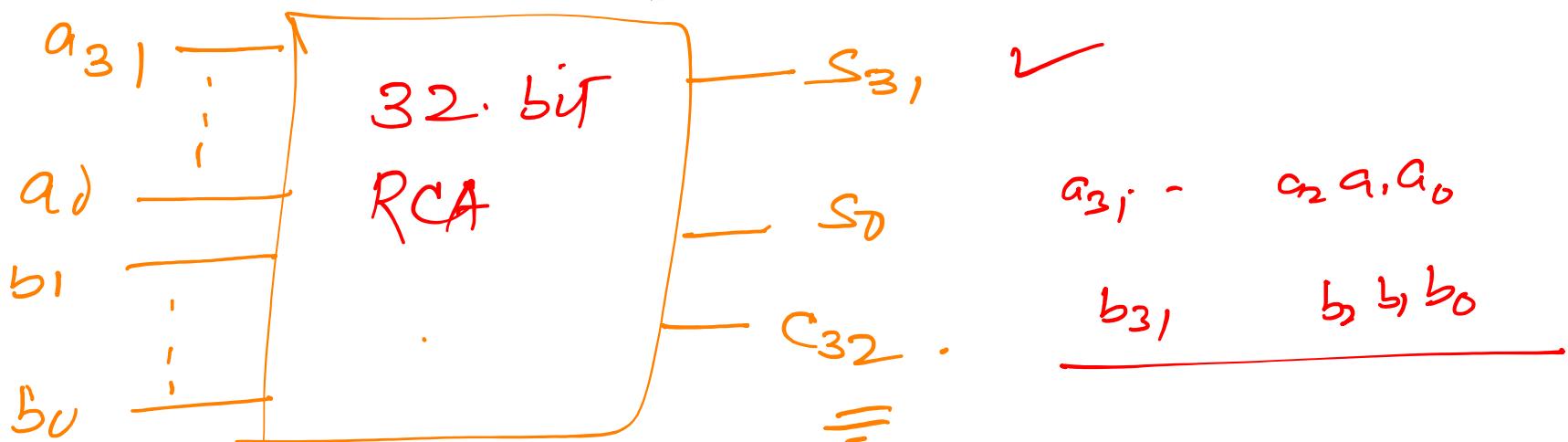
CADSL

$$4T = 40ps$$

$$32 \times 40 = 1280 ps$$

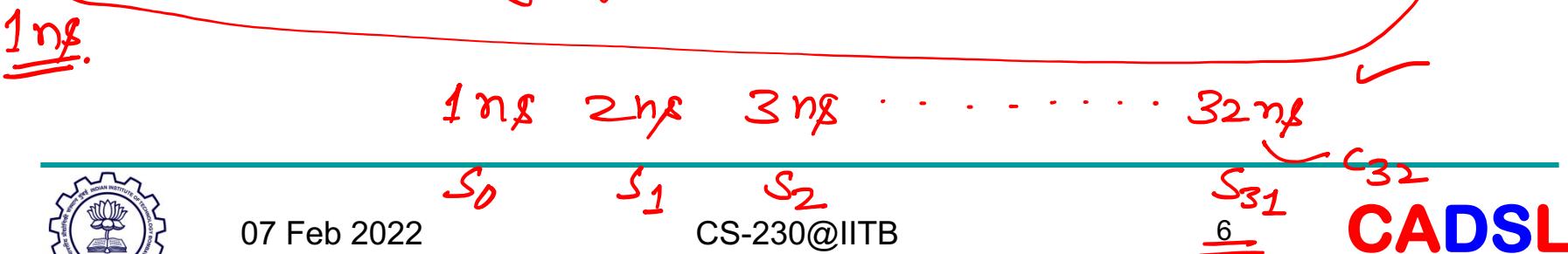
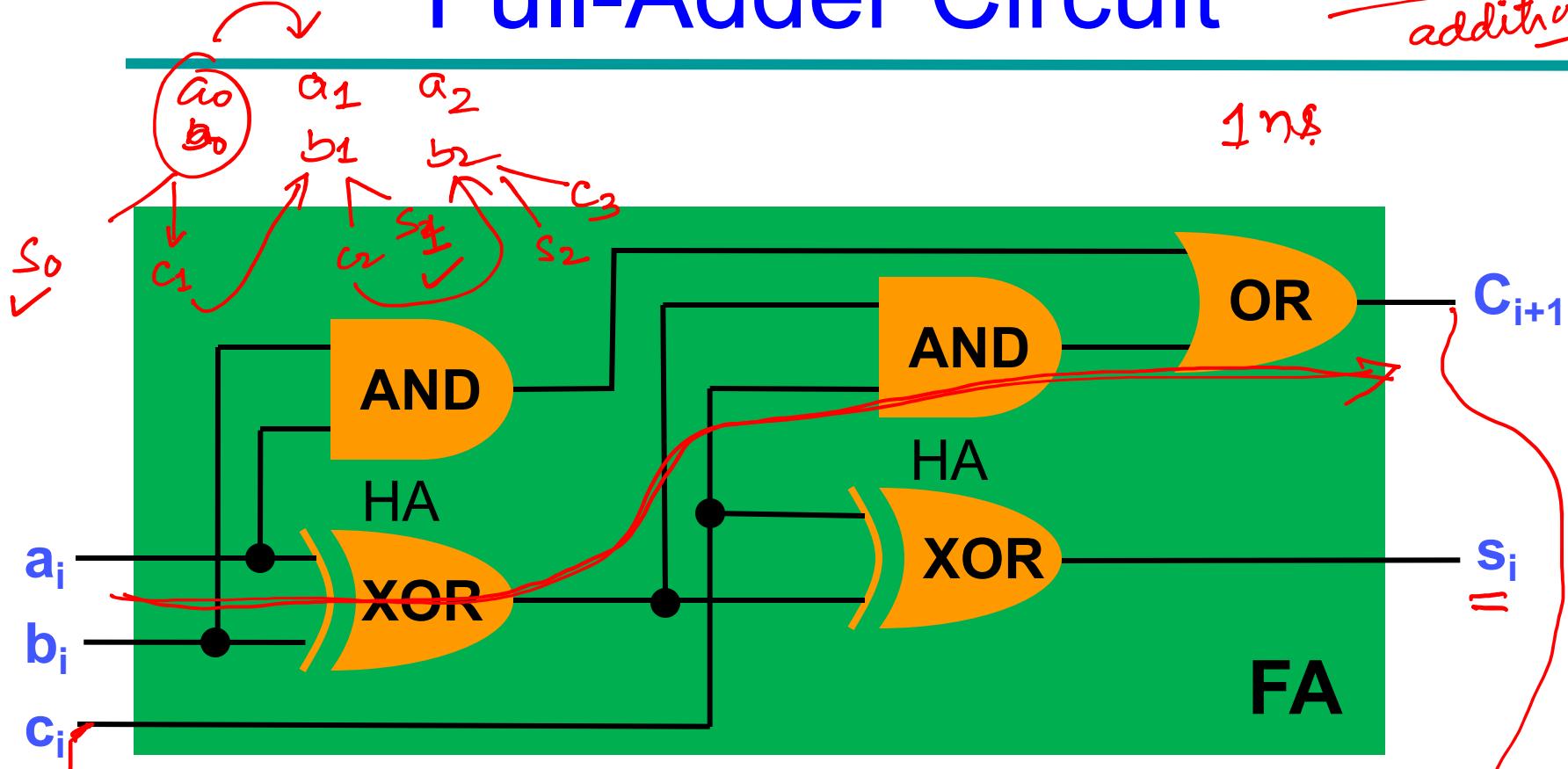


needs  $32 \cdot FA$        $32A =$   
 delay       $\underline{32} \times \underline{\text{delay of one } FA}$   
 $\uparrow$        $\uparrow 1 \text{ ns}$   
 $\checkmark$       result  
 $\checkmark$   $32 \text{ bits}$



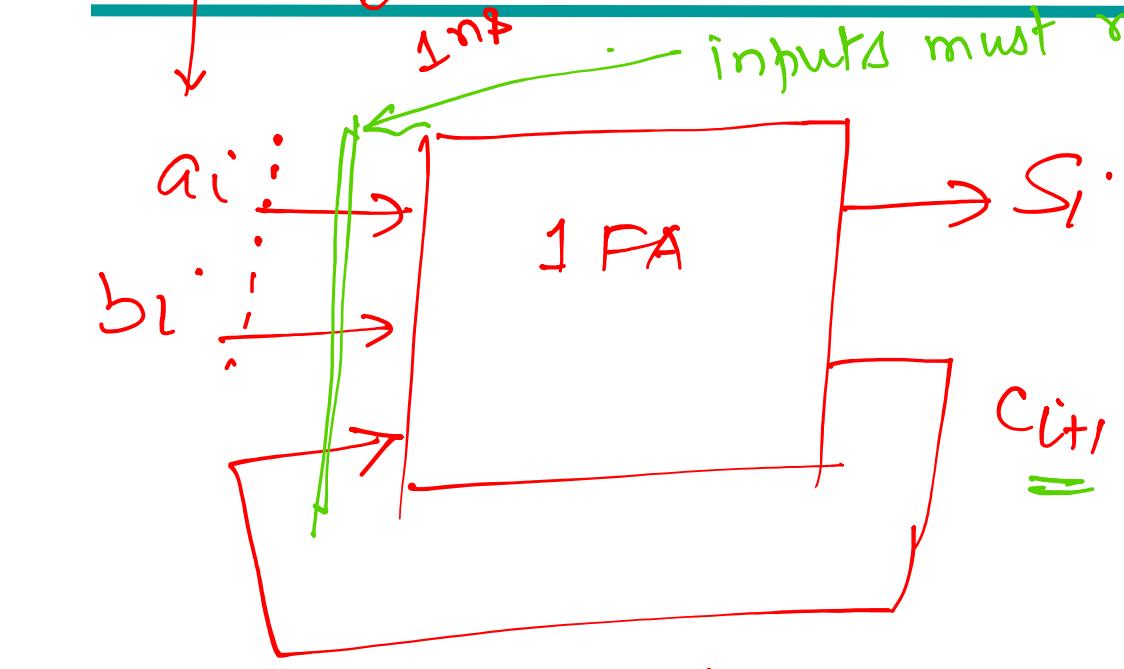
# Full-Adder Circuit

Serial addition



# Serial Adder

inputs must be supplied every  $1 \text{ ns}$



exactly feed it after  
 $1 \text{ ns}$

inputs must remain stable for  $1 \text{ ns}$

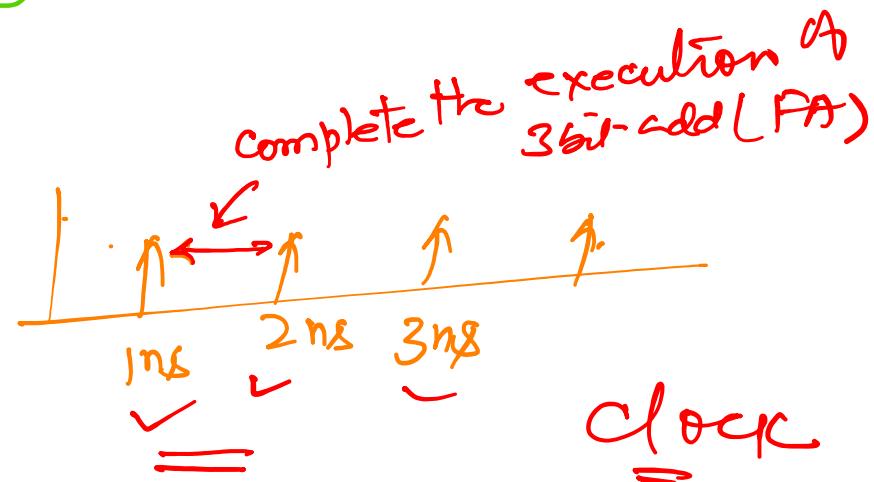
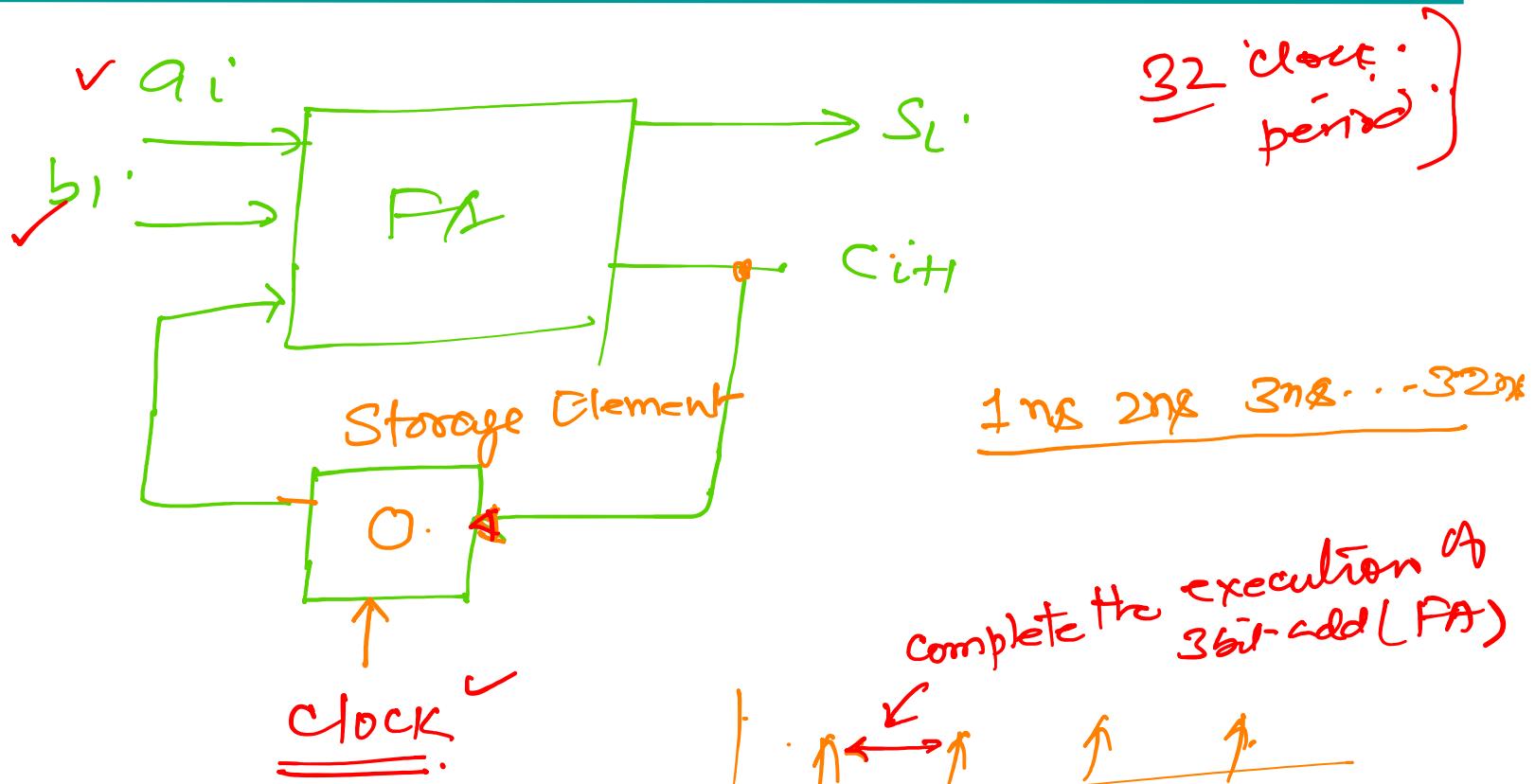
delay ✓

32 ns.

Cost  
1 FA  
Very cheap.



# Serial Adder



$$\text{freq} = \frac{1}{1 \times 11} \text{ GHz} = 10^9 \text{ Hz}$$

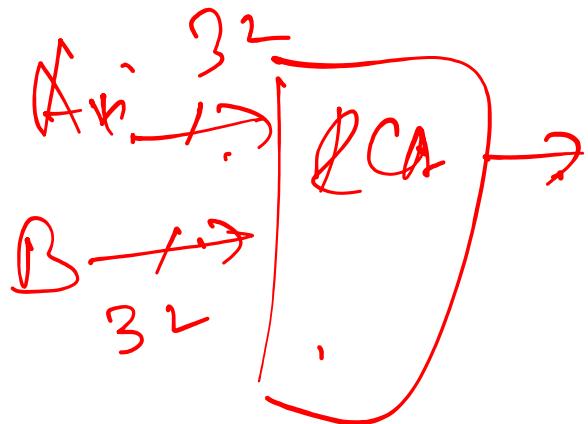


# Serial Adder

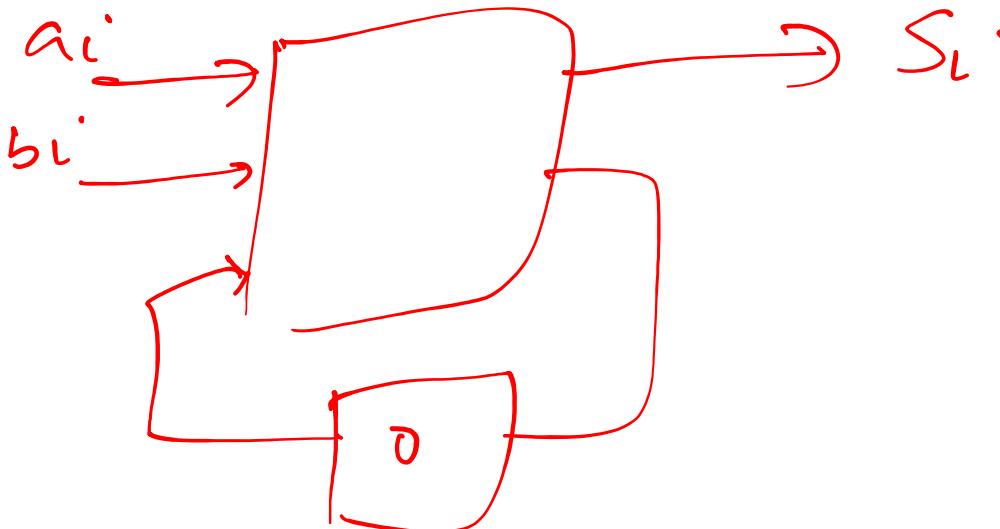
32 bit no → 32 ns) ?

16 bit → 16 ns

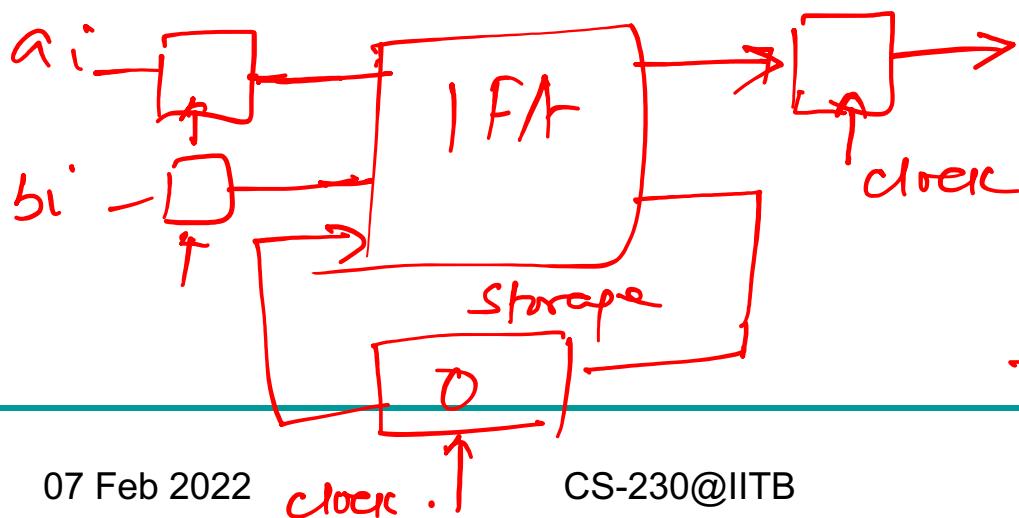
8 bit → 8 ns



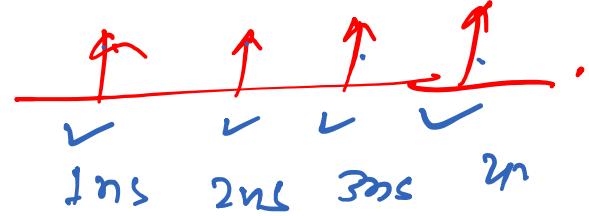
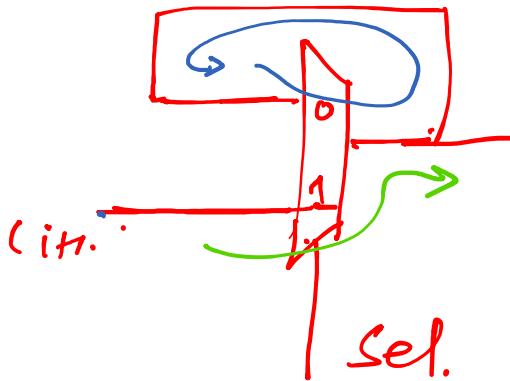
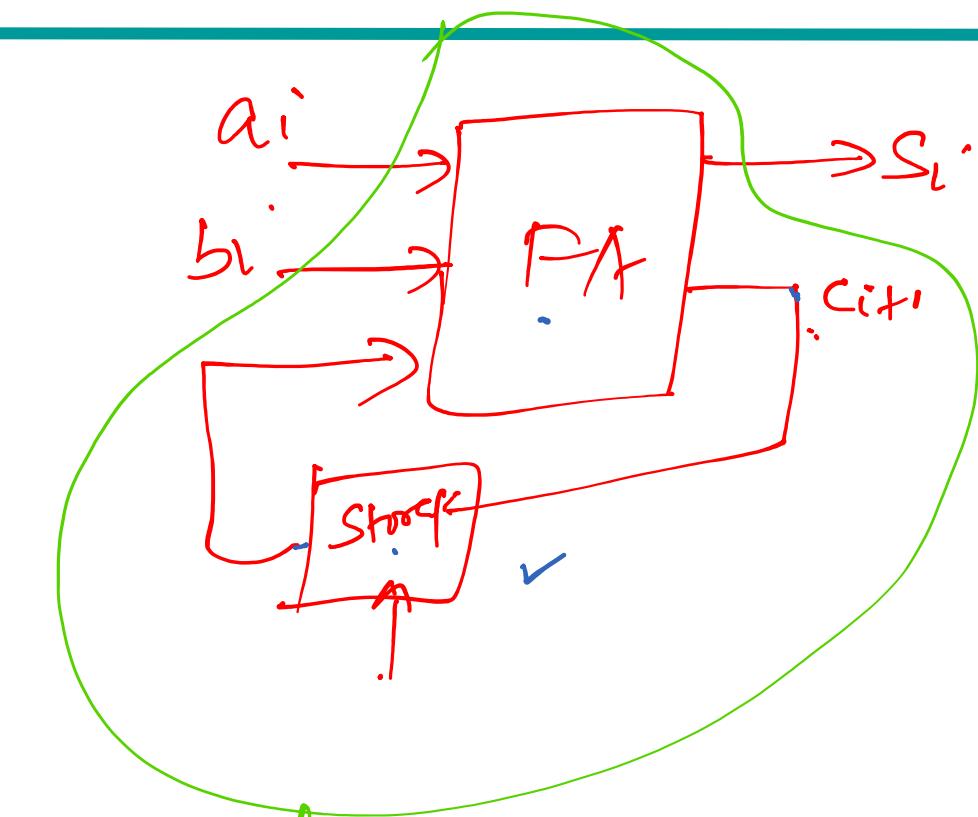
# Serial Adder



$\underbrace{1 \text{ FA}}_{\begin{array}{l} 1. \text{ Storage} \\ \text{element} \end{array}} = \checkmark$



# Serial Adder



# Serial Adder

<i>Input</i> $a_i \ b_i$	$c_i$	$s_i$ - Output
0 0	0	0
0 1	0	1
0 1	1	0
1 0	0	1
1 0	1	0
1 1	0	0
1 1	1	1

Output is not only dependent on the current inputs but also dependent on previous inputs

Sequential behaviour

Synchronous Sequential Circuits



# Serial Adder

Temporal Behaviour  $\rightarrow$  sequential logic

(MEMORY)

STATE OF THE SYSTEM

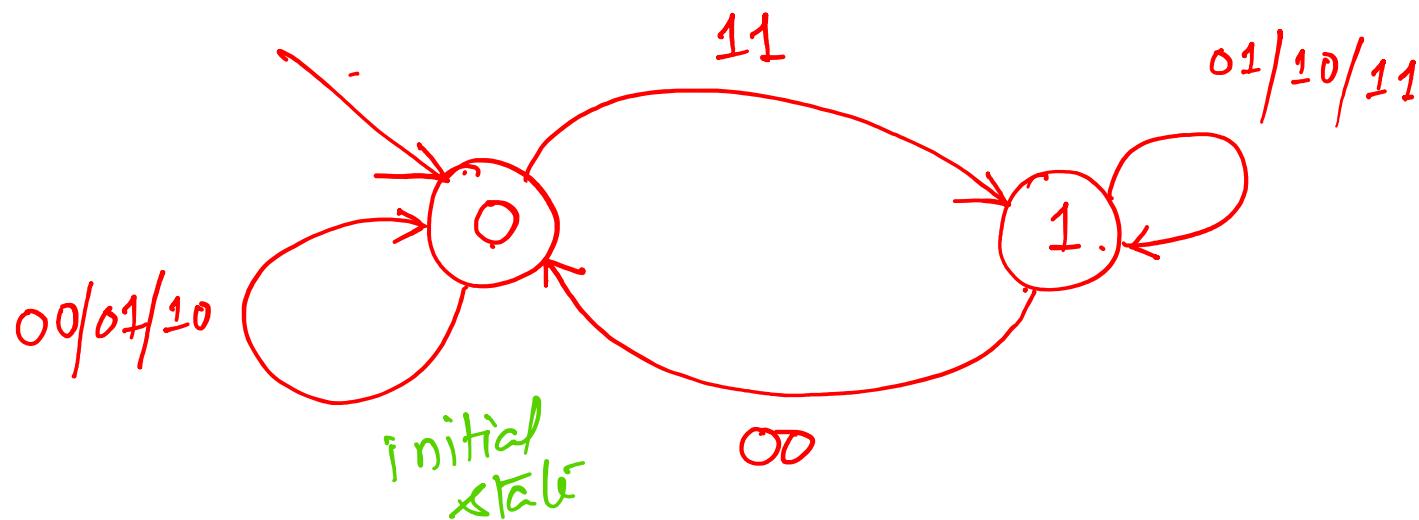
$c_3, c_2, c_1$   
 $a_3, -a_1, a_0$   
 $b_3, b_1, b_0$

Memorize the carry.

Carry = 0      ]  
Carry = 1      ]



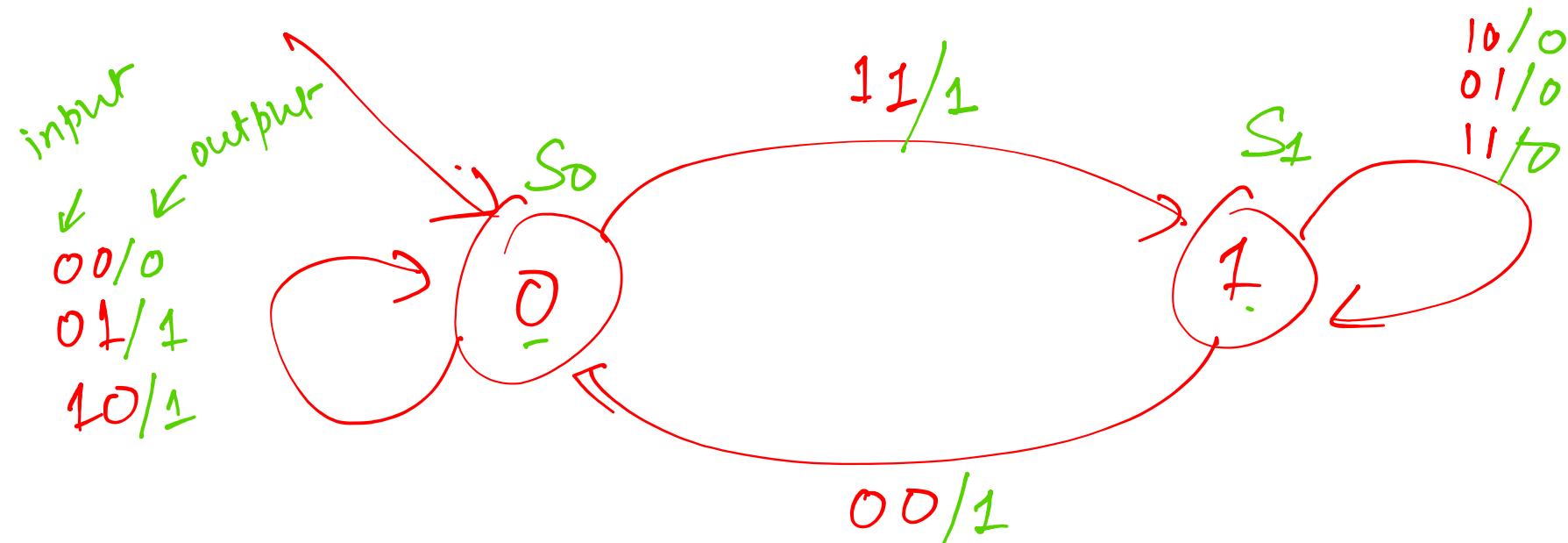
Adder



State Transition Graph. (STG)



# Serial Adder



# Thank You



# Sequential Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: [viren@cse, ee.iitb.ac.in](mailto:viren@cse, ee.iitb.ac.in)

*CS-230: Digital Logic Design & Computer Architecture*

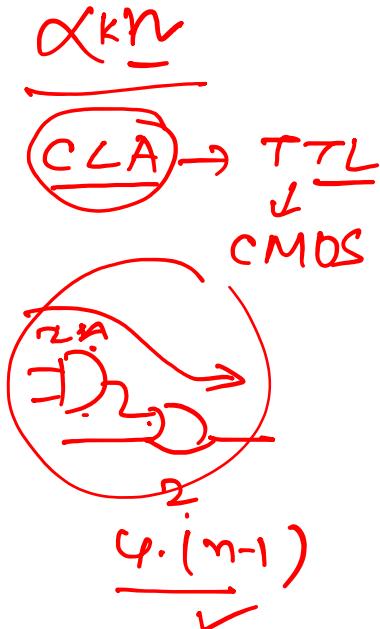
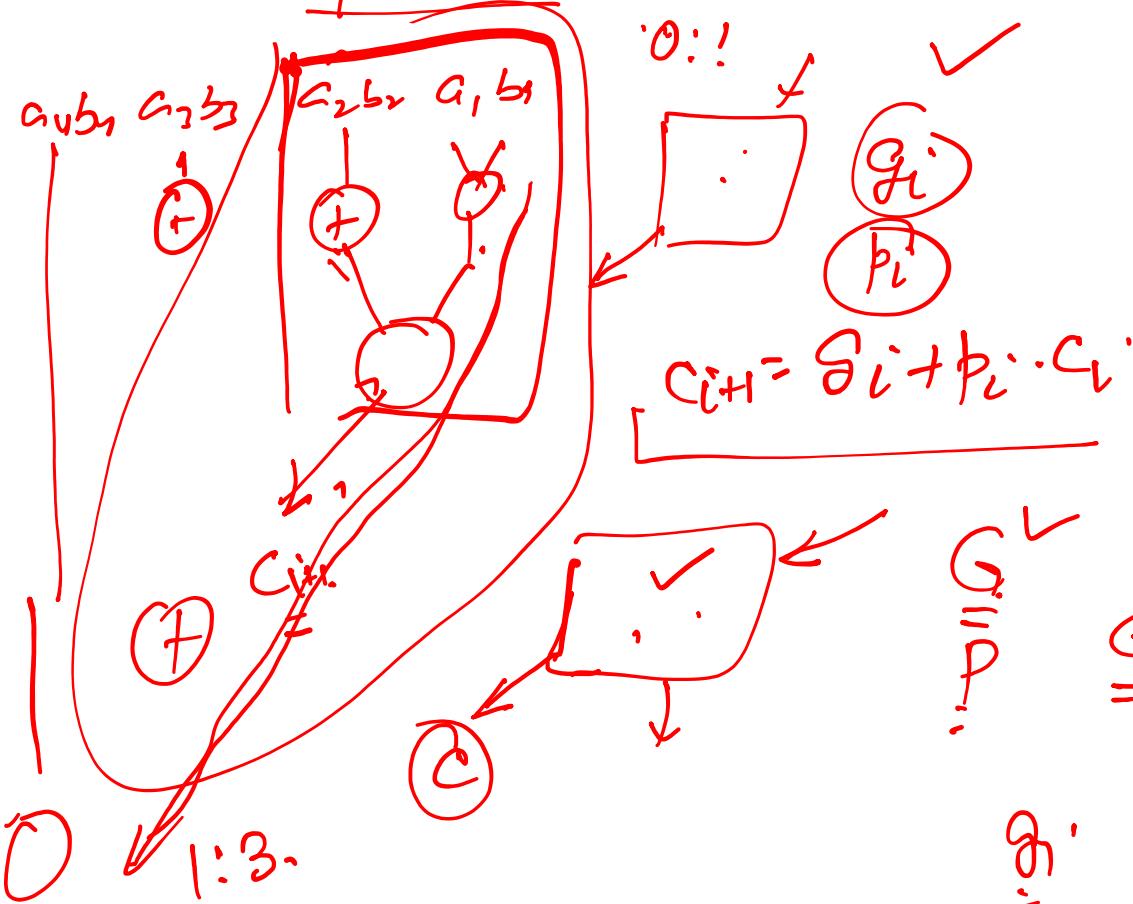
---

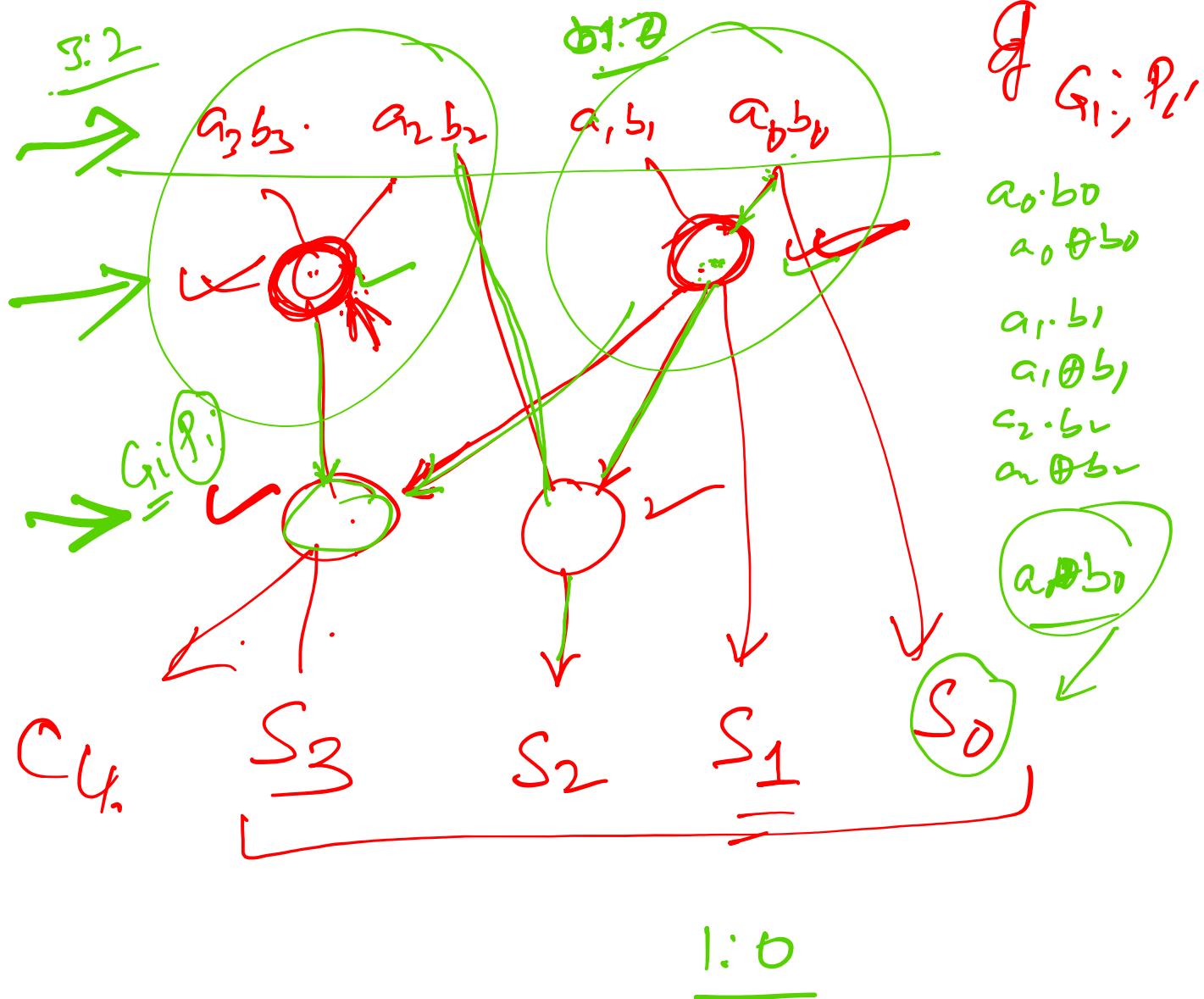
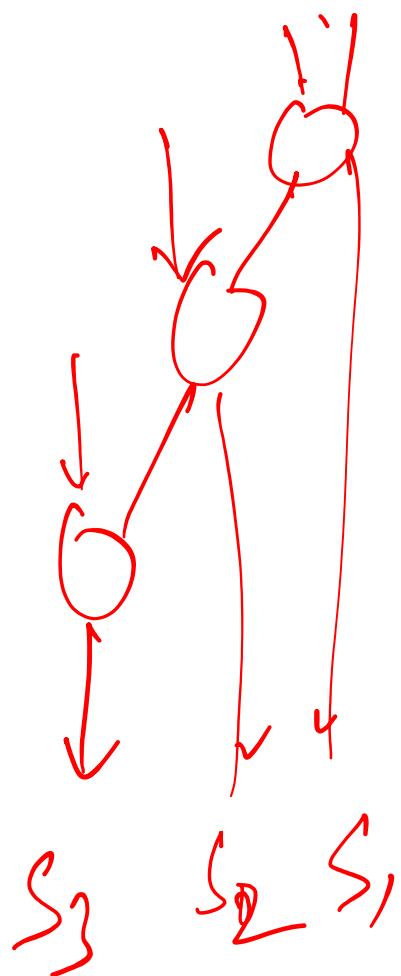


Lecture 15 (08 February 2022)

**CADSL**

Sequential





3:2

$$g_3 \quad q_{3 \cdot b_3}$$

$$g_3 \cdot q_2 \cdot b_2$$

$$q_3 \oplus b_3 = p_3$$

$$q_3 \oplus b_2 = p_2$$

$$\underline{q_{3:2}} = \underline{\underline{q_3 + q_2 \cdot b_3}}$$

generated  
from  
cells 3:2

$$\rightarrow P_{3:2} = \underline{P_3 \cdot b_2}$$

i

$$Q_{3:2} = \underline{q_3 + q_2 \cdot p_3}$$

$$B:2 = P_3 \cdot b_2$$



$$G_{3:2} = g_3 + g_2 \cdot p_3$$

$$P_{3:2} = p_3 \cdot p_2$$

$$a_1, b_1$$

$$a_0, b_0$$

$$g_1 = a_1 \cdot b_1$$

$$g_0 = a_0 \cdot b_0$$

$$p_1 = a_1 \oplus b_1$$

$$p_0 = a_0 \oplus b_0$$

↓

$$G_{1:0} = g_1 + g_0 \cdot p_1$$

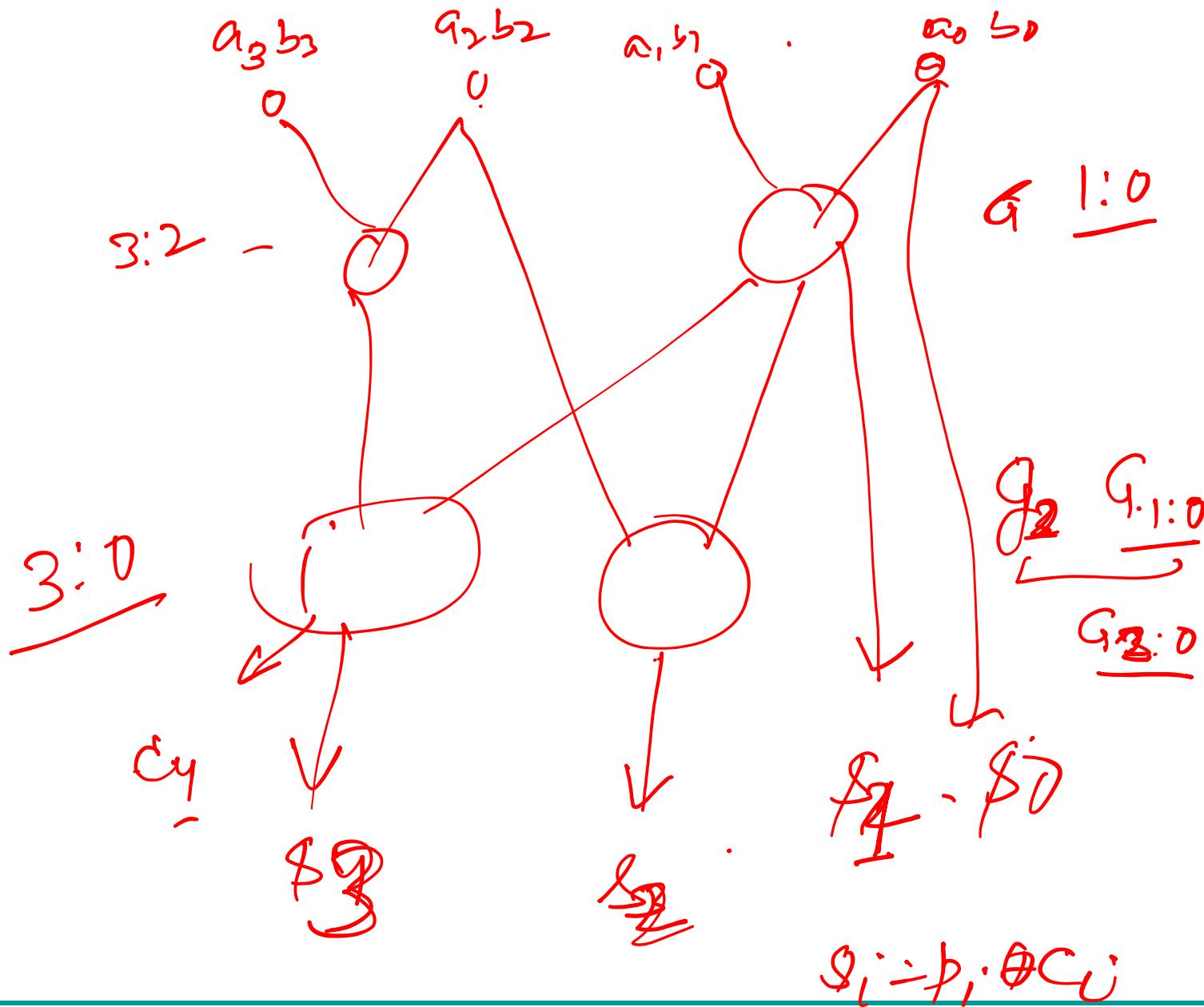
$$P_{1:0} = p_1 \cdot p_0$$

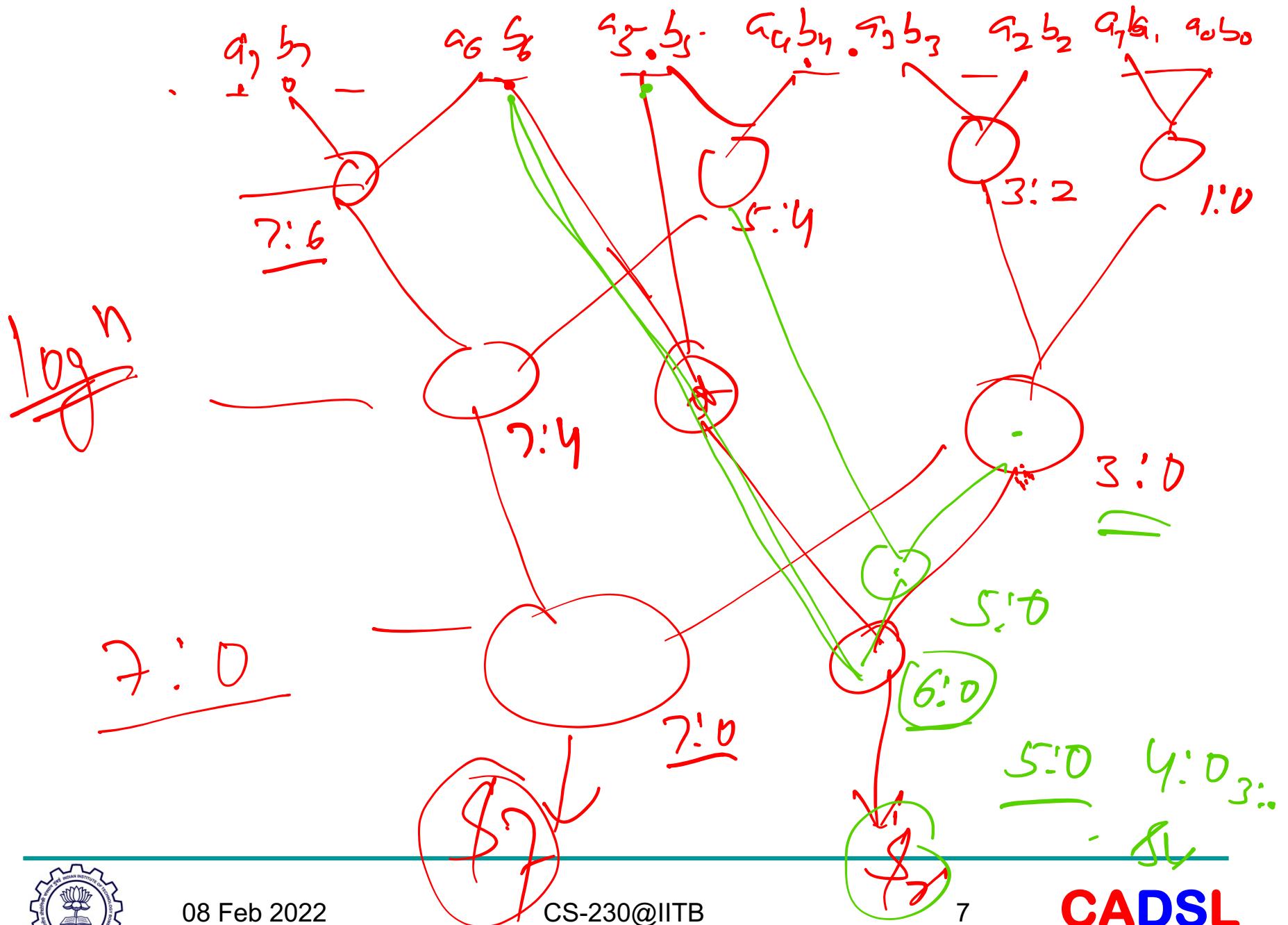
$$\boxed{G_{3:0} = G_{3:2} + P_{3:2} \cdot G_{1:0}}$$

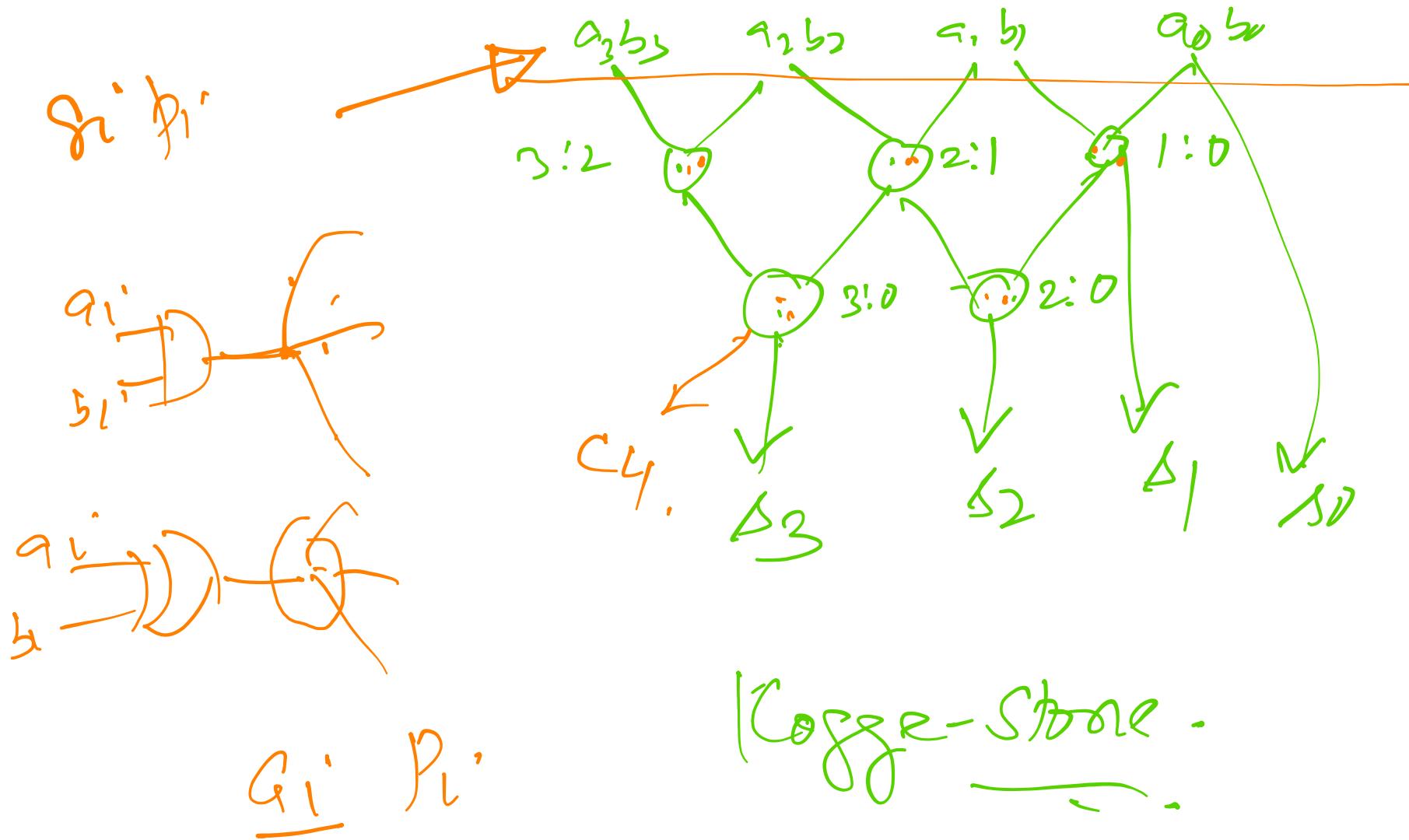
$$P_{3:0} = P_{3:2} \cdot P_{1:0}$$

3 - 0

$$P_n \underset{=}{\text{C}} \underset{\text{def}}{=} \frac{G_{3:0} + P_{3:0} \cdot C_{0:2}}{S_1 = p_1 \oplus S_0 = p_0}$$

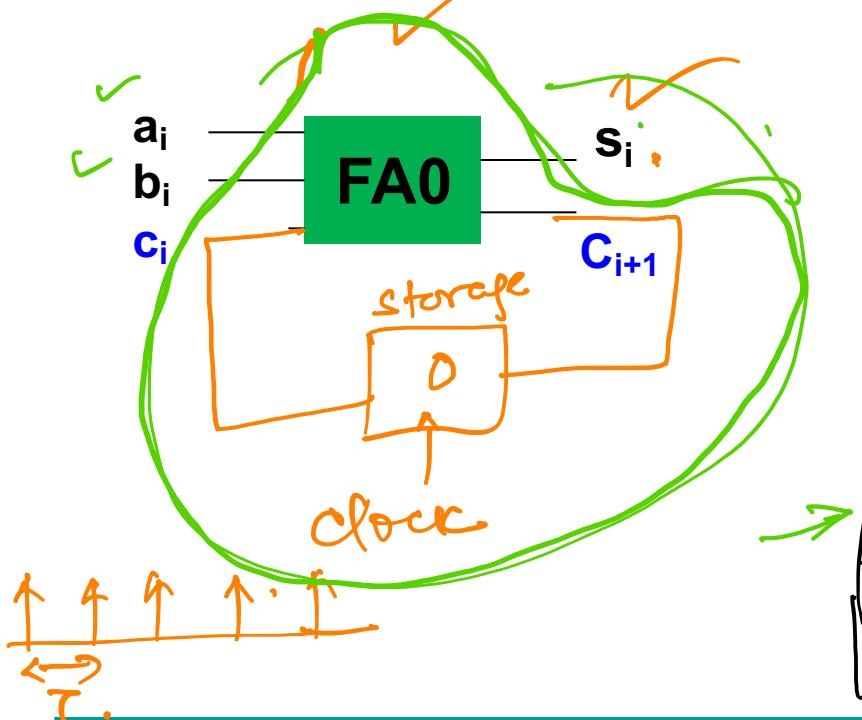
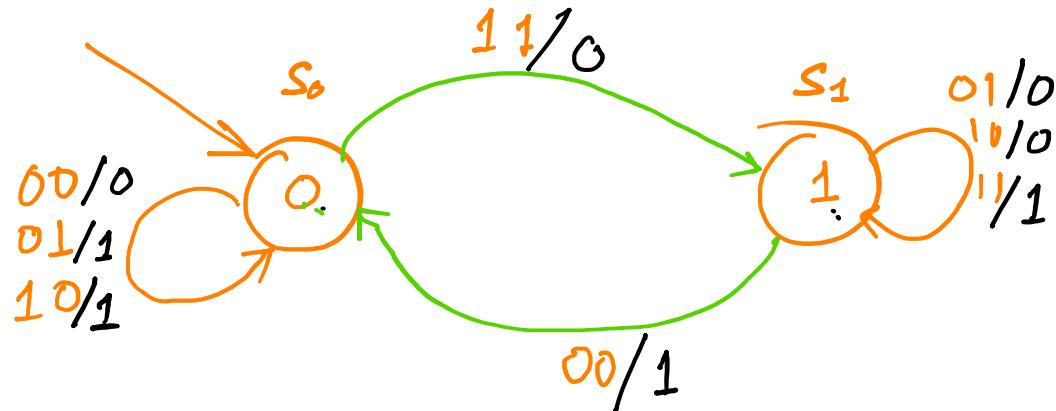






# Serial Adder

$$\begin{array}{r}
 c_{32} \ c_{31} \dots \ c_2 \ c_1 \ 0 \\
 a_{31} \dots \ a_2 \ a_1 \ a_0 \\
 + b_{31} \dots \ b_2 \ b_1 \ b_0 \\
 \hline
 s_{31} \dots \ s_2 \ s_1 \ s_0
 \end{array}$$



STG

State	00	01	10	11
0	0/0	0/1	0/1	1/0
1	0/1	1/0	1/0	1/1

SIT



# State Machine

$$M(I, O, S, S_0, \delta, \lambda)$$

I: Input symbols  
 $\{00, 01, 10, 11\}$

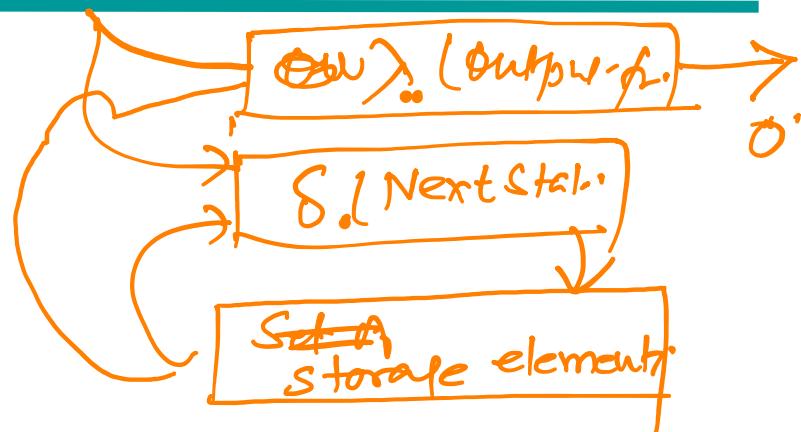
O: Output symbol  
 $\{0, 1\}$

S: Set of States  
 $\{0, 1\}$

$S_0$ : Initial state  $\{0\}$

$\delta$ :  $S \times I \rightarrow S$  Transition function

$\lambda$ :  $S \times I \rightarrow O$  Output function



$\delta \rightarrow$  Combinational logic

Storage:  $\log_2 |S|$



# Thank You



08 Feb 2022

CS-230@IITB

11

**CADSL**

# Sequential Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

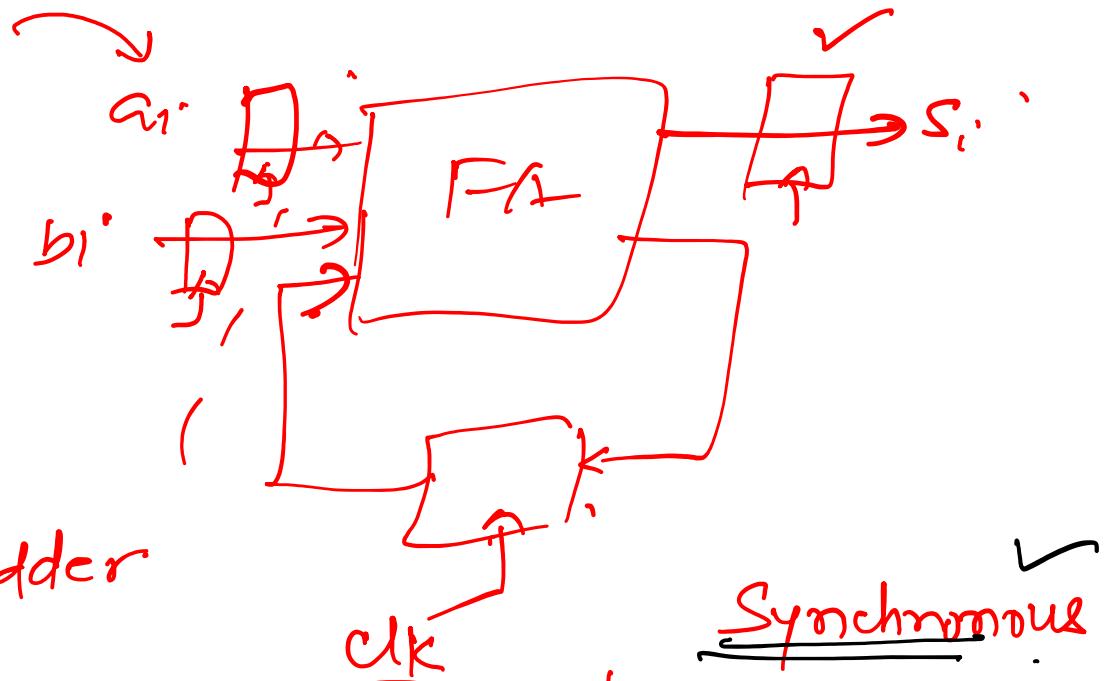
*CS-230: Digital Logic Design & Computer Architecture*

---



Lecture 16 (10 February 2022)

**CADSL**

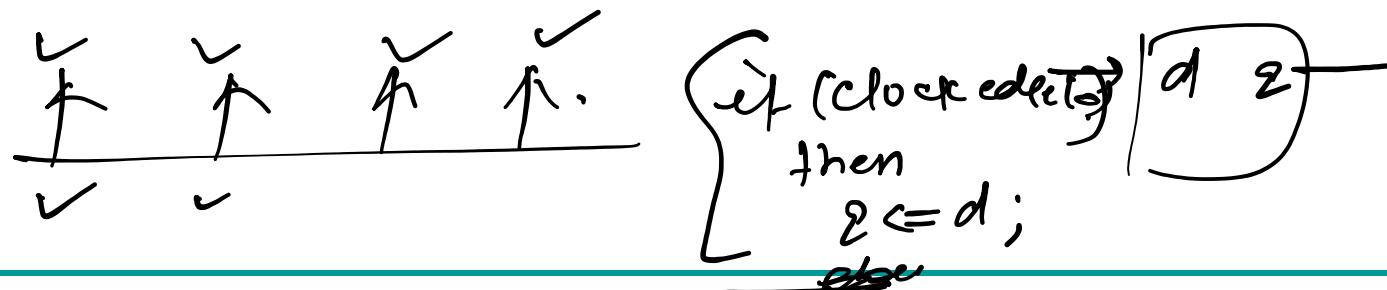


Parallel Adder

↓  
serial adder

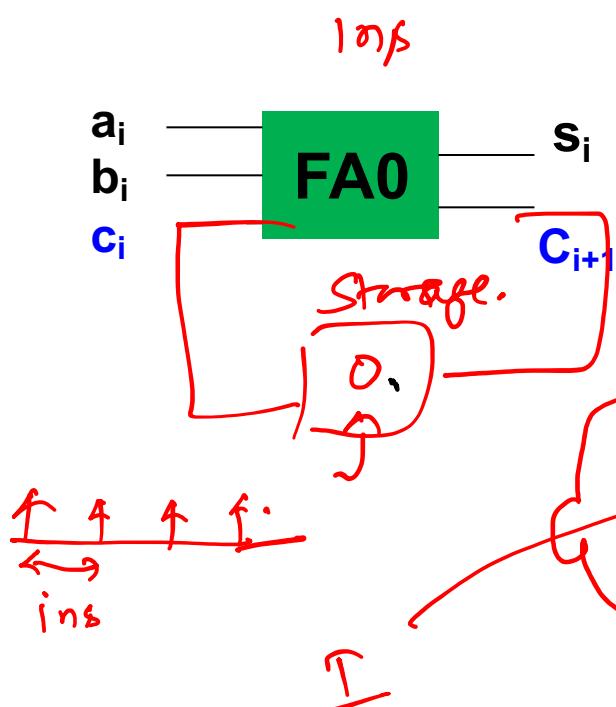
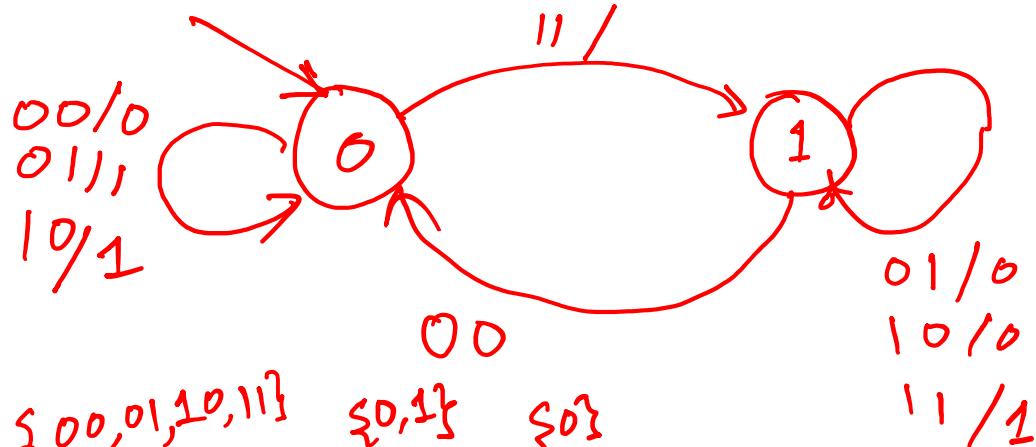
(Temporal)

Synchronous Sequential Circuits



# Serial Adder

$$\begin{array}{r}
 c_{32} \ c_{31} \dots \ c_2 \ c_1 \ 0 \\
 a_{31} \dots \ a_2 \ a_1 \ a_0 \\
 + b_{31} \dots \ b_2 \ b_1 \ b_0 \\
 \hline
 s_{31} \dots \ s_2 \ s_1 \ s_0
 \end{array}$$

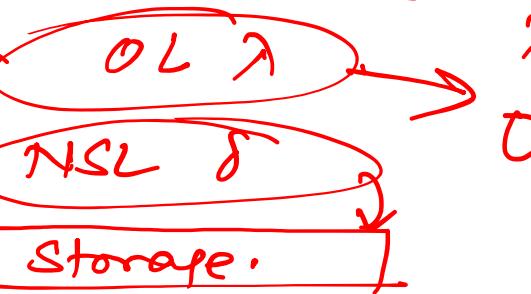


Mathematical representation of the serial adder state transition:

$$M(I, O, S, S_0, \delta, \lambda)$$

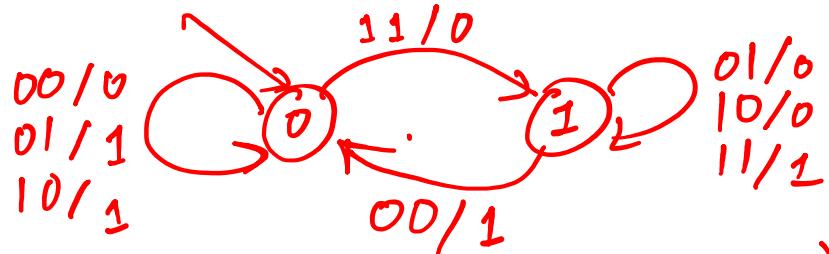
Annotations:

- $I = \{00, 01, 10, 11\}$
- $O = \{0, 1\}$
- $S = \{0\}$
- $S_0 = \{0, 1\}$
- $\delta: S \times I \rightarrow S$
- $\lambda: S \times I \rightarrow O$



$$\log_2 |S|$$

# State Machine



$$\delta = S \times I \rightarrow S$$

$\Delta$

		00	01	11	10
		0	0	1	0
		1	1	1	1
$\delta$					
0	0	0	1	0	0
1	1	1	1	1	0

$$= a \cdot b + \bar{a} \cdot b + \bar{a} \cdot a$$

$$I = \{00, 01, 10, 11\},$$

a, b

$$\delta(\circledast, \delta, a, b)$$

$\delta$	$a, b$	$00$	$01$	$11$	$10$
0	0	0	1	0	1
1	1	1	0	1	0

$$\lambda = \delta \cdot \bar{a} \bar{b} + \bar{\delta} \bar{a} b + \delta a b + \bar{\delta} a \bar{b}$$

$$= \delta \cdot (\bar{a} \bar{b} + a \cdot b) + \bar{\delta} \cdot (\bar{a} b + a \bar{b})$$

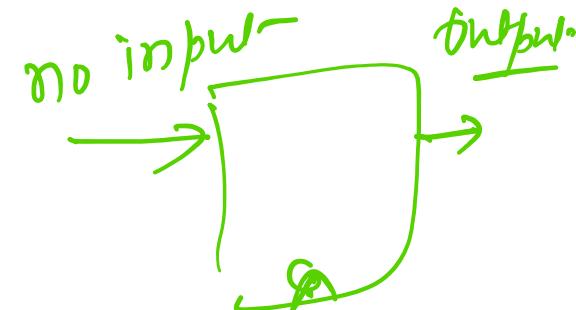
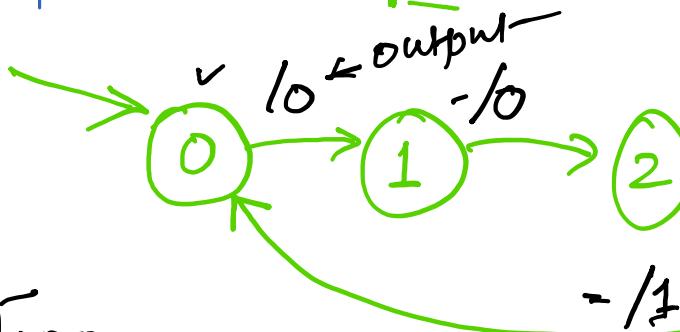
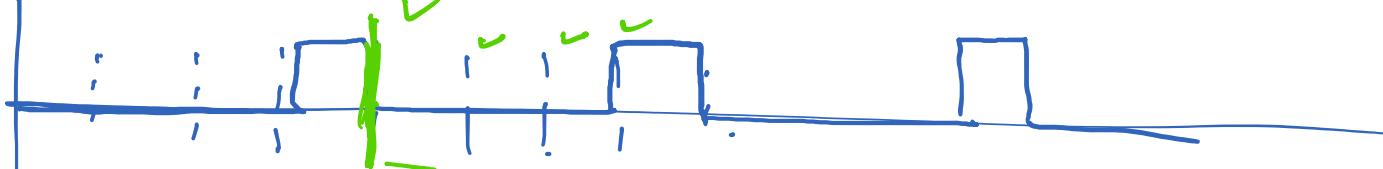
$$= \delta \cdot (\bar{a} \oplus b) + \bar{\delta} \cdot (a \oplus b)$$

# State Machine

output

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3.

0, 1, 2, 3



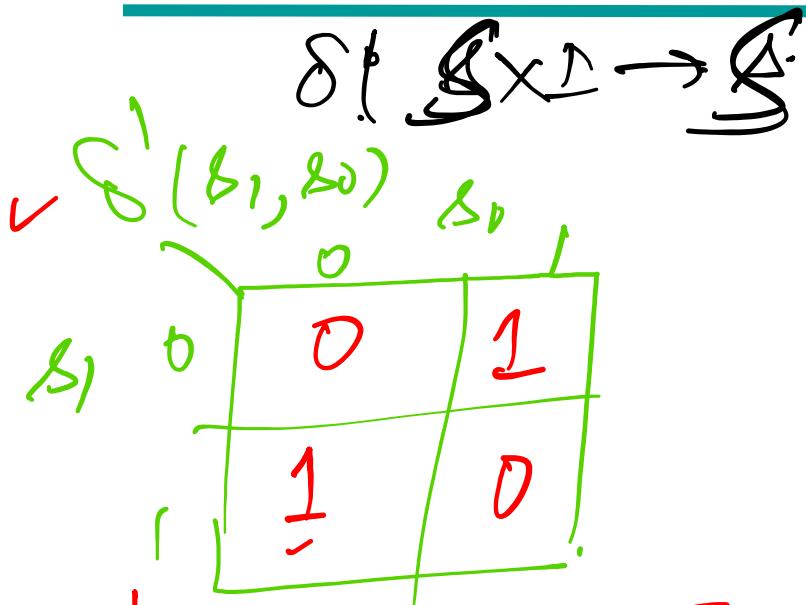
State  
transition  
diagram/ graph.

## 2 Storage element

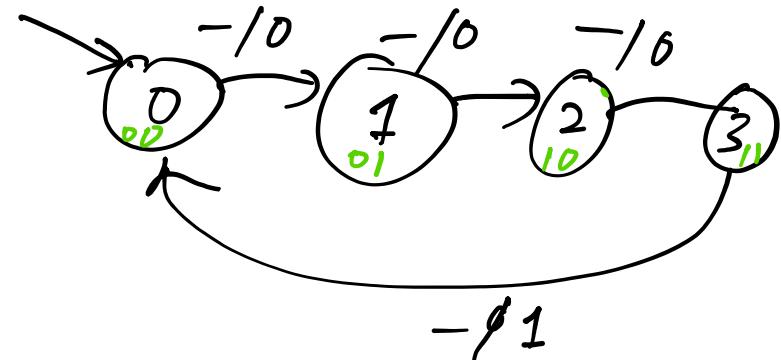
$$\log_2 4 = 2$$



# Finite State Machine

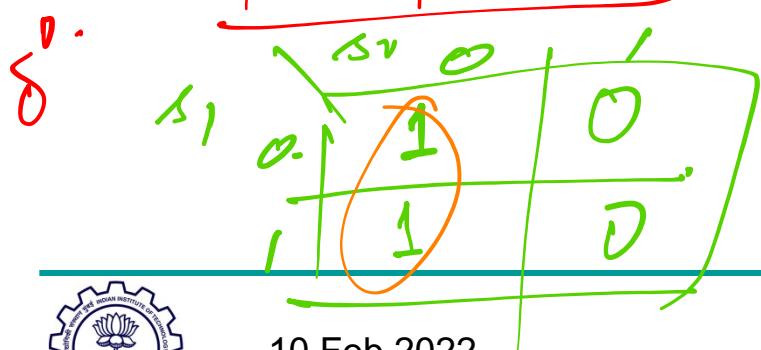


$\delta(\delta_1, \delta_0)$



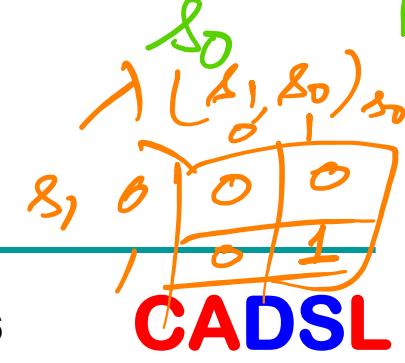
$$\delta^1(\delta_1, \delta_0) = \delta_1 \delta_0 + \bar{\delta}_1 \delta_0$$

$$[\delta^1 = \delta_1 \oplus \delta_0]$$



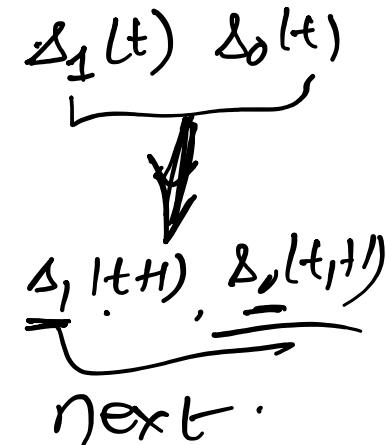
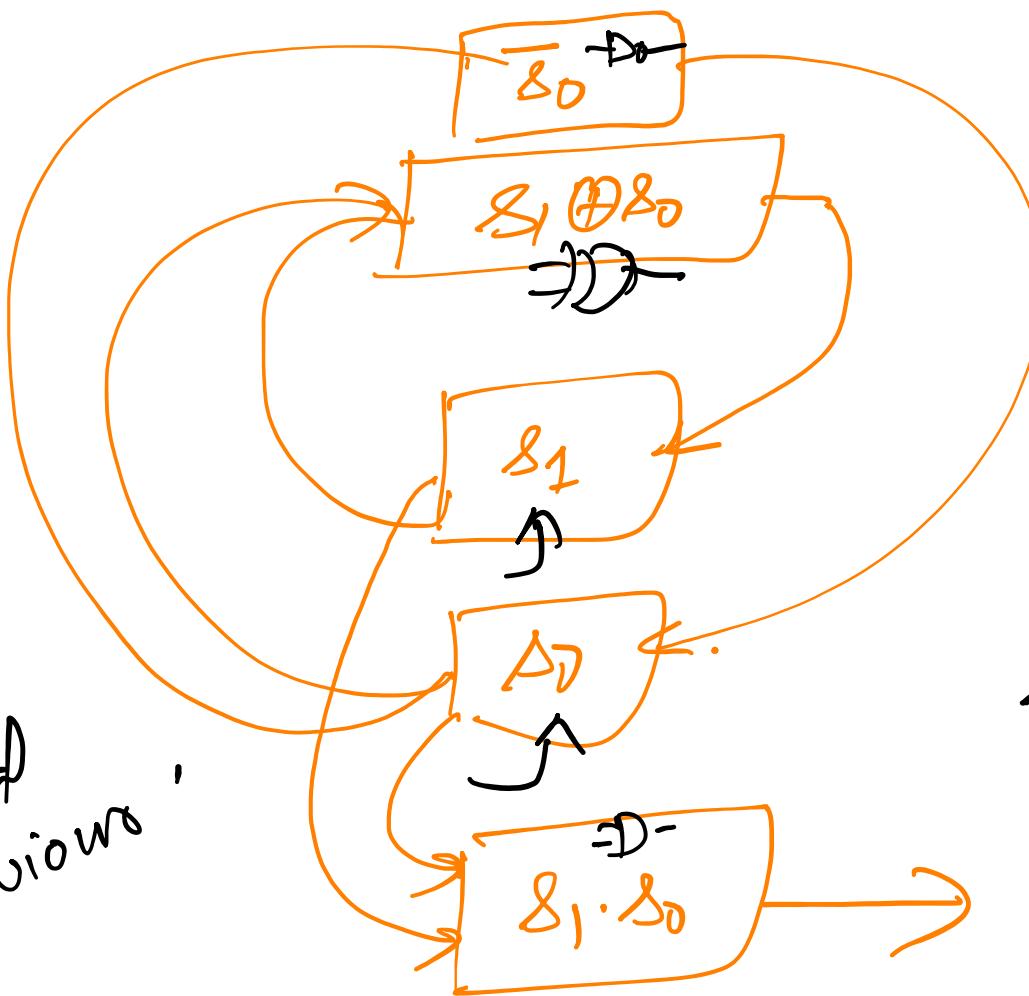
$\delta^1(\delta_1, \delta_0) \leftarrow$  transition A  
 $\delta^0(\delta_1, \delta_0) \rightarrow$  transition B

$$\delta^0(\delta_1, \delta_0) = \bar{\delta}_0$$



# Finite State Machine

temporal  
behaviours



$$\underline{S_1(t+1)} = \underline{S_1(t)} \oplus \underline{S_0(t)}$$
$$\underline{S_0(t+1)} = \overline{\underline{S_0(t)}}$$

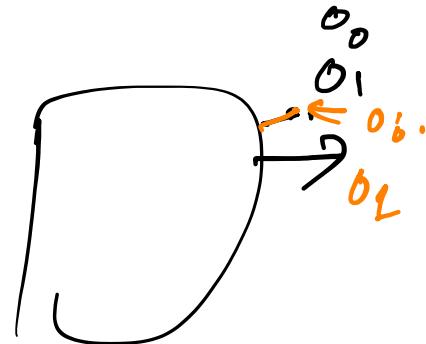
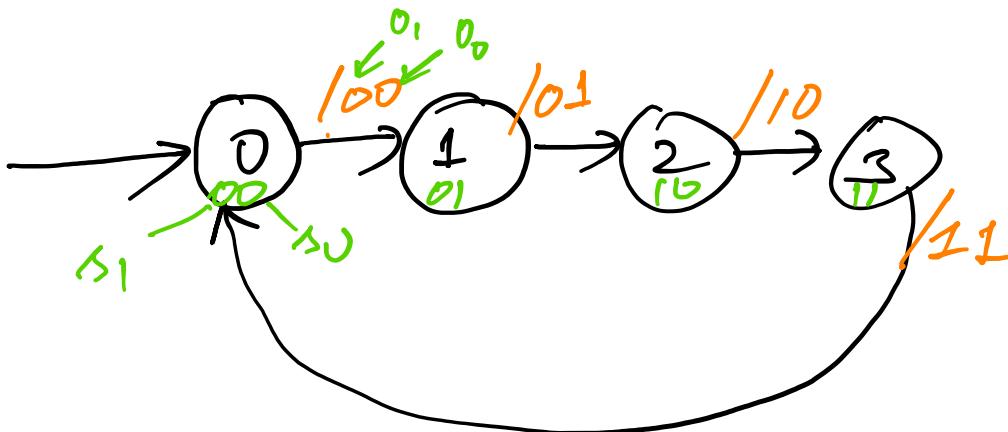
Output

$$\underline{\underline{O(t)}} = \underline{S_1(t)} \cdot \underline{S_0(t)}$$

CADSL



# Finite State Machine

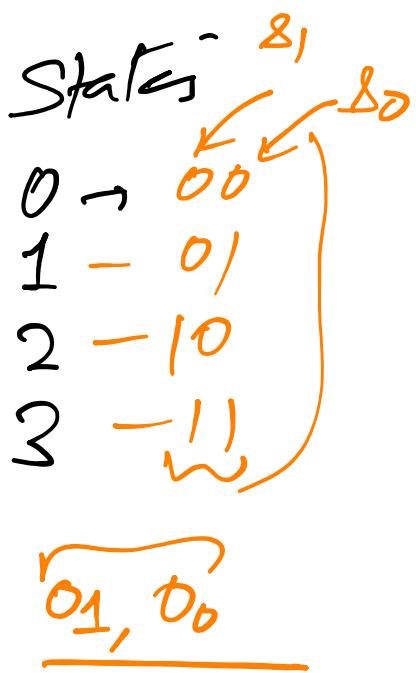


Sequence

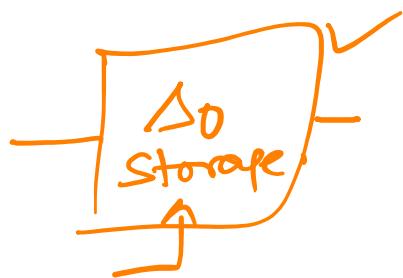
Sequence

Output → 0 1 2 3 0 1 2 3 |  
              ↓                          |  
Output → 00 01 10 11 00 01 10 11  
              ~    ~    ~    ~    ~    ~    ~

## output



# Finite State Machine



$\delta_1$	$\delta_0$
0	0
0	1
1	0
1	1

$$\begin{cases} \delta_1 \\ \delta_0 \end{cases} (\delta_1, \delta_0)$$

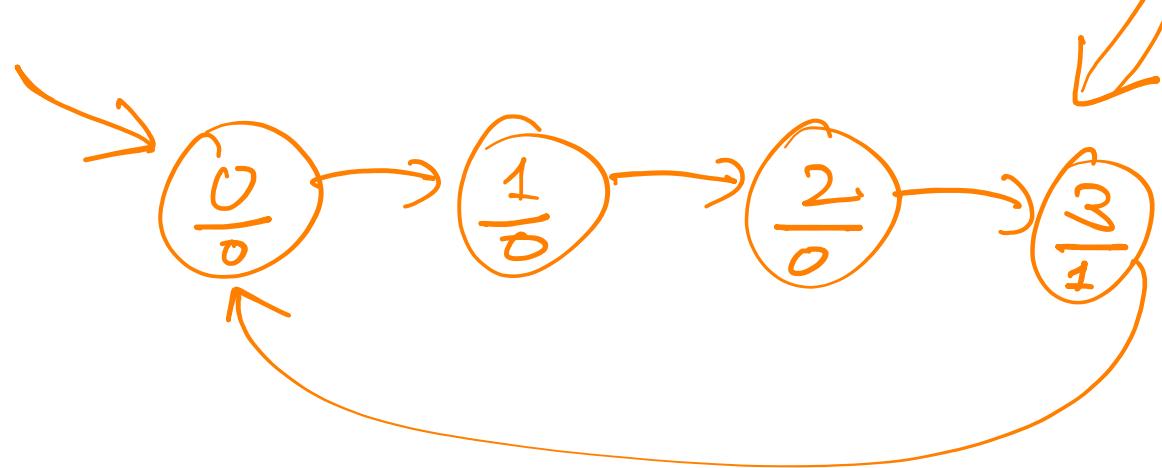
$$\begin{cases} \delta_1 \\ \delta_0 \end{cases} (\delta_1, \delta_0)$$



# Finite State machine

$$M(I, O, S, S_0, \delta, \lambda)$$
$$\delta: S \times I \rightarrow S$$
$$\lambda: S \times I \rightarrow O \leftarrow \text{Mealy machine}$$
$$\lambda: S \rightarrow O \leftarrow \text{Moore Machine.}$$

.

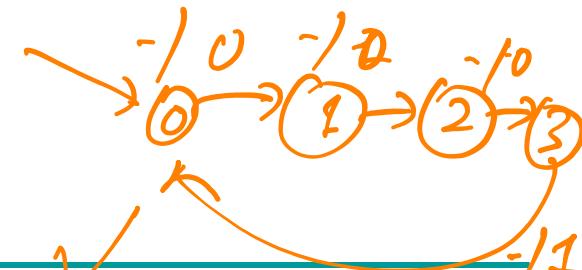


output is labeled  
on edge.



Mealy machine

↑ Output can be  
labeled at  
the state



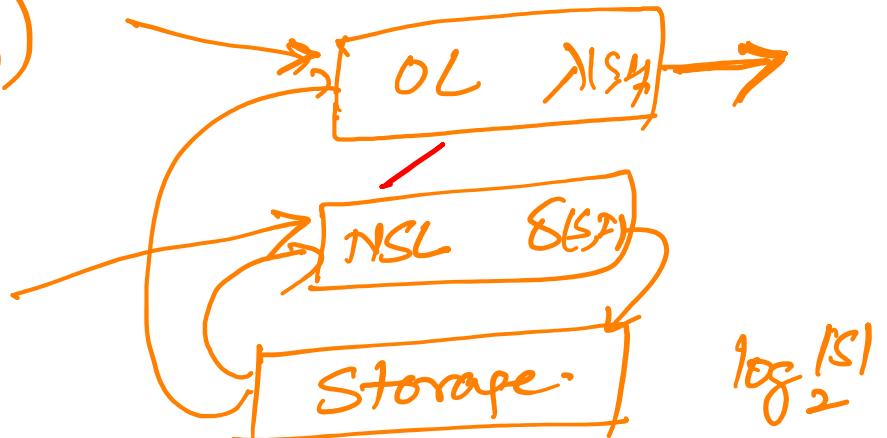
Mealy, moore CADSL



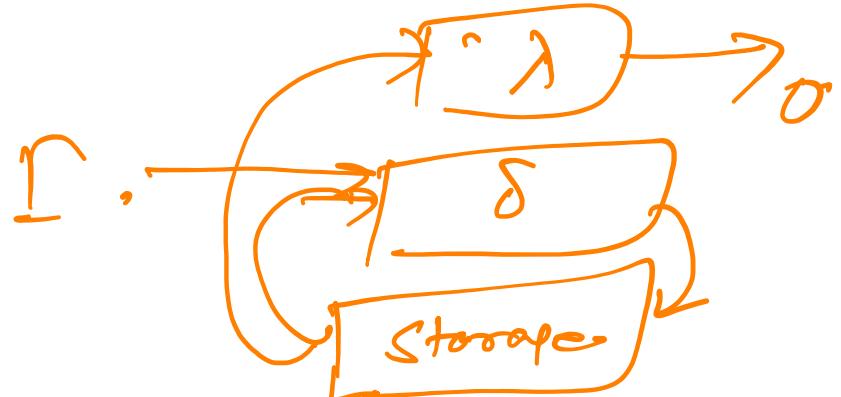
# Finite State machine

$$m(I, O, S, S_0, \delta, \lambda)$$

- How many storage elements
- encode the states
- Compute  $\delta$
- Compute  $\lambda$



Mealy m/c.



MOORE MACHINE'



# Thank You



# Sequential Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

*CS-230: Digital Logic Design & Computer Architecture*

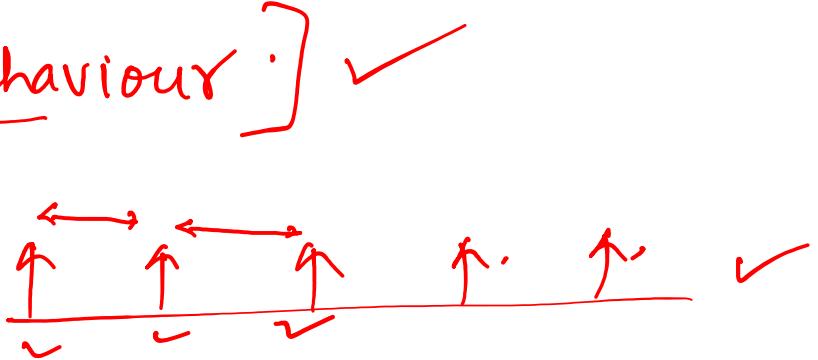
---



Lecture 17 (14 February 2022)

**CADSL**

## Sequential Circuit

- temporal behaviour ✓
- ↑  
clock.      

Synchronous Sequential Circuit

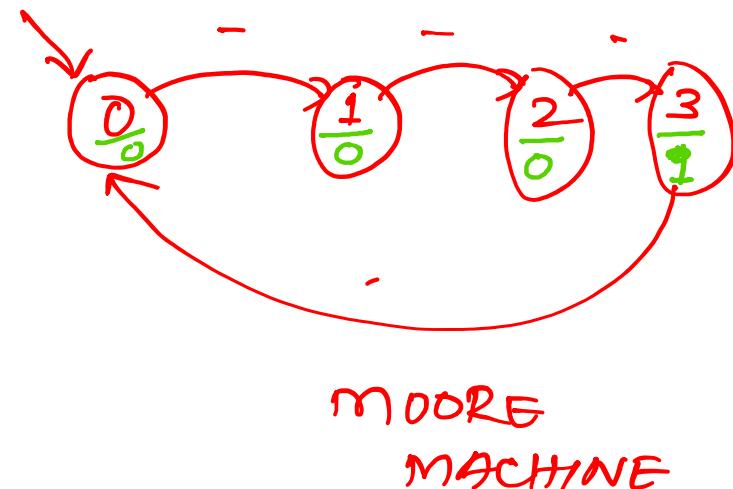
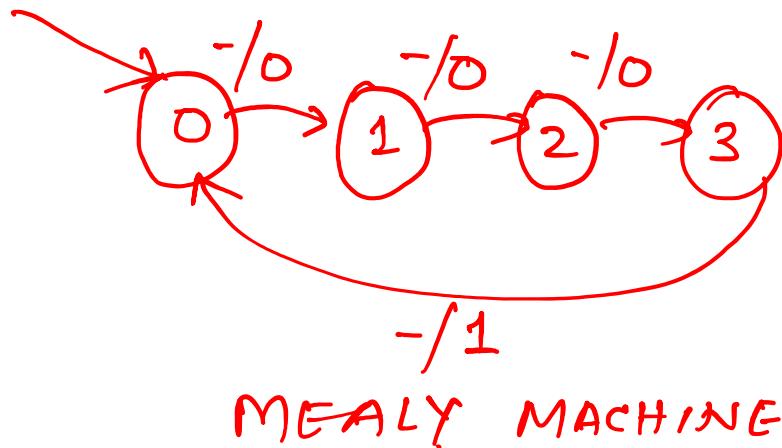
⇒ Asynchronous Sequential Circuits:  
( no clock ).  
Order · the event · ( handshaking )



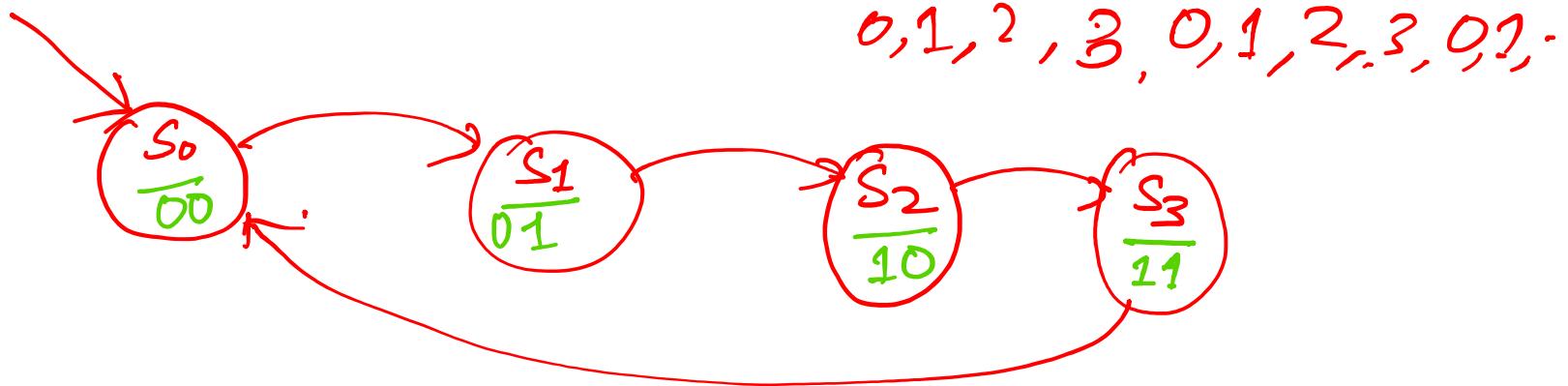
# State Machine

$$M \underbrace{(I, O, S, S_0, \delta, \lambda)}$$
$$\lambda: S \rightarrow O \quad \text{MEALY}$$
$$\lambda: S \times I \rightarrow O \quad \text{MOORE}$$
$$\text{MEALY}$$

0,0,0,1,0,0,0,1 .  

# State Machine



free running counter

$0, 1, 2, 3, 0, 1, 2, 3, \dots$

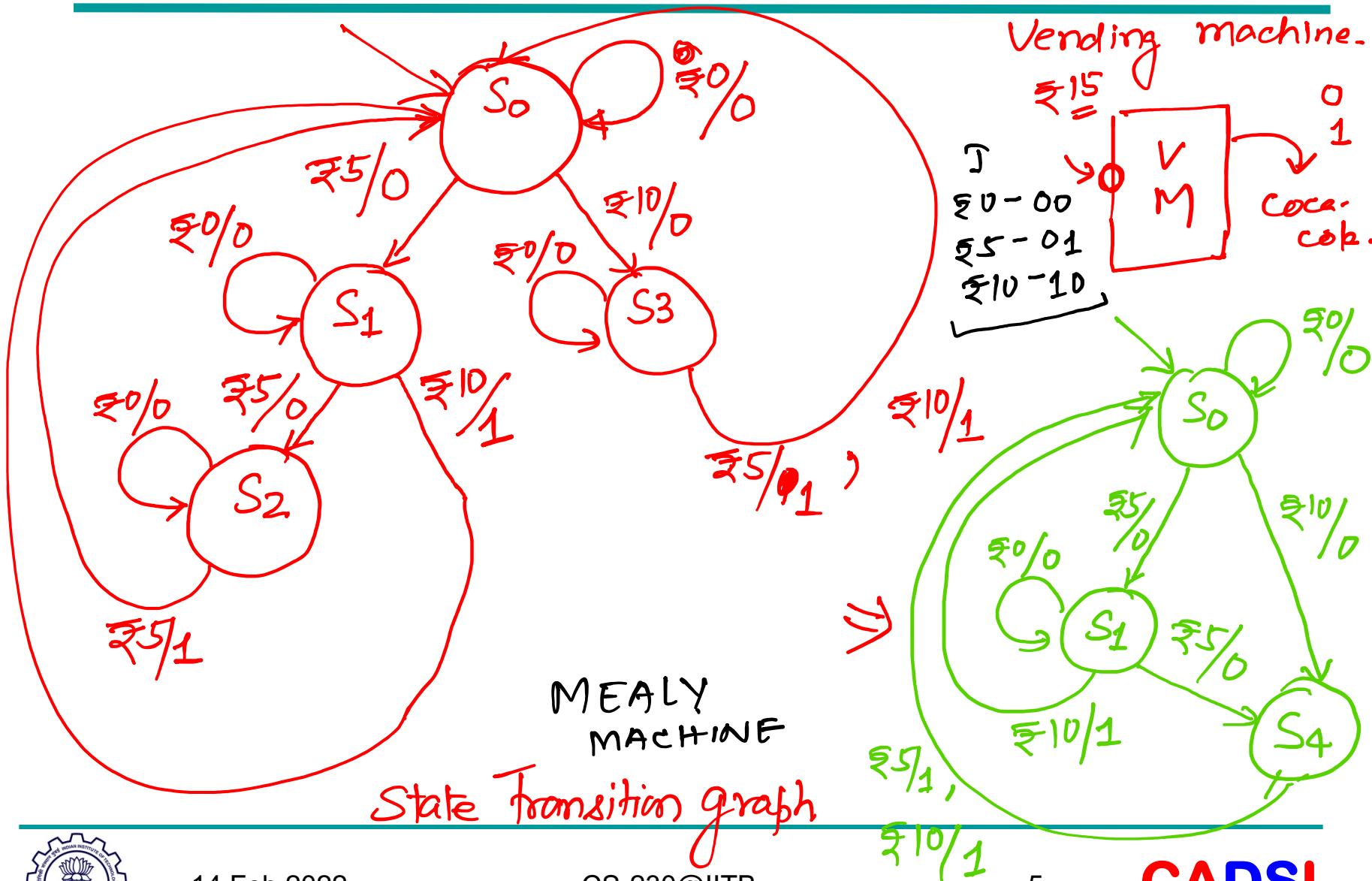
$$O = \{0, 1, 2, 3\}$$

```

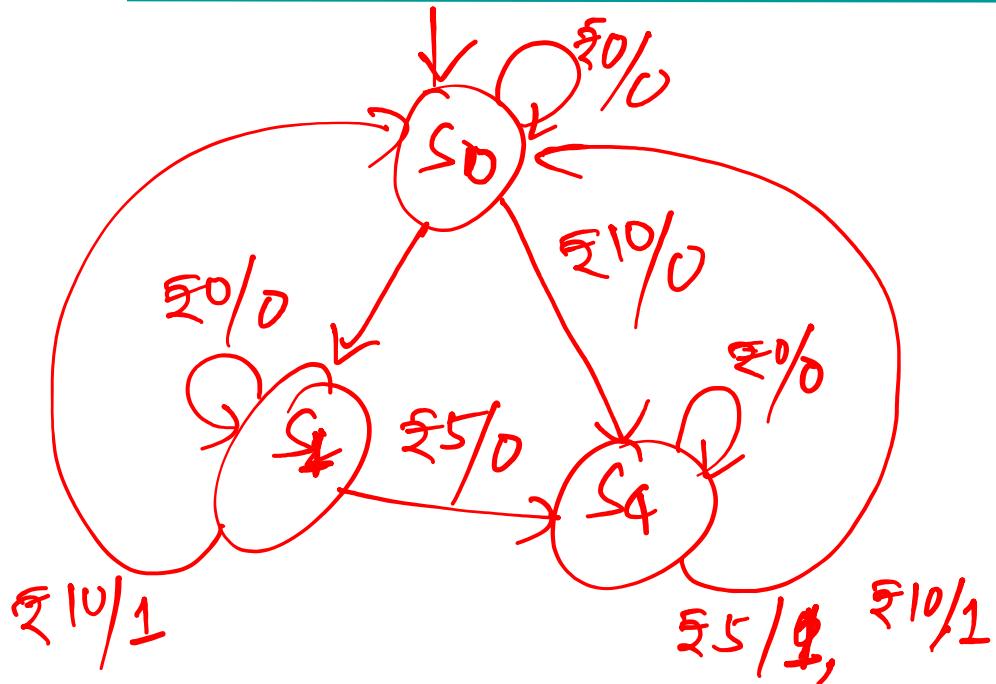
graph TD
    O["O = {0, 1, 2, 3}"]
    O --- 00["00"]
    O --- 01["01"]
    O --- 10["10"]
    O --- 11["11"]
  
```



# Finite State Machine



# Finite State Machine



3 States,  
 $t_1 \underline{t_0}$

$S_0 -$	$\underline{00}$
$S_1 -$	$01$
$S_4 -$	$10$

$I$	$a_1$	$a_0$
$\underline{00}$	00	
$\underline{01}$	01	
$\underline{10}$		10

PS	NS, Output		
	00/00	01 (2)	10 (2)
$S_0$ 00	$S_0, 0$	$S_1, 0$	$S_4, 0$
$S_1$ 01	$S_1, 0$	$S_4, 0$	$S_0, 1$
$S_4$ 10	$S_4, 0$	$S_0, 1$	$S_0, 1$

$$M(I, O, S, S_0, \delta, \lambda)$$

$$I = \{00, 01, 10\} \quad O = \{0, 1\}$$

$$S = \{S_0, S_1, S_4\} \quad S_0 = \{S_0\}$$

$$\delta' \quad \delta, \lambda$$



# Finite State Machine

$$S^{t_1}(t_1, t_0, a_1, q_0)$$

$$S^o(t_1, t_0, a_1, q_0)$$

$$\lambda(t_1, t_0, a_1, q_0)$$

Diagram illustrating a state transition function  $\lambda$  for a Finite State Machine. The input  $t_1, t_0$  and output  $a_1, a_0$  are shown at the top. The state transition matrix is as follows:

$t_1, t_0$	$a_1, a_0$	00	01	11	10
$t_1, t_0$	00	0	0	X	0
01	00	0	0	X	1
11	00	X	X	X	X
10	01	0	1	X	1

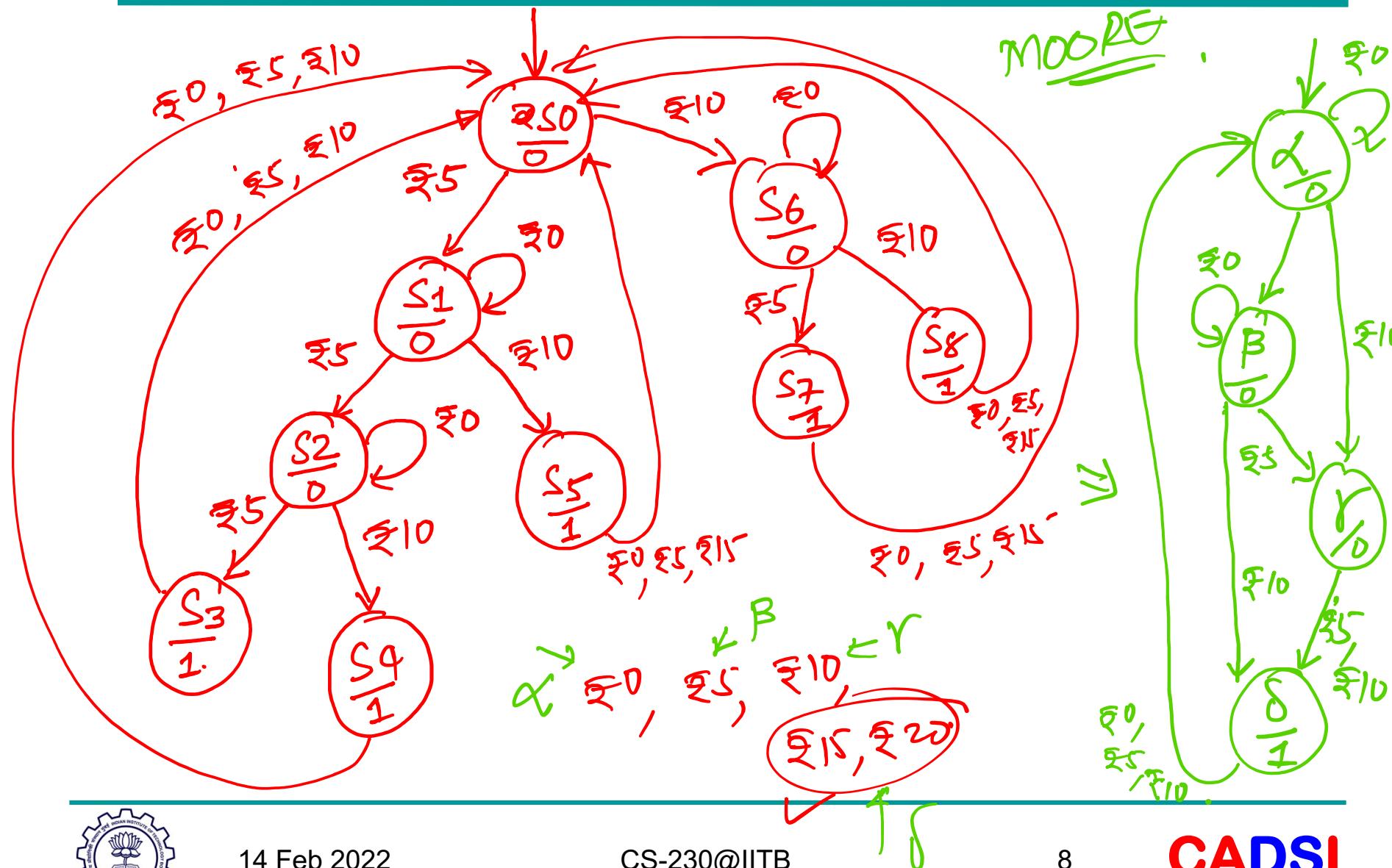
Three states (00, 01, 11) are highlighted with black ovals. Transitions from state 00 to 01, 01 to 00, and 11 to 01 are marked with 'X'. Transitions from state 00 to 11, 01 to 11, and 11 to 11 are marked with '1'.

$$\lambda = t_0 q_1 + t_1 q_0 + t_0 q_0$$

$$\begin{aligned} S^{t_1} &= \\ S^{t_0} &= \end{aligned}$$



# Finite State Machine



# Finite State machine

MOORE MACHINE

$$\xrightarrow{\quad \alpha, \beta, f, \delta \quad}$$
$$\stackrel{00}{\leftarrow} = \stackrel{01}{-} \stackrel{10}{-} \stackrel{11}{-}$$

$t_1, t_0$

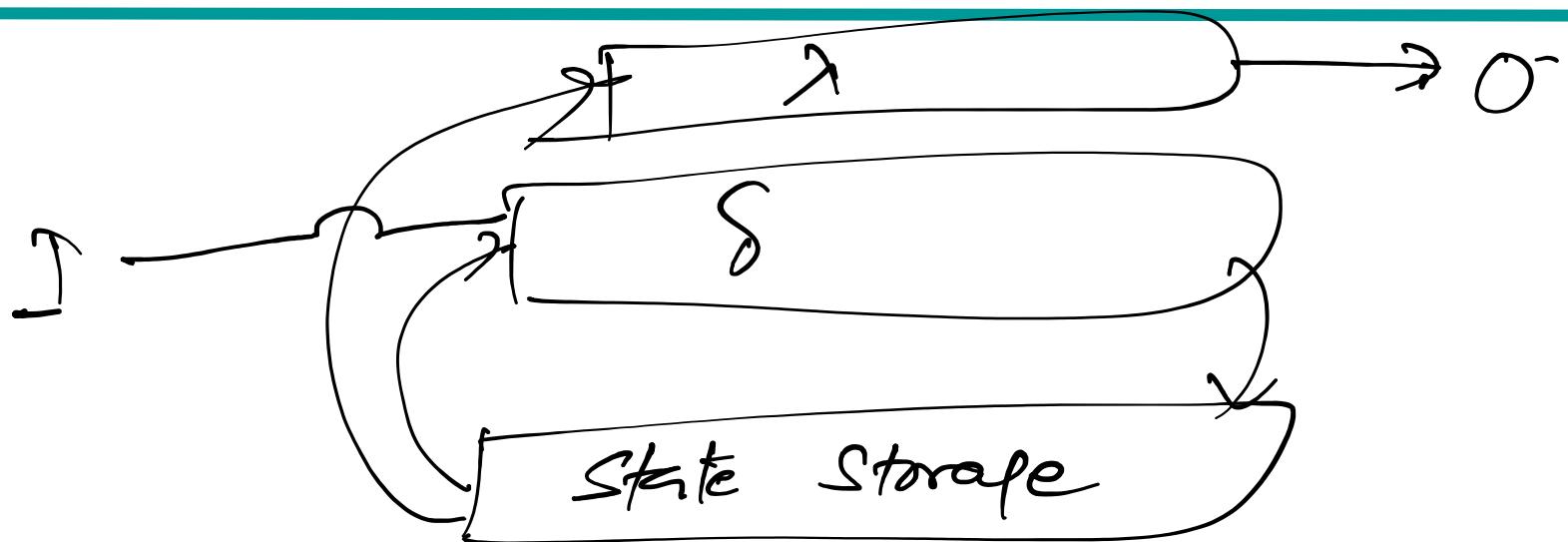
$$\left. \begin{array}{l} S^t(t_1, t_0, q_1, q_0) \\ \delta^t(t_1, t_0, q, q_0) \\ \gamma(t_1, t_0) \end{array} \right\}$$

$a, a_0$

$$I = \{ \begin{matrix} 00 & 01 \\ 01 & 00 \\ 10 & 10 \end{matrix} \}$$



# Finite State machine



# Thank You



14 Feb 2022

CS-230@IITB

11

**CADSL**

# Sequential Circuits

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

*CS-230: Digital Logic Design & Computer Architecture*

---



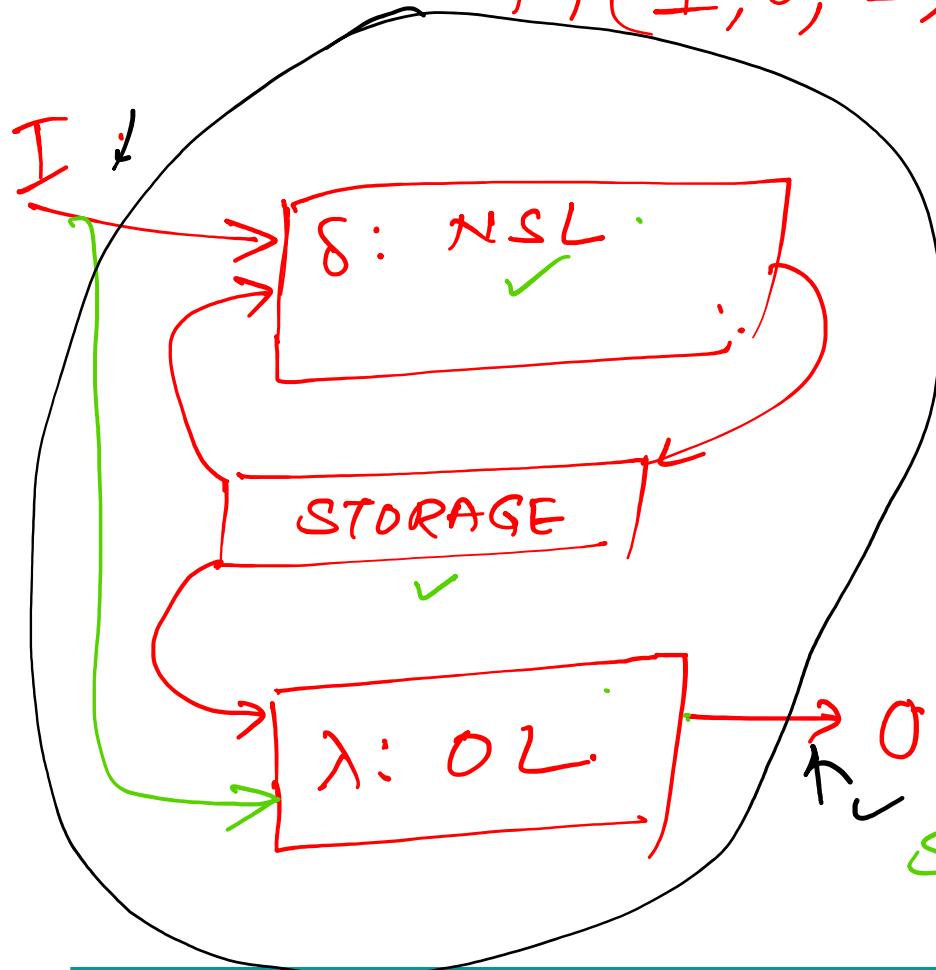
Lecture 18 (15 February 2022)

**CADSL**

Sequential  
C) temporal behaviour

circuits ]  
MEALY  
MOORE

$M(I, O, S, S_0, \delta, \lambda)$



to # storage element  
 $= \log_2^n$ .

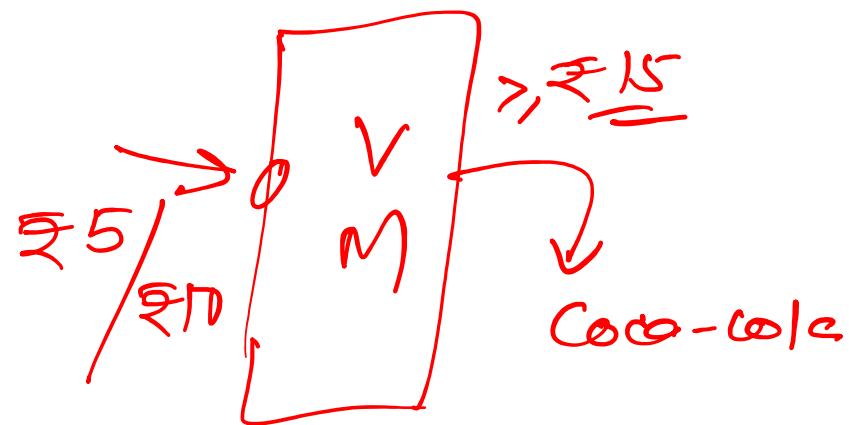
$n = \# \text{states in STS}$   
 $\Downarrow$   
 minimize # states

$\delta$  &  $\lambda$  are function of  
Current state + input  
State encoding

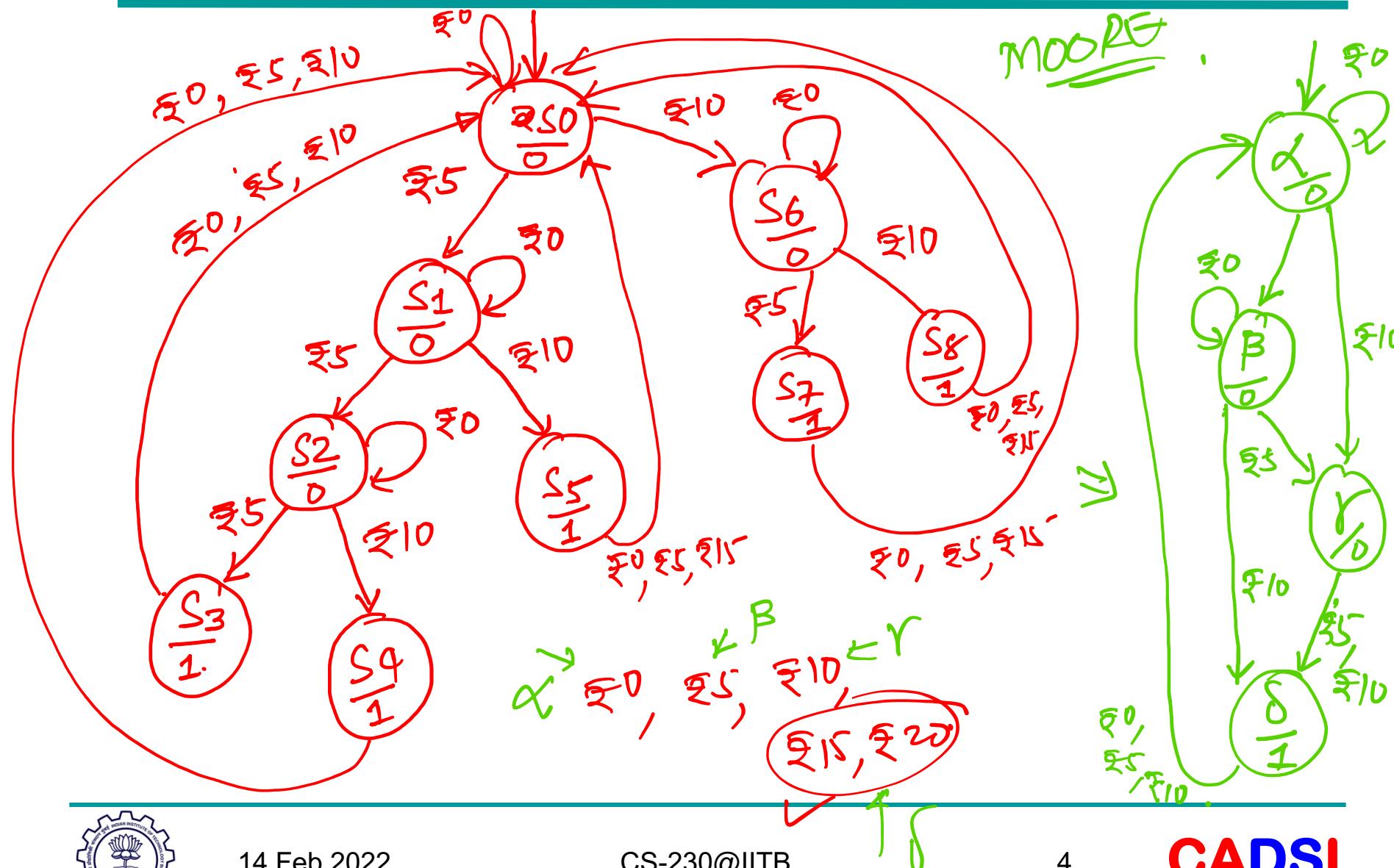


# State Machine

Minimize #States



# Finite State Machine



# State Minimization

---

X-Successor – If an input sequence X takes a machine from state  $S_i$  to state  $S_j$ , then  $S_j$  is said to be the X-successor of  $S_i$

✓ **Strongly connected**:– If for every pair of states ( $S_i, S_j$ ) of a machine M there exists an input sequence which takes M from state  $S_i$  to  $S_j$ , then M is said to be strongly connected

$$S_i \xrightarrow{0} S_k \xrightarrow{1} S_j$$

$$S_j \xrightarrow{0,1} S_i'$$

$$S_i \xrightarrow{x} S_j \checkmark$$

$$\begin{aligned} S_0 &\xrightarrow{\epsilon_0} S_0 \\ S_0 &\xrightarrow{\epsilon^5} S_1 \end{aligned}$$



# State Equivalence

---

- Two states  $S_i$  and  $S_j$  of machine M are **distinguishable** if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether  $S_i$  or  $S_j$  is the initial state

$S_i$

$S_j$

$S_i \xrightarrow{0} S_p \xrightarrow{1} S_q$   
 $S_j \xrightarrow{0} S_k \xrightarrow{1} S_t$

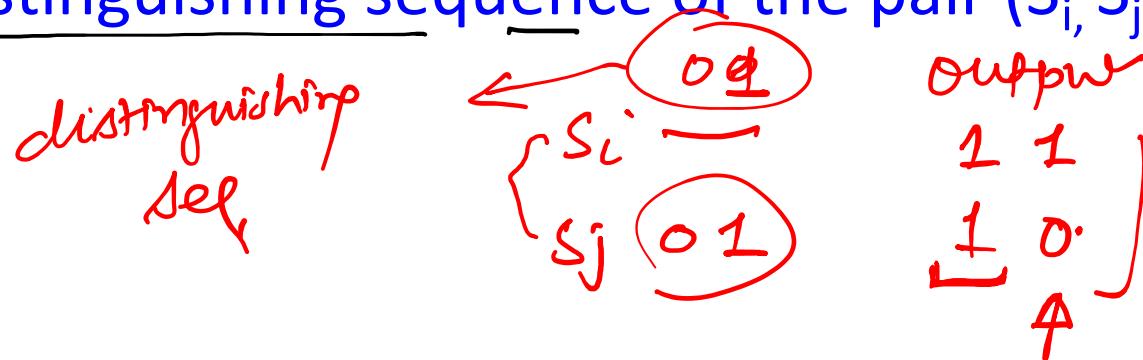
$\left. \begin{array}{l} S_i \xrightarrow{0} 1 \\ S_j \xrightarrow{0} 1 \end{array} \right\}$   
 $\left. \begin{array}{l} S_i \xrightarrow{1} 1 \\ S_j \xrightarrow{1} 1 \end{array} \right\}$



# State Equivalence

---

- Two states  $S_i$  and  $S_j$  of machine M are **distinguishable** if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether  $S_i$  or  $S_j$  is the initial state
- The sequence which distinguishes these states is called a distinguishing sequence of the pair  $(S_i, S_j)$



# State Equivalence

---

- Two states  $S_i$  and  $S_j$  of machine M are **distinguishable** if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether  $S_i$  or  $S_j$  is the initial state
- The sequence which distinguishes these states is called a **distinguishing sequence** of the pair  $(S_i, S_j)$
- If there exists for pair  $(S_i, S_j)$  a distinguishing sequence of length k, the states in  $(S_i, S_j)$  are said to be k-distinguishable

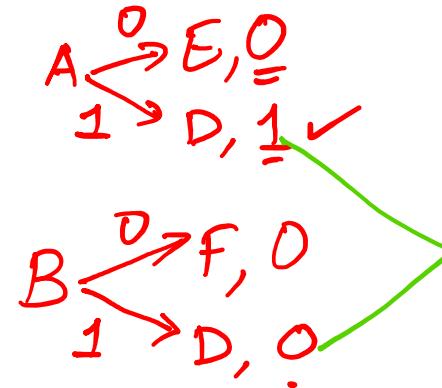


# State Equivalence

Machine M1 ✓

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$(A, B) - 1$  Distinguishable



# State Equivalence

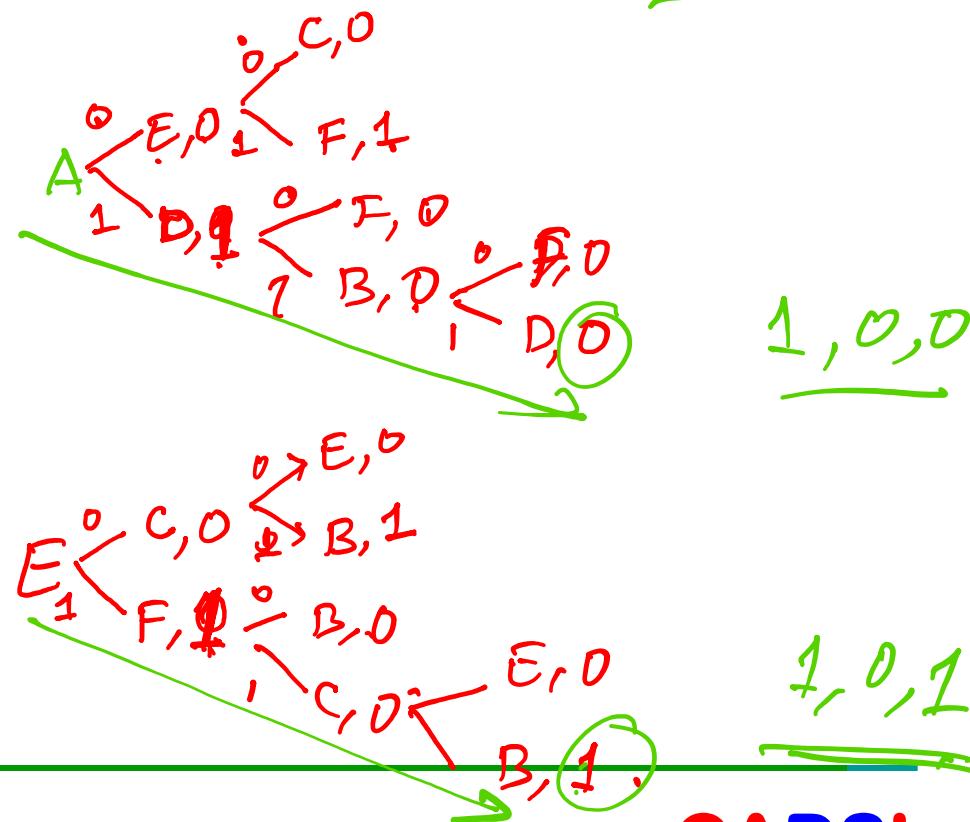
Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

Q - Indistinguishable

(A, E) – 3 Distinguishable

Seq - 111



# State Equivalence

---

## Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

(A, B) – 1 Distinguishable

(A, E) – 3 Distinguishable

Seq - 111

k-equivalent – The states that are not k-distinguishable are said to be k-equivalent

Also r-equivalent r < k



# Distinguishable States

---

K-equivalent

K-indistinguishable.



# State Equivalence

---

- States  $S_i$  and  $S_j$  of machine  $M$  are said to be equivalent if and only if, for every possible input sequence, the same output sequence will be produced regardless of whether  $S_i$  or  $S_j$  is the initial state
- States that are  $k$ -equivalent for all  $k < n-1$ , are equivalent
- $S_i \equiv S_j$ , and  $S_j \equiv S_k$ , then  $S_i \equiv S_k$

$n = \frac{\text{no. of states in state } m/c}{r}$



# State Equivalence

---

- The set of states of a machine M can be partitioned into disjoint subsets, known as equivalence classes
- Two states are in the same equivalence class if and only if they are equivalent, and are in different classes if and only if they are distinguishable

**Property:** If  $S_i$  and  $S_j$  are equivalent states, their corresponding X-successors, for all X, are also equivalent

---



# State Minimization Procedure

---

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $I_i$  successors, for all possible  $I_i$ , are also 1-equivalent

MOORE MINIMIZATION

$$\underline{P_0} = (\text{ABCDEF})$$

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0



# State Minimization Procedure

---

1. Partition the states of  $M$  into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $\underline{l_i}$  successors, for all possible  $l_i$ , are also *1-equivalent*

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$$P_0 = (ABCDEF)$$

$$P_1 = (\underline{\text{ACE}}), (\underline{\text{BDF}})$$



# State Minimization Procedure

1. Partition the states of  $M$  into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $l_i$  successors, for all possible  $l_i$ , are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$$P_0 = (ABCDEF)$$

$$P_1 = (\underline{ACE}), (\underline{BDF})$$

$$P_2 = (\underline{\underline{ACE}}), (\underline{BD}), (F)$$



# State Minimization Procedure

1. Partition the states of  $M$  into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $l_i$  successors, for all possible  $l_i$ , are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$$P_0 = (ABCDEF)$$

$$P_1 = (\textcolor{blue}{ACE}), (\textcolor{red}{BDF})$$

$$P_2 = \underline{(\textcolor{blue}{ACE})}, \underline{(\textcolor{brown}{BD})}, \underline{(\textcolor{red}{F})}$$

$$P_3 = \underline{\underline{(\textcolor{blue}{AC})}}, \underline{\underline{(\textcolor{blue}{E})}}, \underline{(\textcolor{brown}{BD})}, (\textcolor{red}{F})$$



# State Minimization Procedure

1. Partition the states of  $M$  into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $l_i$  successors, for all possible  $l_i$ , are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	✓D, 1
B	✓F, 0	D, 0
C	E, 0	✓B, 1
D	✓F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$$P_0 = (ABCDEF)$$

$$P_1 = (\textcolor{blue}{ACE}), (\textcolor{red}{BDF})$$

$$P_2 = (\textcolor{blue}{ACE}), (\textcolor{black}{BD}), (\textcolor{black}{F})$$

$$P_3 = (\textcolor{green}{AC}), (\textcolor{green}{E}), (\textcolor{green}{BD}), (\textcolor{black}{F})$$

$$P_4 = (\textcolor{green}{AC}), (\textcolor{green}{E}), (\textcolor{green}{BD}), (\textcolor{black}{F})$$



# State Minimization Procedure

1. Partition the states of  $M$  into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their  $l_i$  successors, for all possible  $l_i$ , are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$$P_0 = (ABCDEF)$$

$$P_1 = (\textcolor{blue}{ACE}), (\textcolor{red}{BDF})$$

$$P_2 = (\textcolor{blue}{ACE}), (\textcolor{black}{BD}), (\textcolor{black}{F})$$

$$P_3 = (\textcolor{black}{AC}), (\textcolor{blue}{E}), (\textcolor{black}{BD}), (\textcolor{black}{F})$$

$$P_4 = (\textcolor{black}{AC}), (\textcolor{blue}{E}), (\textcolor{black}{BD}), (\textcolor{black}{F})$$



# Machine Equivalence

- Two machines M1, M2 are said to be equivalent if and only if, for every state in M1, there is corresponding equivalent state in M2
- If one machine can be obtained from the other by relabeling its states they are said to be **isomorphic** to each other

PS	NS, z	
	$X = 0$	$X = 1$
AC - a	$\beta, 0$	$\gamma, 1$
E - $\beta$ .	$a, 0$	$\delta, 1$
BD - $\gamma$ .	$\delta, 0$	$\gamma, 0$
F - $\delta$	$\gamma, 0$	$a, 0$

4

# State Equivalence - Example

Machine M2

PS	NS, z	
	X = 0	X = 1
A	E, 0	C, 0
B	C, 0	A, 0
C	B, 0	G, 0
D	G, 0	A, 0
E	F, 1	B, 0
F	E, 0	D, 0
G	D, 0	G, 0

$$\underline{P_0} = (\text{ABCDEFG})$$

$$P_1 = (\text{ABCDEFG}) (\text{E})$$

$$P_2 = (\text{AF}) (\text{BCDG}) (\text{E})$$

$$P_3 = (\text{AF}) (\text{BD}) (\text{CG}) (\text{E})$$

$$P_4 = (\text{A}) (\text{F}) (\text{BD}) (\text{CG}) (\text{E})$$

$$P_5 = (\text{A}) (\text{F}) (\text{BD}) (\text{CG}) (\text{E})$$

S'



# Thank You



# Sequential Circuits

## Equivalence of State Machines

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.cse.iitb.ac.in/~viren/>

E-mail: viren@{cse, ee}.iitb.ac.in

*CS-230: Digital Logic Design & Computer Architecture*

---



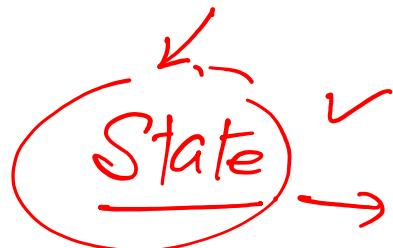
Lecture 19 (17 February 2022)

**CADSL**

## State machine

↳ temporal. behaviour.

## Finite State Machine (FSM)



STG (State Transition Graph)  
↓

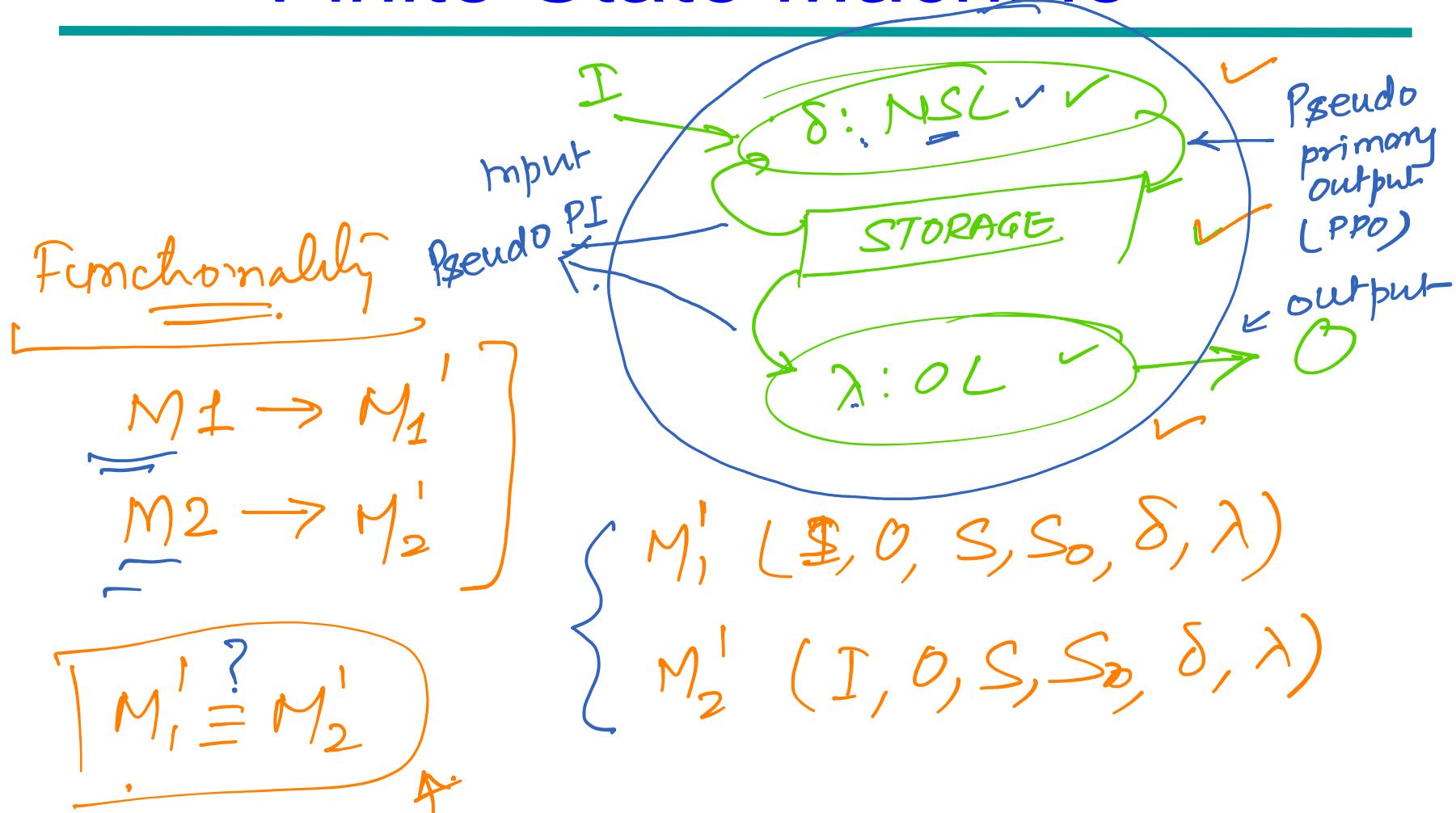
minimization (automatic)

↓  
Equivalence checking

↓  
minimized State machine.



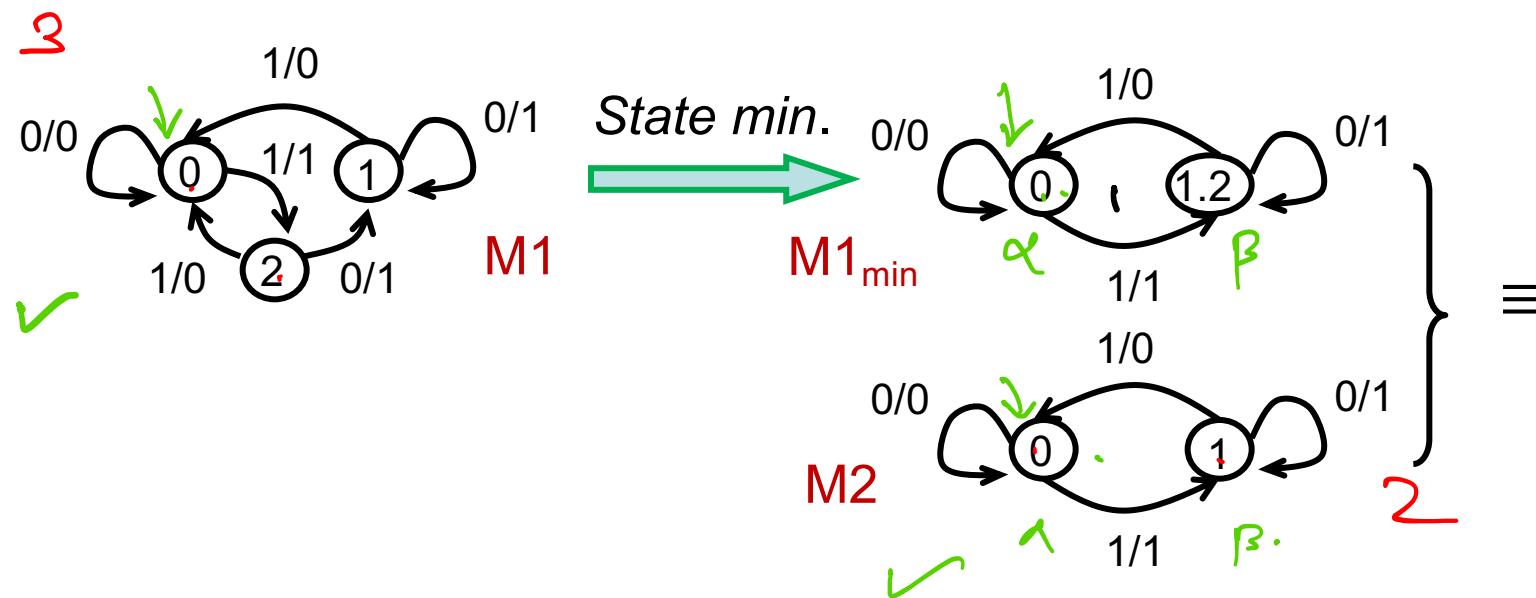
# Finite State Machine



# Sequential Verification

- Approach 1: Based on isomorphism of state transition graphs
  - two machines  $M_1, M_2$  are *equivalent* if their state transition graphs (STGs) are *isomorphic*
  - perform state minimization of each machine
  - check if  $\text{STG}(M_1)$  and  $\text{STG}(M_2)$  are isomorphic

$M'_1$        $M'_2$



# Machine Equivalence

---

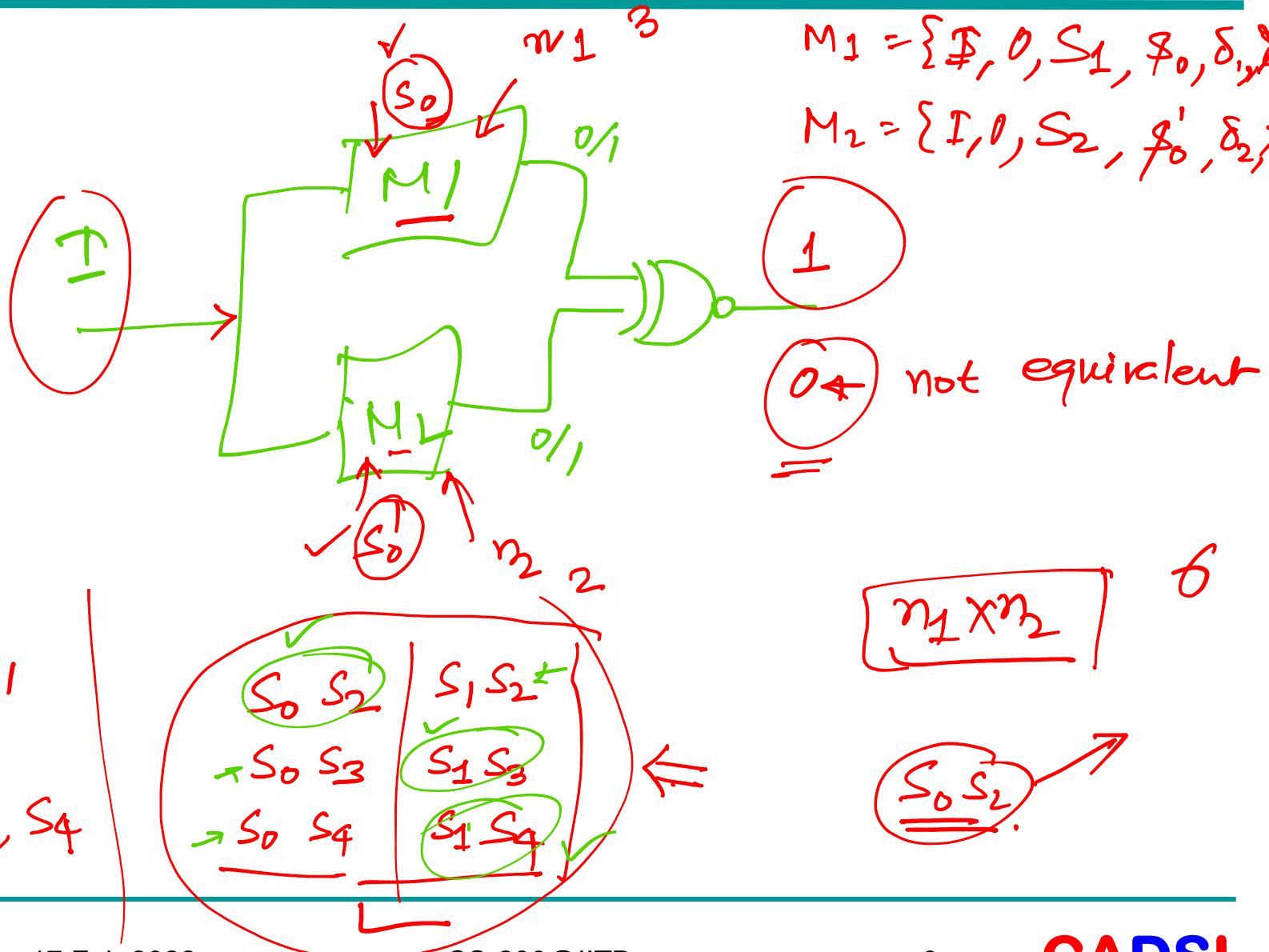
- Two machines  $M_1, M_2$  are said to be equivalent if and only if, for every state in  $M_1$ , there is corresponding equivalent state in  $M_2$
- If one machine can be obtained from the other by relabeling its states they are said to be **isomorphic** to each other



# Finite State Machine

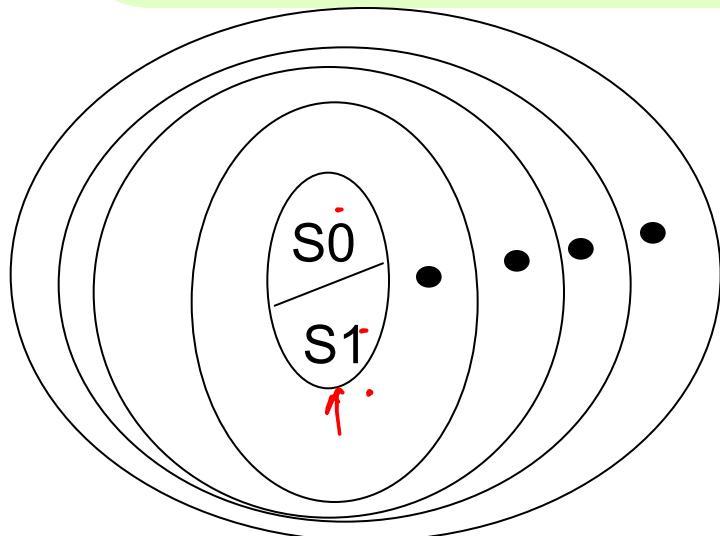
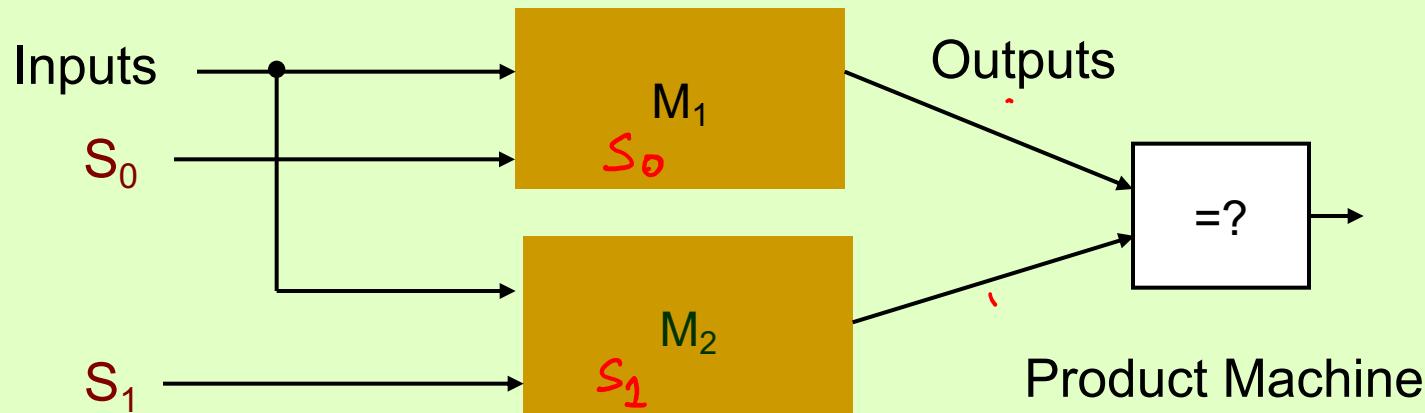
(D)

10  
1024



# Reachability-Based Equivalence Checking

Approach 2: Symbolic Traversal Based Reachability Analysis ✓



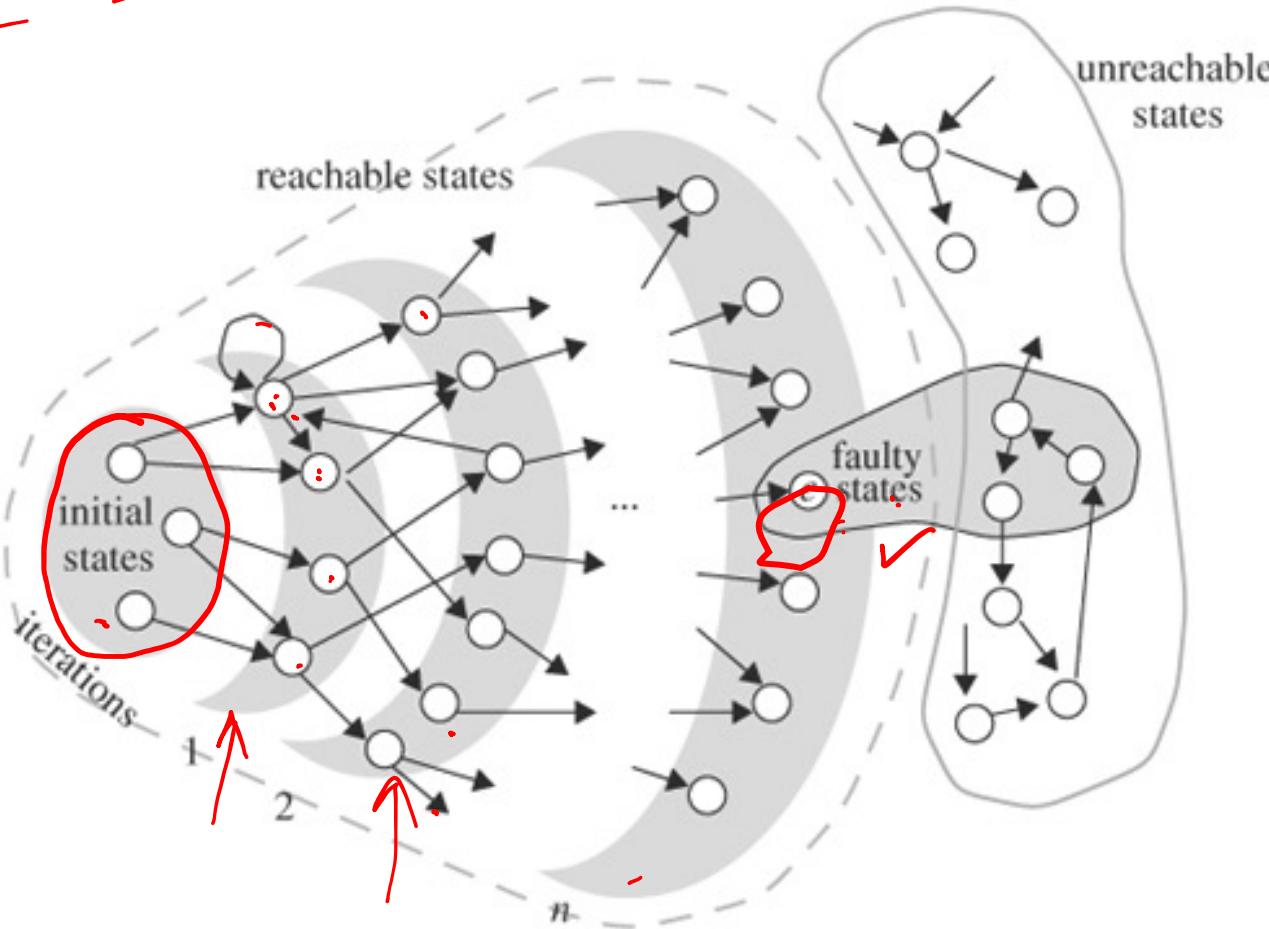
- Build product machine of  $M_1$  and  $M_2$
- Traverse state-space of product machine starting from reset states  $S_0, S_1$
- Test equivalence of outputs in each state
- Can use any state-space traversal technique

$$M \cong M_1 \times M_2$$



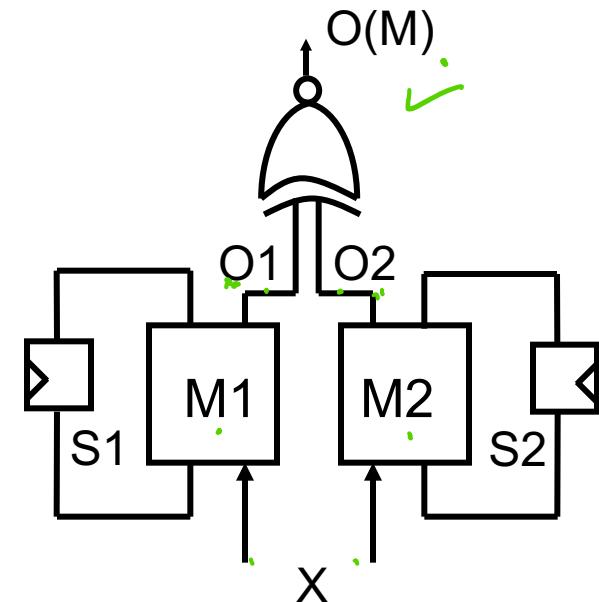
# Forward Reachability

Counter example



# Sequential Verification

- Symbolic FSM traversal of the product machine
- Given two FSMs:  $M_1(X, S_1, \delta_1, \lambda_1, O_1)$ ,  $M_2(X, S_2, \delta_2, \lambda_2, O_2)$
- Create a product FSM:  $M = M_1 \times M_2$ 
  - traverse the states of  $M$  and check its output for each transition
  - the output  $O(M) = 1$ , if outputs  $O_1 = O_2$
  - if all outputs of  $M$  are 1,  $M_1$  and  $M_2$  are equivalent
  - otherwise, an error state is reached
  - error trace is produced to show:  $M_1 \neq M_2$



# Product Machine - Construction

- Define the product machine  $M(X, S, S^0, \delta, \lambda, O)$   $|S| = |S_1| \times |S_2|$

– states,

$$S = S_1 \times S_2$$

– next state function,  $\delta(s, x) : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$

– output function,  $\lambda(s, x) : (S_1 \times S_2) \times X \rightarrow \{0, 1\}$

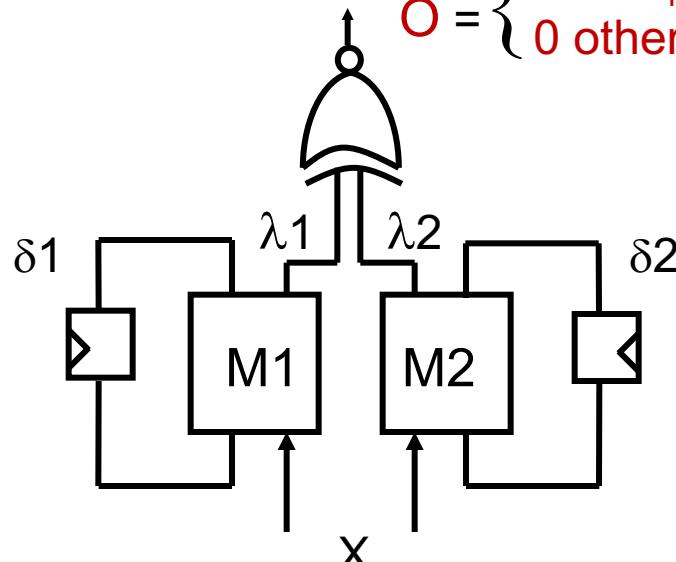
$$\lambda(s_1, x) = \lambda_1(s_1, x)$$
$$\lambda(s_2, x)$$

$$\lambda(s, x) = \lambda_1(s_1, x) \oplus \lambda_2(s_2, x)$$

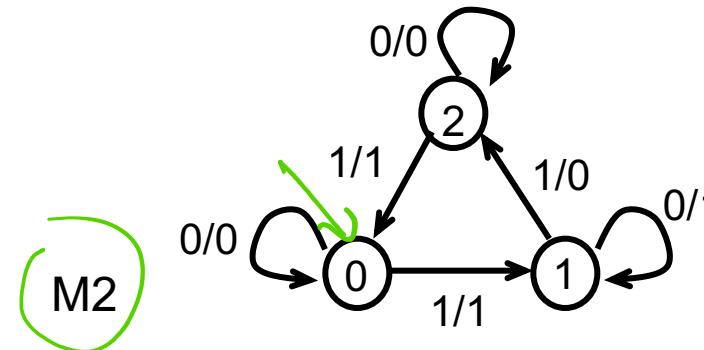
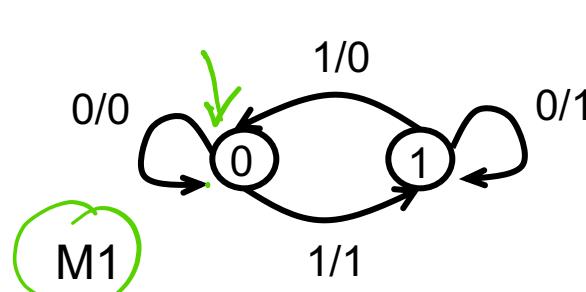
$$O = \begin{cases} 1 & \text{if } O_1 = O_2 \\ 0 & \text{otherwise} \end{cases}$$

- Error trace (*distinguishing sequence*) that leads to an error state

- sequence of inputs which produces 1 at the output of  $M$
- produces a state in  $M$  for which  $M_1$  and  $M_2$  give different outputs



# FSM Traversal in Action

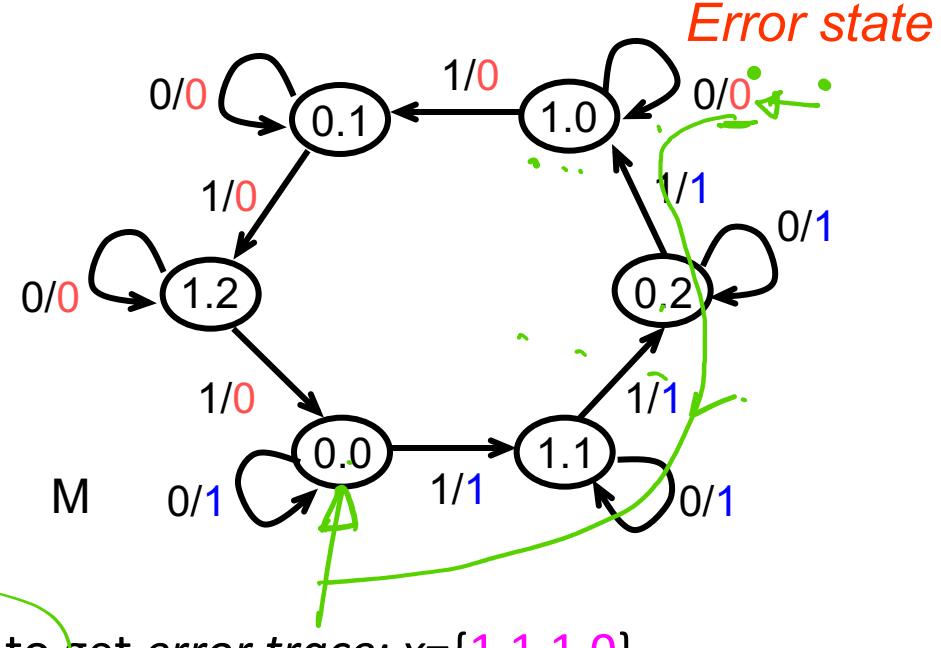


Initial states:  $s_1=0$ ,  $s_2=0$ ,  $s=(0,0)$

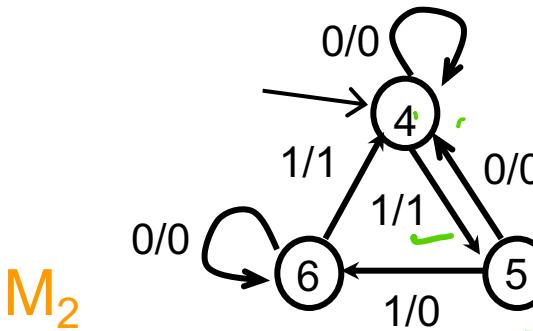
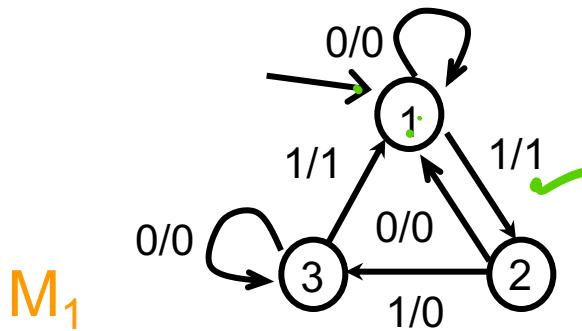
<i>State reached</i>	$Out(M)$ $x=0 \ x=1$
----------------------	-------------------------

- $New^0 = (0.0)$  1 1
  - $New^1 = (1.1)$  1 1
  - $New^2 = (0.2)$  1 1
  - $New^3 = (1.0)$  0 0

- **STOP** - backtrack to initial state to get error trace:  $x=\{1,1,1,0\}$



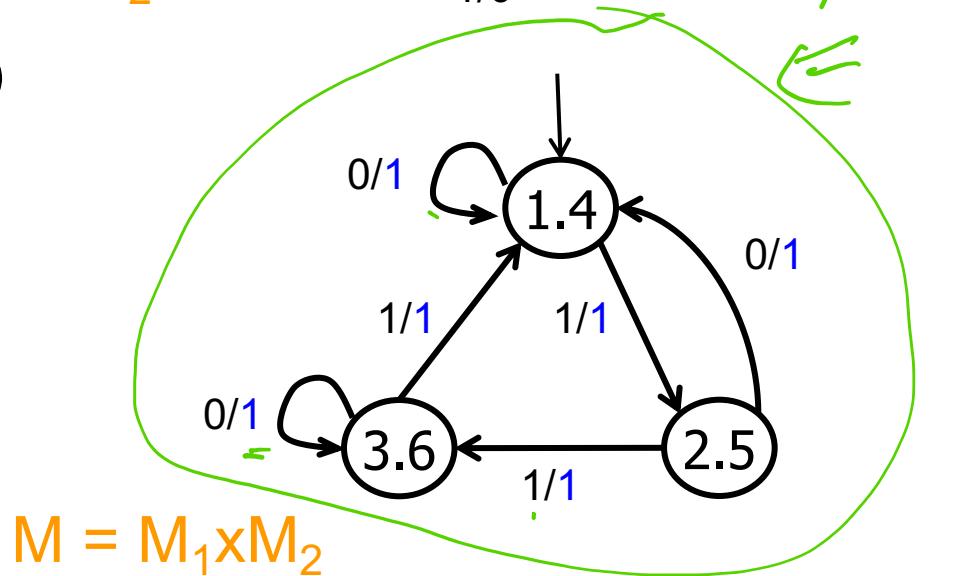
# FSM Traversal in Action



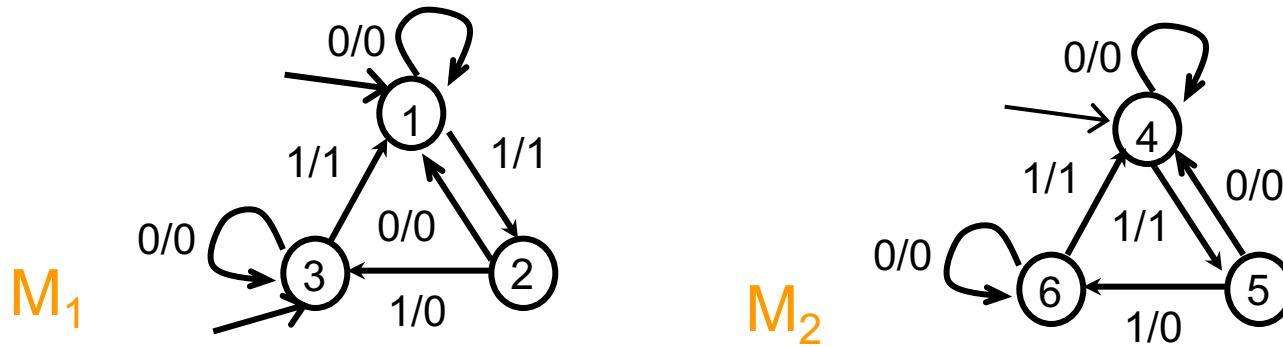
Initial states:  $s_1=1$ ,  $s_2=4$ ,  $s=(1.4)$

	$Out(M)$
State reached	$x=0 \quad x=1$

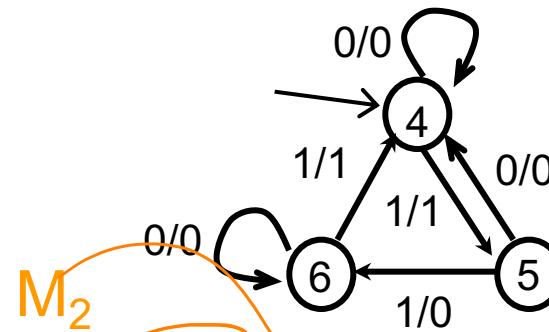
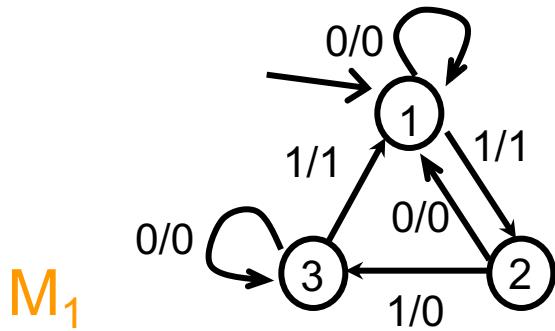
- $New^0 = (1.4) \quad 1 \quad 1$
- $New^1 = (2.5) \quad 1 \quad 1$
- $New^2 = (3.6) \quad 1 \quad 1$



# FSM Traversal in Action



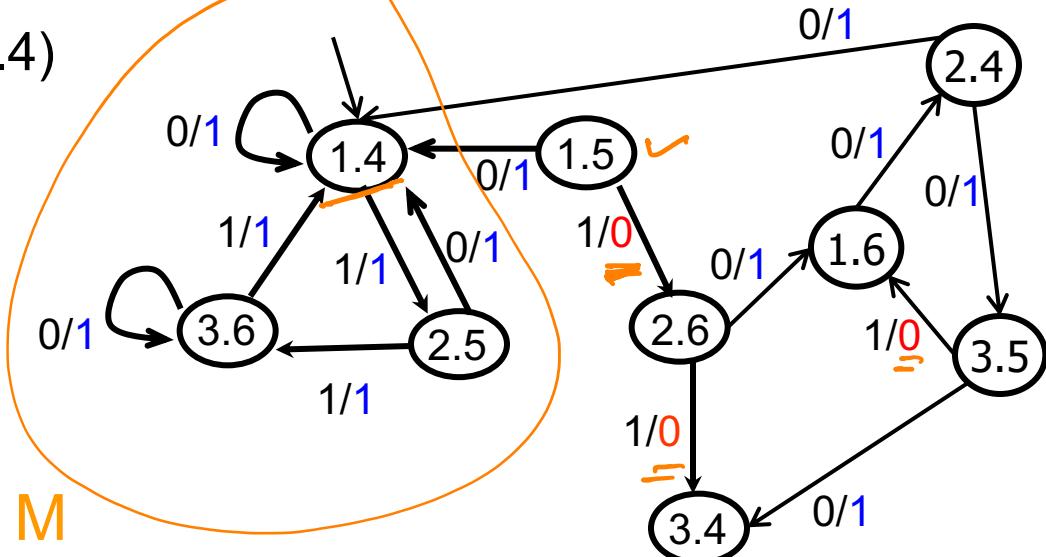
# FSM Traversal in Action



Initial states:  $s_1=1$ ,  $s_2=4$ ,  $s=(1,4)$

	$Out(M)$
State reached	$x=0 \quad x=1$

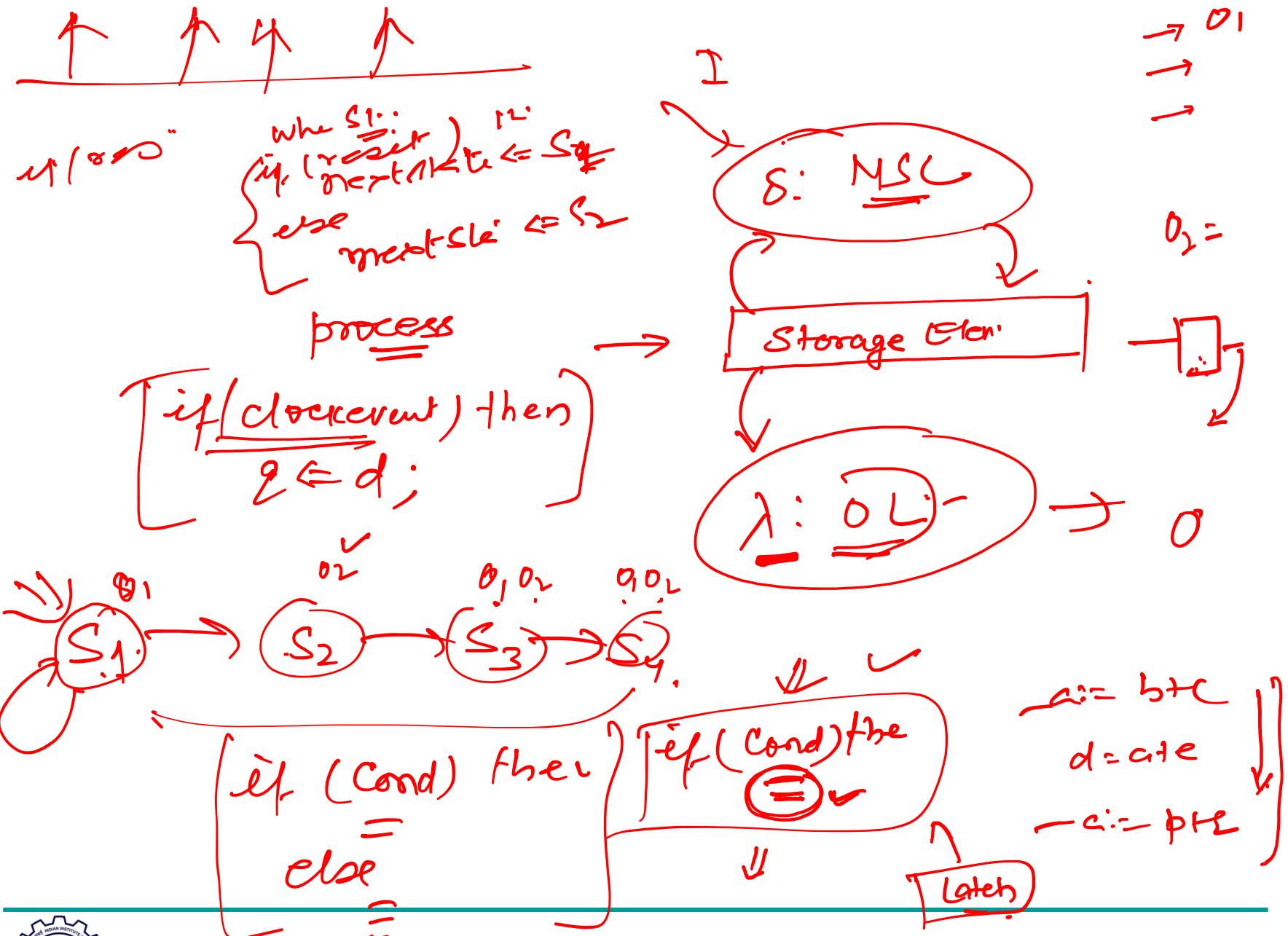
- $New^0 = (1,4) \quad 1 \quad 1$
- $New^1 = (2,5) \quad 1 \quad 1$
- $New^2 = (3,6) \quad 1 \quad 1$



- Erroneous states are not reachable

$$M \models \Gamma, D, S, S_0, \delta, \lambda$$

$$M \models M_2$$



# FSM Traversal - Algorithm

---

- Traverse the product machine  $M(X, S, \delta, \lambda, O)$ 
  - start at an initial state  $S_0$
  - iteratively compute symbolic image  $\text{Img}(S_0, R)$  (set of *next states*):

$$\text{Img}(S_0, R) = \exists_x \exists_s S_0(s).R(x, s, t)$$

$$R = \prod_i R_i = \prod_i (t_i \equiv \delta_i(s, x))$$

until an *error state* is reached

- transition relation  $R_i$  for each next state variable  $t_i$  can be computed as  $t_i = (t \otimes \delta(s, x))$   
(this is an alternative way to compute transition relation, when design is specified at gate level)



# Thank You



17 Feb 2022

CS-230@IITB

17

**CADSL**

# Logic Testing

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.ee.iitb.ac.in/~viren/>

E-mail: [viren@ee.iitb.ac.in](mailto:viren@ee.iitb.ac.in)

*Cs-226: Digital Logic Design*

---



*Lecture 32-A: 20 April 2021*

**CADSL**

# Digital System Realization Process

Customer's need

Determine requirements

Write specifications

ADL

Design synthesis and Verification

Test development

Fabrication

Manufacturing test

① Discrete component  
(PCB)

② Integrated circuits  
(IC)

Chips to customer

System

2

CADSL



# Optimization Parameters

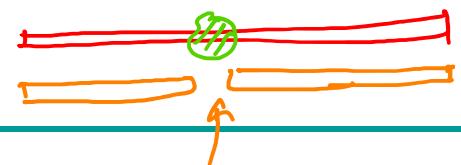
---

- Area (Cost)
  - No. of gates (switches)
- Performance (delay)
  - No. of switches in signal propagation path (Input to output)
- Power
  - No of gates (indirectly the area as first order model)
- Testability

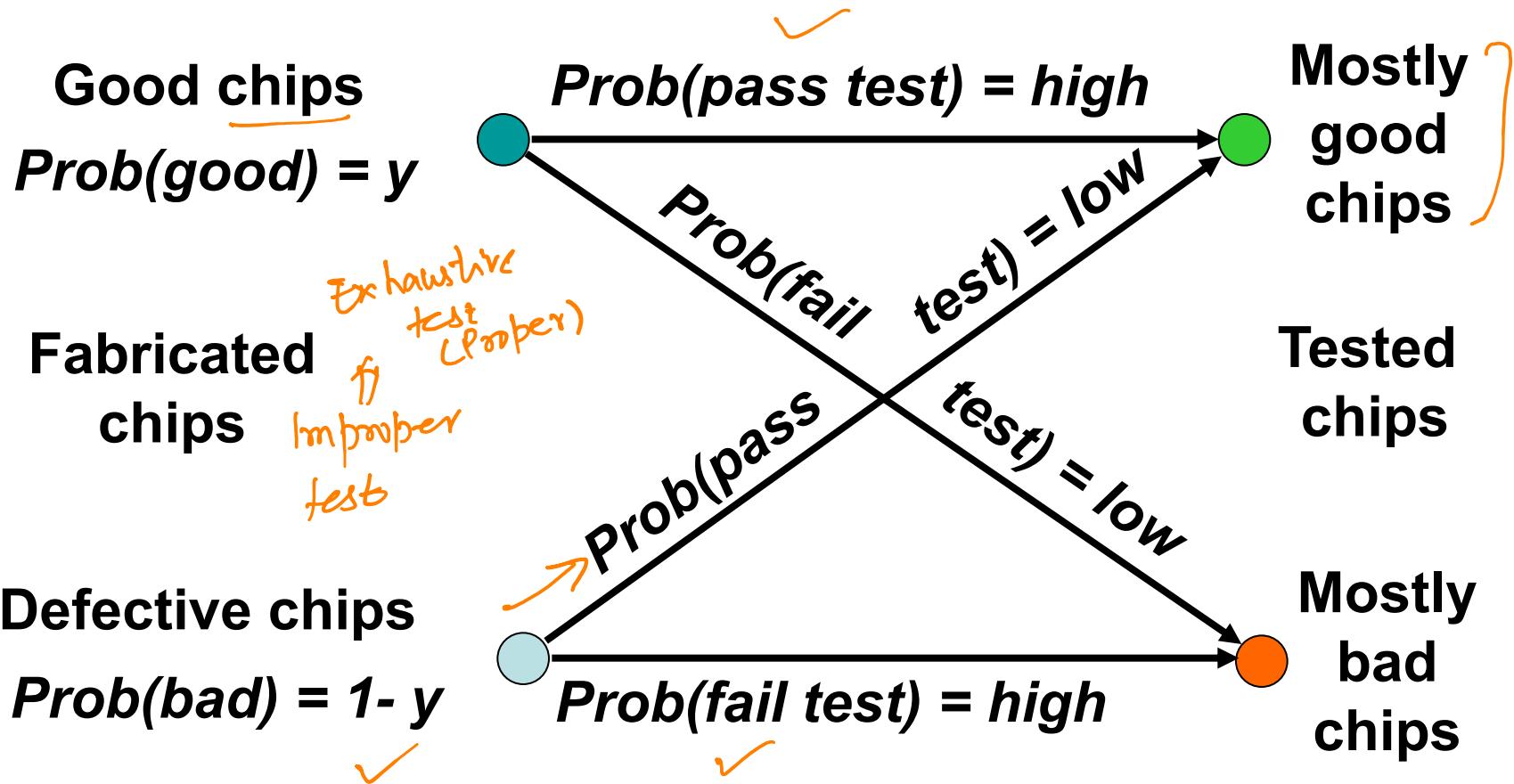


# Definitions ✓

- ❖ Design synthesis: Given an I/O function, develop a procedure to manufacture a device using known materials and processes. BA, K-Map, QM ✓
- ❖ Verification: Predictive analysis to ensure that the synthesized design, when manufactured, will perform the given I/O function. ROBDD / SAT
- ❖ Test: A manufacturing step that ensures that the physical device, manufactured from the synthesized design, has no manufacturing defect.



# Testing as Filter Process

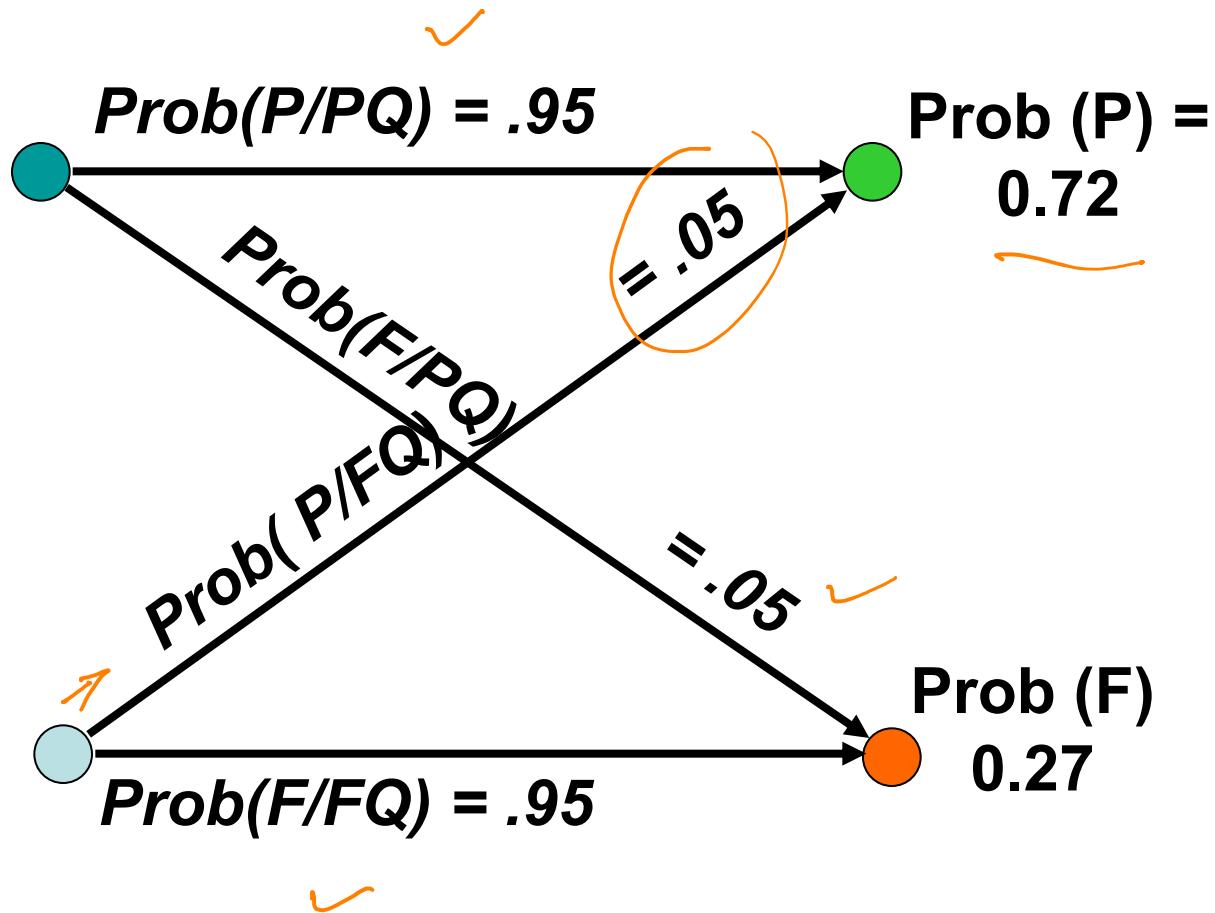


# Students Examination

Pass quality  
 $\text{Prob}(PQ) = .75$

All Students

Fail quality  
 $\text{Prob}(FQ) = .25$



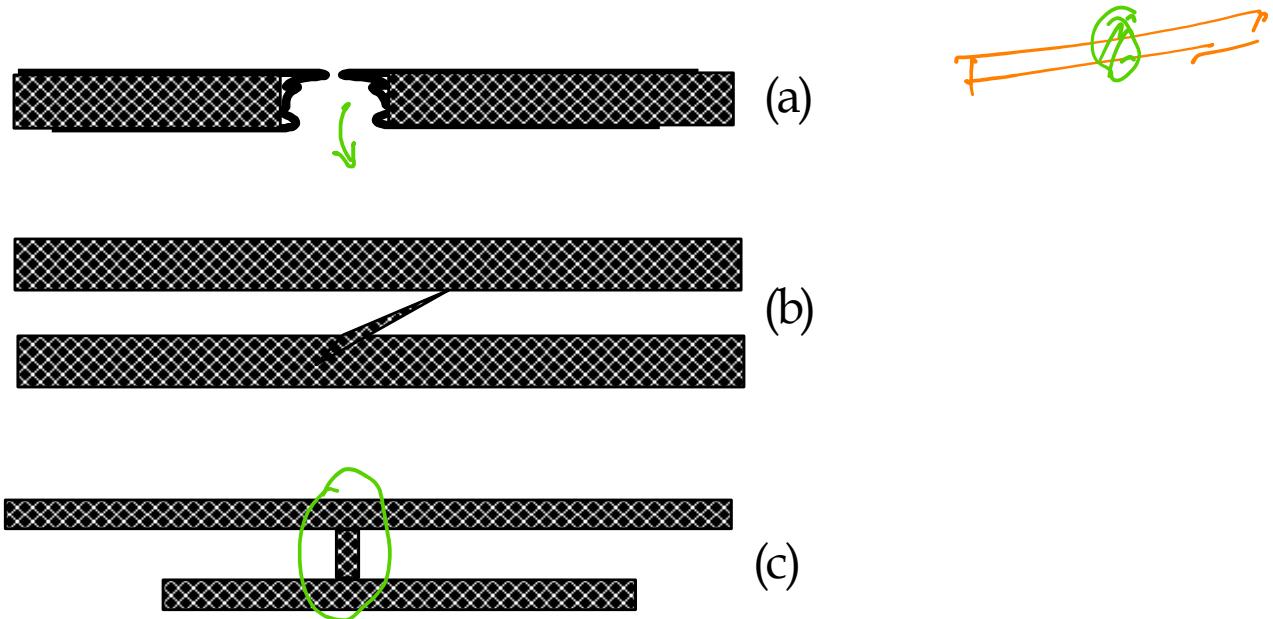
# Ideal Tests

---

- Ideal tests **detect all defects** produced in the manufacturing process.
- Ideal tests pass all functionally good devices.
- Very large numbers and varieties of possible defects need to be tested..



# Defect: Electromigration



- (a) Open in a line
- (b) Short between two lines (whisker)
- (c) Short between lines on different layers (hillock)

# Real Tests

- Based on analyzable fault models, which may not map on real defects.
- Incomplete coverage of modeled faults due to high complexity.
- Some good chips are rejected. The fraction (or percentage) of such chips is called the yield loss.
- Some bad chips pass tests. The fraction (or percentage) of bad chips among all passing chips is called the defect level.  


DL  
minimize



# Roles of Testing

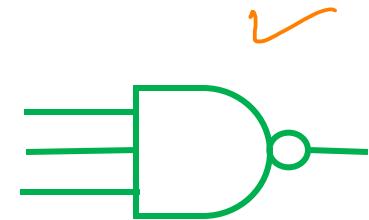
---

- ❖ **Detection:** Determination whether or not the *device under test* (DUT) has some fault. *→ modelled faults*
- ❖ **Diagnosis:** Identification of a specific fault that is present on DUT.
- ❖ Device characterization: Determination and correction of errors in design and/or test procedure.
- ❖ **Failure mode analysis (FMA):** Determination of manufacturing process errors that may have caused defects on the DUT.



# IC Testing is a Difficult Problem

- Need  $2^3 = 8$  input patterns to exhaustively test a 3-input NAND
- $2^N$  tests needed for N-input circuit
- Many ICs have > 100 inputs



3-input NAND

$$2^{100} = 1.27 \times 10^{30}$$

$\frac{10^{30}}{10^9} = 10^{21}$

Applying  $10^{30}$  tests at  $10^9$  per second (1 GHZ) will require  $10^{21}$  secs = 400 billion centuries!

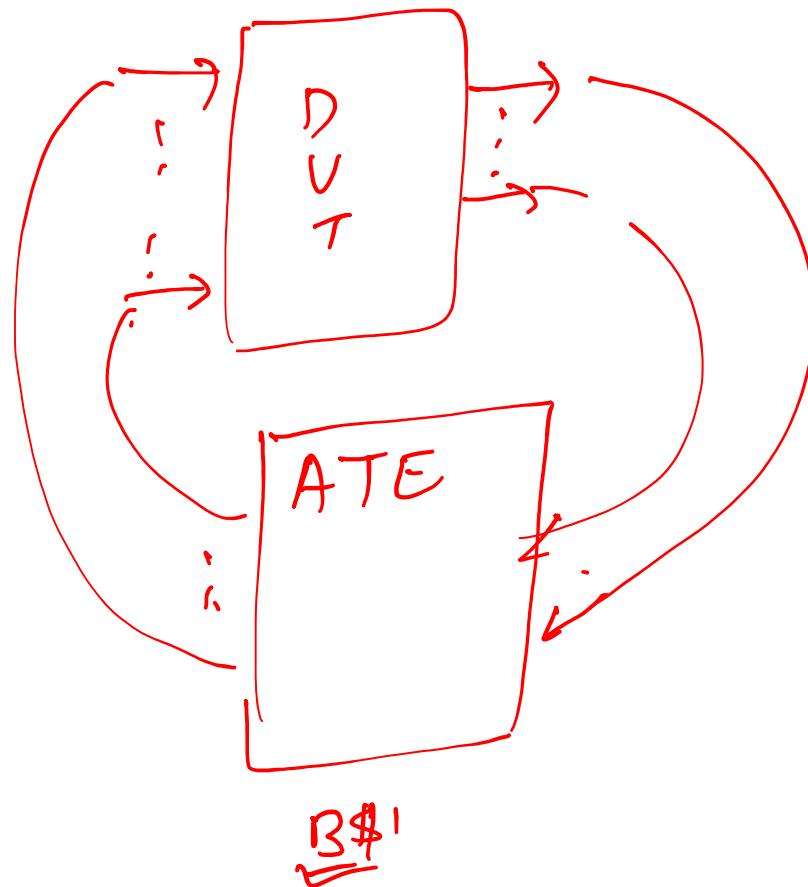
→ impractical

- Only a very few input combinations can be applied in practice

reasonable time ?



Reasonable



₹ ५-६ / sec

1 minute ✓

$$5 \times 60 = \underline{\underline{₹ 300}}$$

Reasonable time

↪ few seconds

↓  
application



# IC Testing in Practice

---

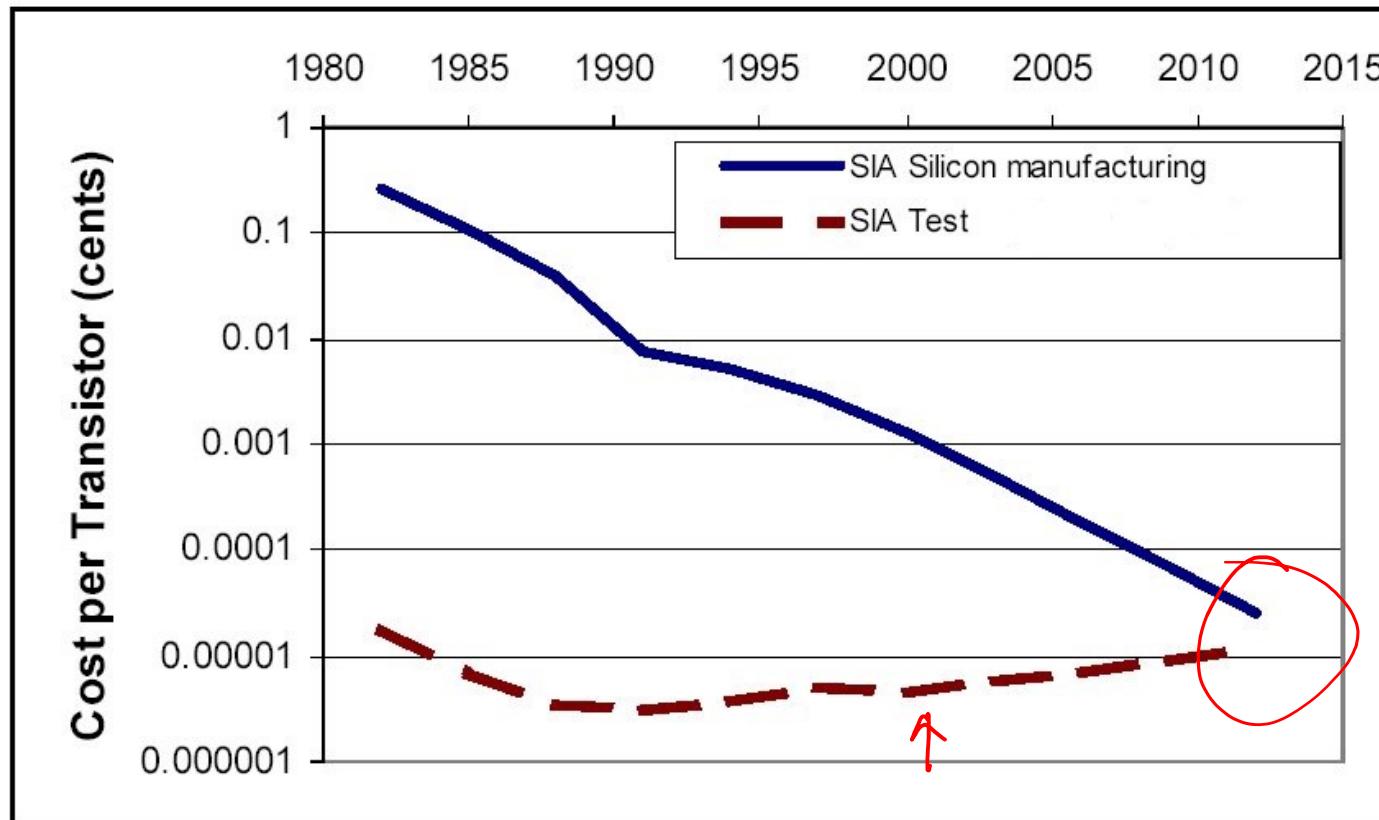
For high end circuits ✓

- A few seconds of test time on very expensive production testers (*LATE*)
  - Many thousand test patterns applied
  - Test patterns carefully chosen to detect
  - High economic impact  
- test costs are approaching manufacturing costs
- Exhaustive*  
*3 Years → few secnd.*



# Microprocessor Cost per Transistor

Cost of testing will *EXCEED* cost of design/manufacturing



(Source: ITR-Semiconductor, 2002)



# Definitions

---

- ❖ **Defect:** A defect in an electronic system is the unintended difference between the implemented hardware and its intended design
  
  - ❖ **Error:** A wrong output signal produced by defective system is called error. An error is an effect whose cause is some defect ] 1/10
  
  - ❖ **Fault:** A representation of a defect at the abstracted function level is called a fault FAULT
- 



# Why Model Faults?

---

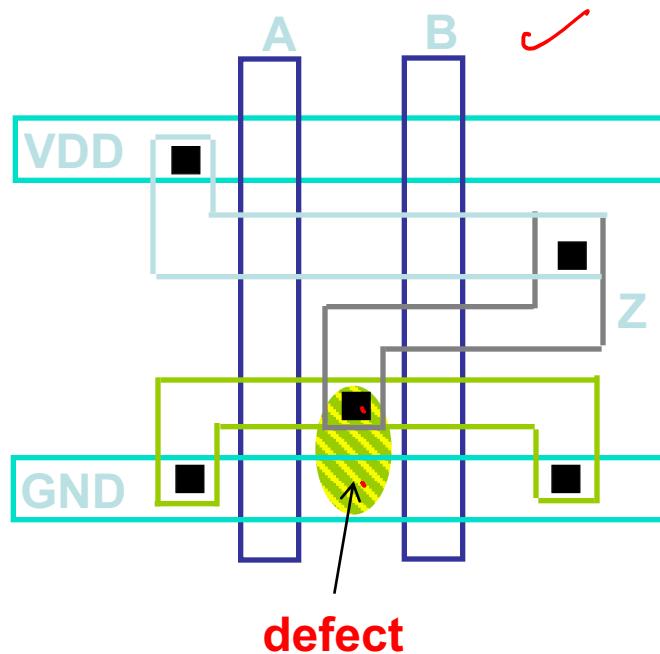
- ❖ I/O function tests inadequate for manufacturing (functionality versus component and interconnect testing)  

- ❖ Real defects (often mechanical) too numerous and often not analyzable
- ❖ A fault model identifies targets for testing
- ❖ A fault model makes analysis possible
- ❖ Effectiveness measurable by experiments

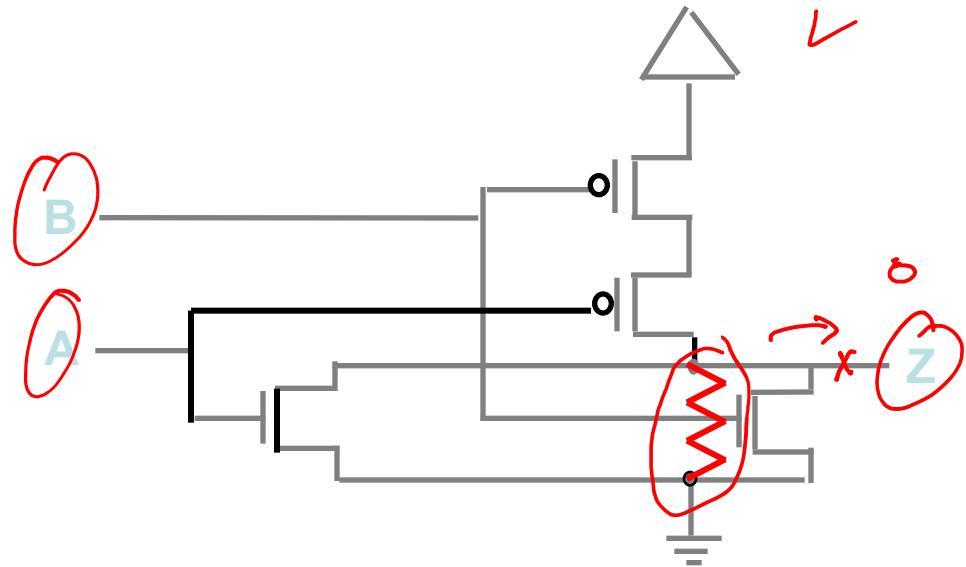


# Fault Modeling

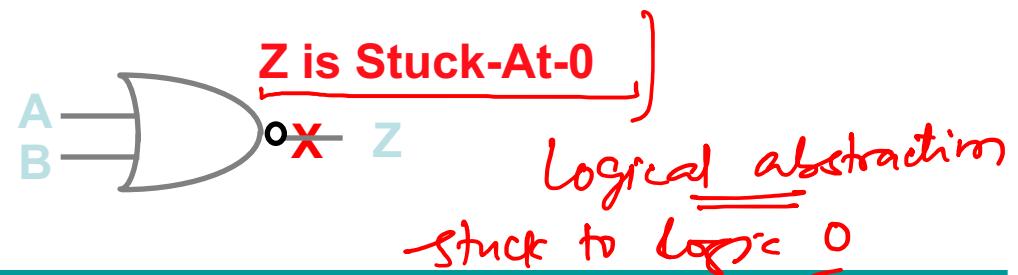
- Physical Model



- Electrical Model



- Logical Model

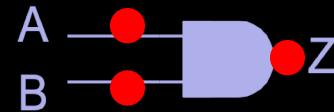


# Stuck-At Fault as a Logic Fault

- Stuck-at Fault is a *Functional Fault* on a Boolean (Logic) Function Implementation
- It is not a Physical Defect Model
  - Stuck-at 1 does not imply line is shorted to  $V_{DD}$
  - Stuck-at 0 does not imply line is grounded!
- It is an Abstract fault model
  - A logic stuck-at 1 means when the line is applied a logic 0, it produces a logical error
  - A Logic Error means 0 becomes 1 or vice versa
- It is independent of the underlying technology
  - CMOS, BJT, III-V Semiconductor, Carbon nanotubes etc.



# SA Faults



Inputs	FF	Faulty Response					
AB	Response	A/0	B/0	Z/0	A/1	B/1	Z/1
-----							
{ 00	0	0	0	0	0	0	1
	01	0	0	0	1	0	1
	10	0	0	0	0	1	1
	11	1	0	0	0	1	1

# faults  $\propto$  # nets

$$\approx g.(\underline{\text{nets}})$$

fault

no uniqueness

SINGLE  
STUCK - AT  
FAULT  
MODEL

Fault  $\rightarrow$  detected by multiple vectors  
Vector  $\rightarrow$  can detect multiple faults

optimization

Select minimum no. of vectors to detect all possible faults.



# Thank You



# Logic Testing

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.ee.iitb.ac.in/~viren/>

E-mail: [viren@ee.iitb.ac.in](mailto:viren@ee.iitb.ac.in)

*CS-226: Digital Logic Design*

---



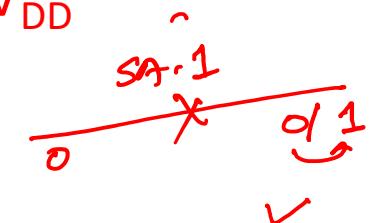
Lecture 32-B: 20 April 2021

**CADSL**

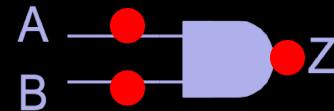
# Stuck-At Fault as a Logic Fault

---

- Stuck-at Fault is a *Functional Fault* on a Boolean (Logic) Function Implementation
- It is not a Physical Defect Model
  - Stuck-at 1 does not imply **line is shorted to  $V_{DD}$**
  - Stuck-at 0 does not imply **line is grounded!**
- It is an **Abstract fault model**
  - A logic stuck-at 1 means when the line is applied a logic 0, it produces a logical error
  - A Logic Error means 0 becomes 1 or vice versa
- It is independent of the underlying technology
  - CMOS, BJT, III-V Semiconductor, Carbon nanotubes etc.



# SA Faults



Inputs	FF	Faulty Response
AB	Response	A/0 B/0 Z/0 A/1 B/1 Z/1

00	0	0 0 0 0 0 1
01	0	0 0 0 1 0 1
10	0	0 0 0 0 1 1
11	1	0 0 0 1 1 1

$\{B_1, Z_1\}$   
 $\{A_0, B_0, Z_0\}$



Fault → detectable by multiple test vectors

Vector → can detect multiple faults

OPTIMIZE

↑ # test vectors

s.t TV<sub>s</sub> can cover all  
the faults

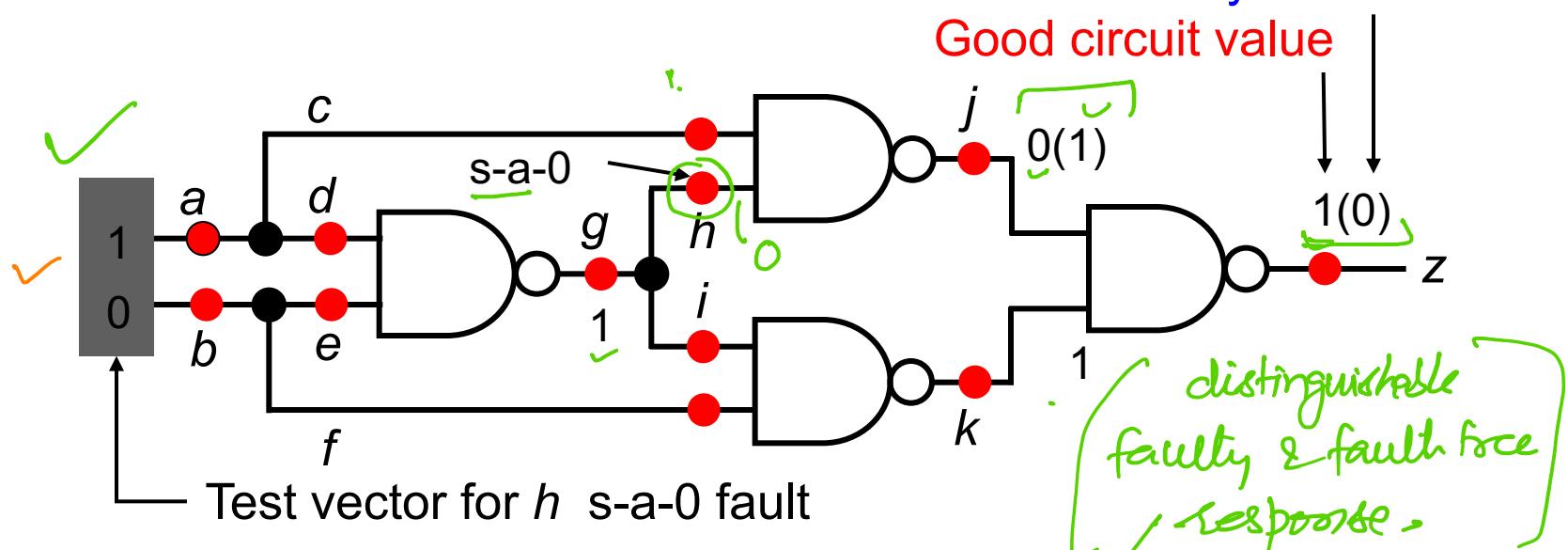
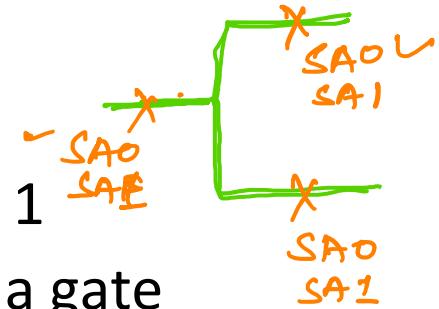
Functional X

Structural test ✓



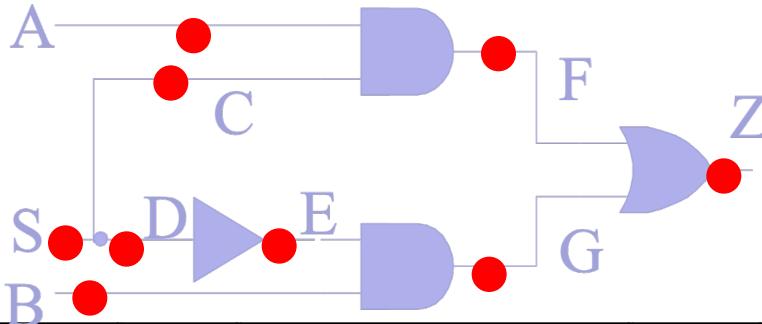
# Single Stuck-at Fault

- Three properties define a single stuck-at fault
  - Only one line is faulty
  - The faulty line is permanently set to 0 or 1
  - The fault can be at an input or output of a gate
- Example: XOR circuit has 12 fault sites (●) and 24 single stuck-at faults



# SA Faults

MUX



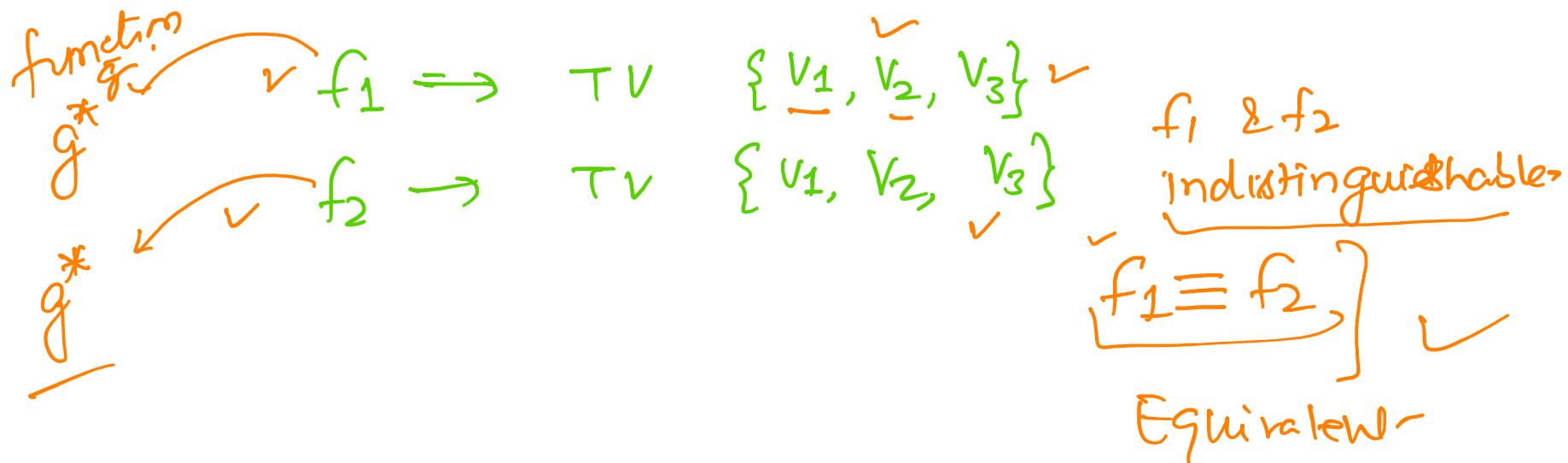
Inp	Response							
	SA	FF	S/0	S/1	C/0	C/1	D/0	D/1
000	000	0	0	0	0	0	0	0
001	001	1	1	0	1	1	1	0
010	010	0	0	1	0	1	0	0
011	011	1	1	1	1	1	1	0
100	100	0	0	0	0	0	0	0
101	101	0	1	0	0	0	1	0
110	110	1	0	1	0	1	1	1
111	111	1	1	1	0	1	1	1



# minimum no. of test vectors

Set of faults → take one fault & generate the vector which can detect

$\min \{\# \text{ faults}\}$  → for which we need to generate TV.



# Fault Equivalence

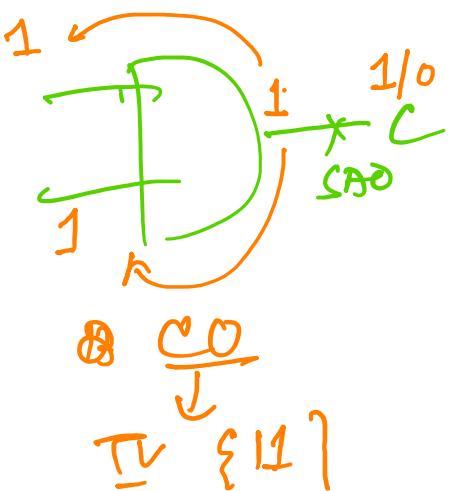
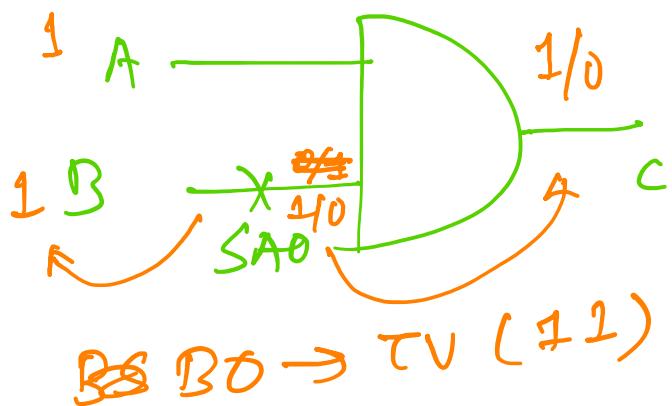
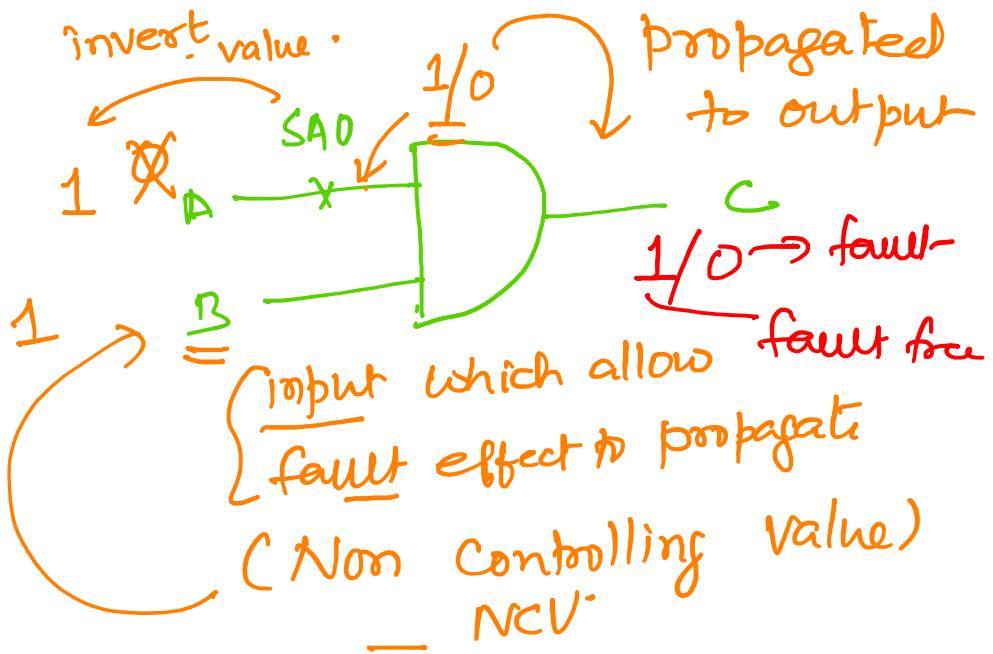
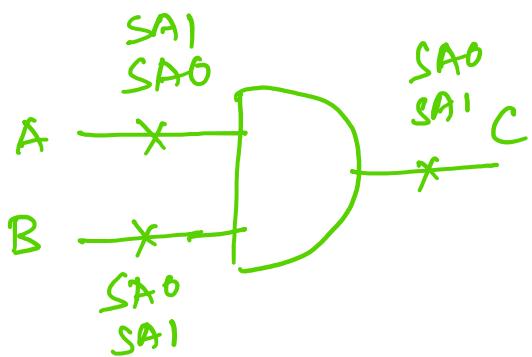
---

- Number of fault sites in a Boolean gate circuit =  $\# \text{PI} + \# \text{gates} + \# (\text{fanout branches})$ .
- **Fault equivalence:** Two faults  $f_1$  and  $f_2$  are equivalent if all tests that detect  $f_1$  also detect  $f_2$ .
- If faults  $f_1$  and  $f_2$  are equivalent then the corresponding faulty functions are identical.
- **Fault collapsing:** All single faults of a logic circuit can be divided into disjoint equivalence subsets, where all faults in a subset are mutually equivalent. A collapsed fault set contains one fault from each equivalence subset.

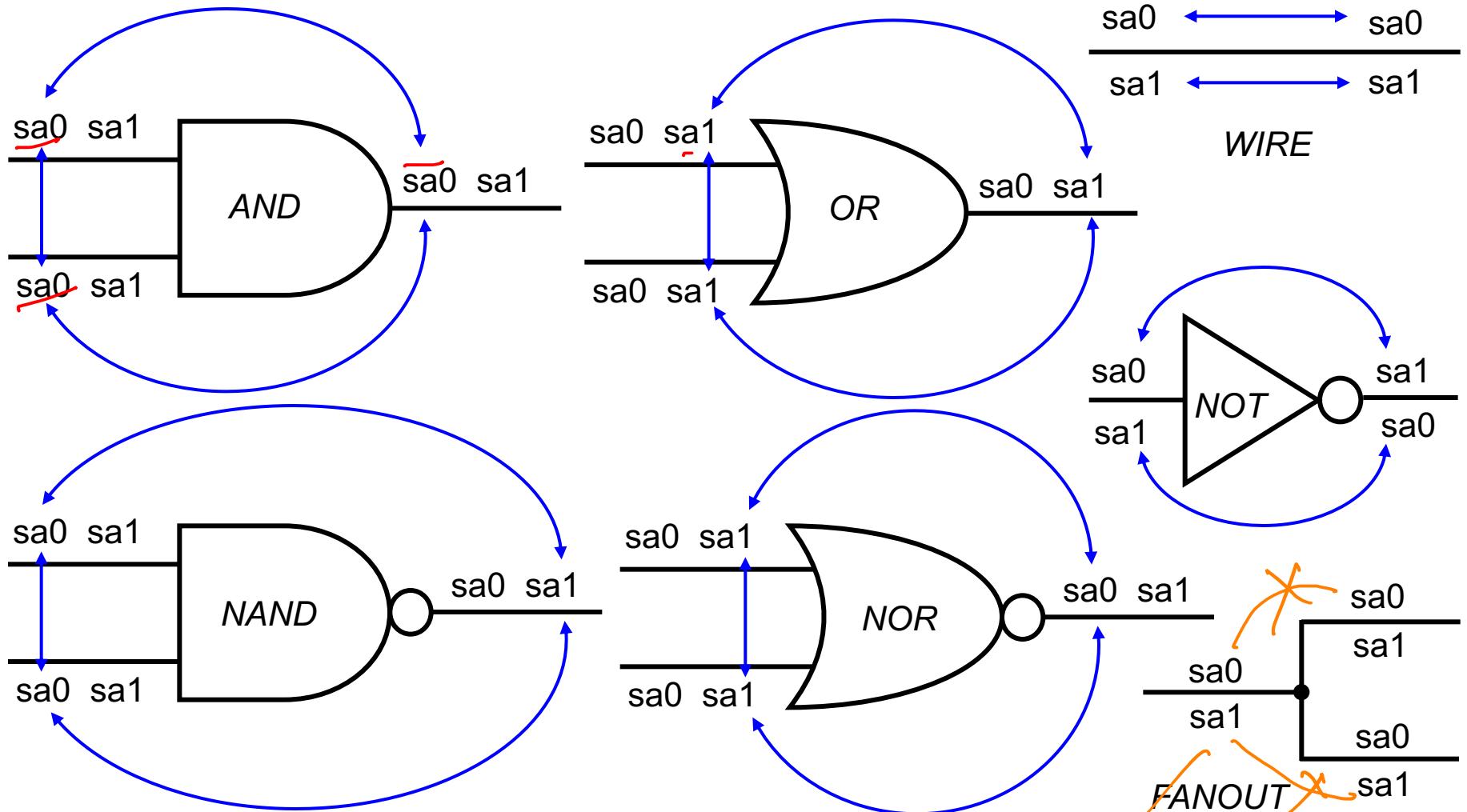
$\{f_1, f_2\}$ ,  $\{f_3, f_5, f_6\}$ ,  $\{f_4, f_8\}$ ,  $f_7$

target FL =  $\{f_1, f_3, f_4, f_7\}$

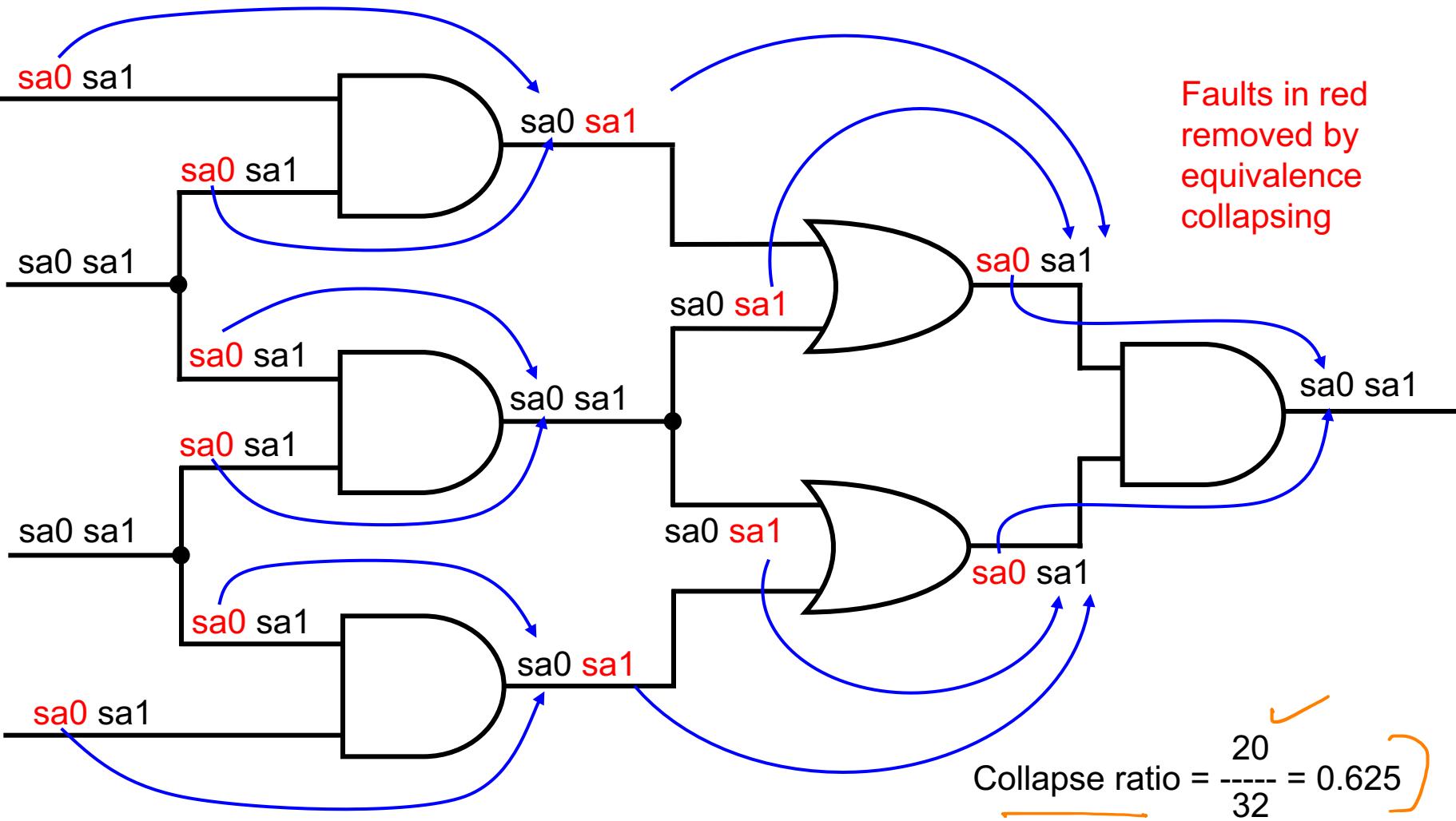


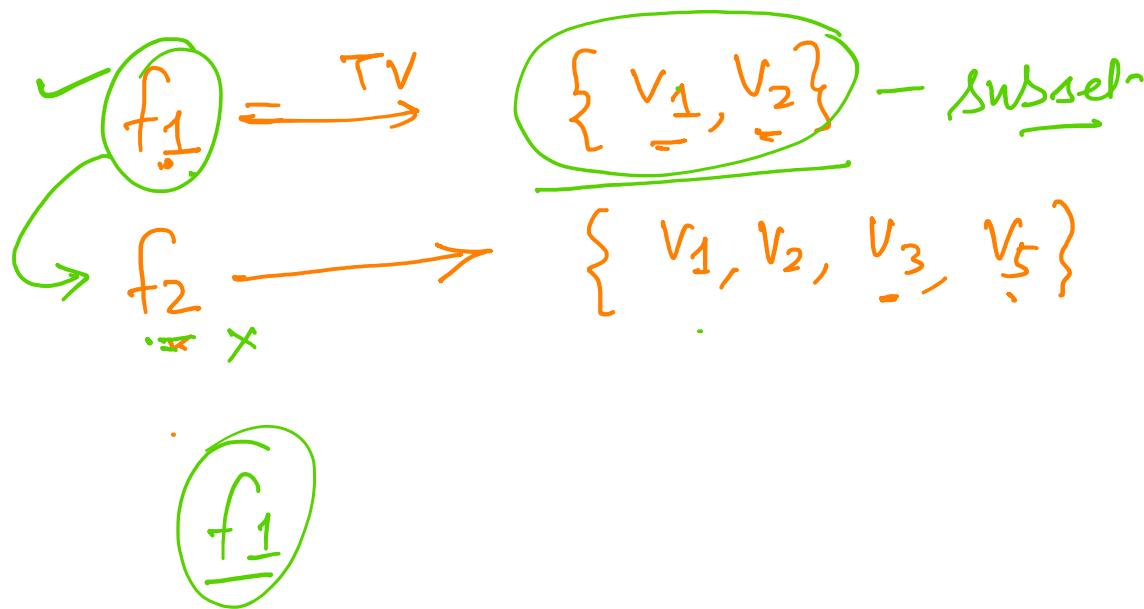


# Equivalence Rules



# Equivalence Example





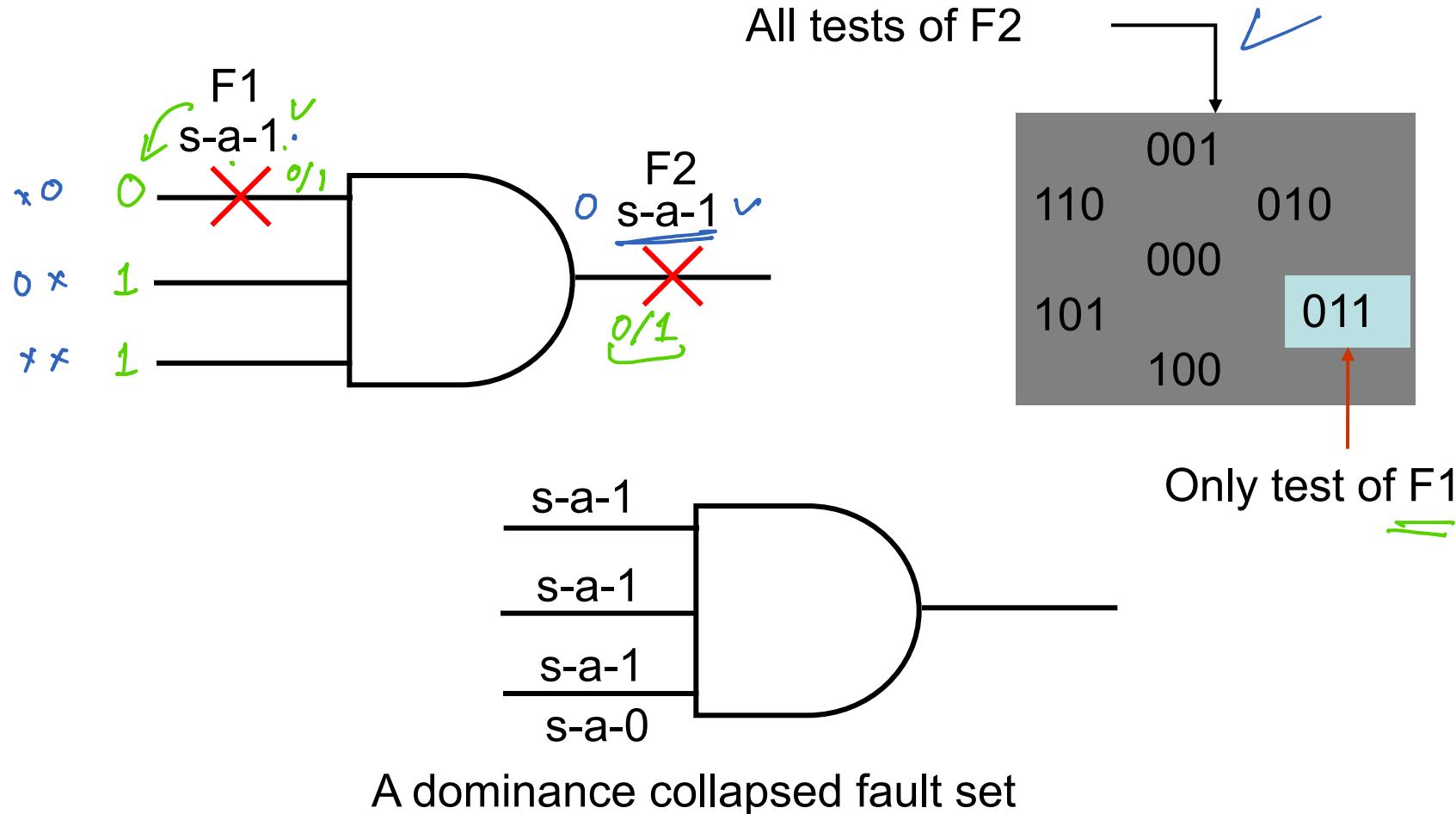
# Fault Dominance ✓

---

- ❖ If all tests of some fault F1 detect another fault F2, then F2 is said to dominate F1.
- ❖ Dominance fault collapsing: If fault F2 dominates F1, then F2 is removed from the fault list.
- ❖ When dominance fault collapsing is used, it is sufficient to consider only the input faults of Boolean gates.
- ❖ In a tree circuit (without fanouts) PI faults form a dominance collapsed fault set.
- ❖ If two faults dominate each other then they are equivalent.



# Dominance Example



# Test Generation

---

By using  $\underline{FE} \geq \underline{FD}$ ] reduce the no. of faults

take one by one fault &  
generate the test.

Minimize  $\sum V$ .



# Thank You



# Logic Testing: Test Generation

---

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

<http://www.ee.iitb.ac.in/~viren/>

E-mail: [viren@ee.iitb.ac.in](mailto:viren@ee.iitb.ac.in)

*CS-226: Digital Logic Design*

---



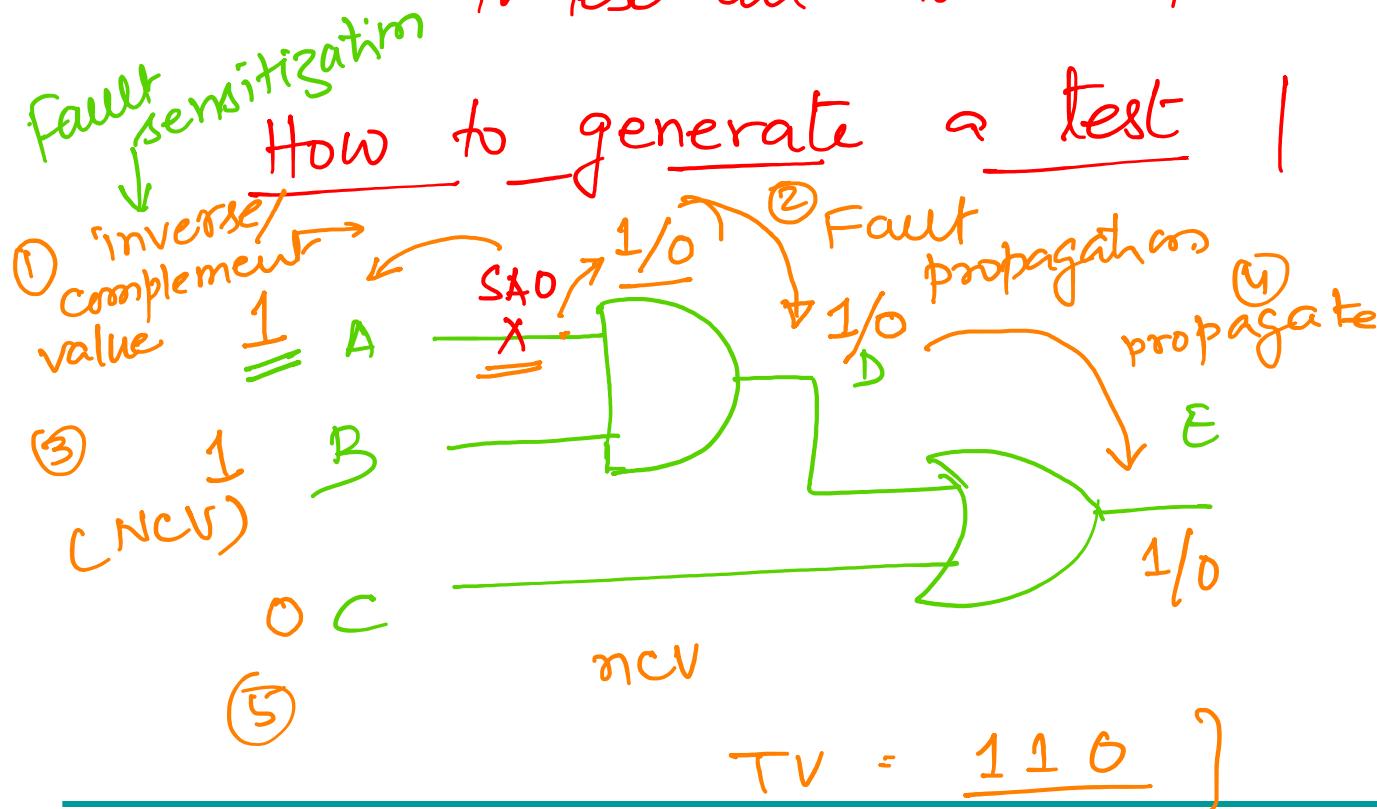
Lecture 32-C: 20 April 2021

**CADSL**

# Test Generation

## Objective

# Find the minimum no. of test vectors  
to test all modeled faults of a circuit

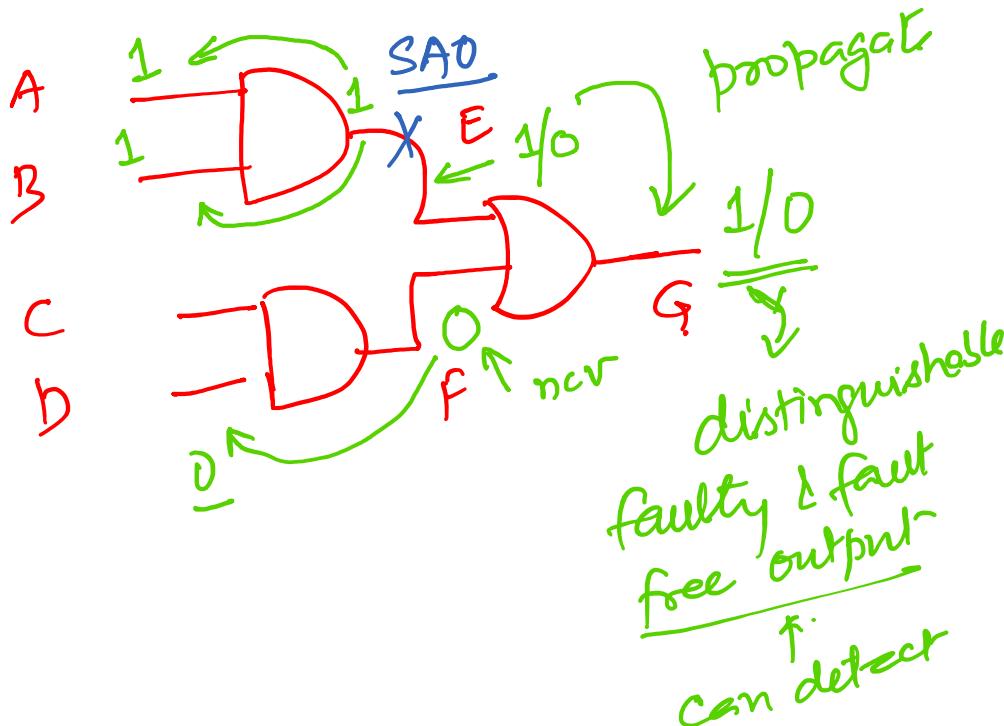


# Test Generation

- ① Define the fault location (net & type of fault)
- ② Activate/Sensitize the fault (having complementary value. jhor before fault site)
- ③ Propagate the fault effect towards PO (primary output)
- ④ need new assignment at the gates which are in propagation paths
- ⑤ If fault effect reaches to PO, - done  
Input Assignment is Test Vector (TV)



# Test Generation



$A = 1 \quad B = 1$   
Sensitize the fault  
Propagation  
n/w at F  
 $F = 0$

$$\text{IV} = \underline{\underline{1 \ 1 \ X \ 0}}$$
$$\left[ \begin{array}{c} 1110 \\ 1100 \end{array} \right]$$

- ① ALGORITHMIC ✓
- ② ALGEBRAIC ✓



# Boolean Difference

$$F(x_1, x_2 \dots x_i \dots x_n) = \underbrace{x_i f_{xi} \oplus \bar{x}_i f_{\bar{xi}}}$$

- Shannon's Expansion Theorem:

$$F(X_1, X_2, \dots, X_n) = X_2 \quad F(X_1, 1, \dots, X_n) + X_2 \quad F(\overline{X_1}, 0, \dots, X_n)$$

- Boolean Difference (partial derivative):

$$\frac{\partial F_j}{\partial g} = F_j(1, x_1, x_2, \dots, x_n) \oplus F_j(0, x_1, \dots, x_n)$$

Boolean Difference

$$\frac{\partial f}{\partial x_i} = \underbrace{f_{xi} \oplus f_{\bar{xi}}}$$

- Fault Detection Requirements:

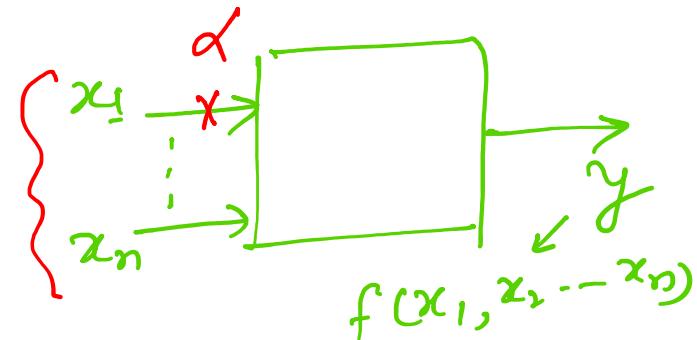
$$G(X_1, X_2, \dots, X_n) = 1$$

$$\frac{\partial F_j}{\partial g} = F_j(1, x_1, x_2, \dots, x_n) \oplus F_j(0, x_1, \dots, x_n) = 1$$



# Test Generation

An input assignment can be a test vector if it produces distinguishable faulty & fault free response.



$$y = f(x_1, x_2, \dots, x_i, \dots, x_n)$$

$$\boxed{f = \underline{x_i} f_{xi} \oplus \overline{x_i} f_{\bar{x}_i}} \quad (\text{Shannon's expansion})$$

Assume there is a fault  $\delta$  at  $\underline{x_i}$

faulty output  $f_\delta$

$$\begin{aligned} \text{Fault - } \underline{x_i} \xrightarrow{\text{SAO}} f_\delta &= \underline{f}_{\bar{x}_i} \checkmark \\ \underline{x_i} \xrightarrow{\text{SAI}} f_\delta &= f_{xi} \checkmark \end{aligned}$$



# Test Generation

$$f(x) \oplus f_{\bar{x}}(x) = 1$$

*distinguishable*

$f(x) \rightarrow f_{\bar{x}}(x)$

$x = \underline{x_1, x_2, x_3, \dots, x_n}$

if  $x$  is a Test Vector

$$\underset{x_i}{x_i} f(x) \oplus \underset{\bar{x}_i}{\bar{x}_i} f_{\bar{x}_i}(x) \oplus \underset{=}{f_{\bar{x}}(x)} = 1$$

Let  $x_i \leq 0$ ,  $f_{\bar{x}}(x) = f_{\bar{x}_i}(x)$

$$\underset{x_i}{x_i} f_{\bar{x}_i} \oplus \underset{\bar{x}_i}{\bar{x}_i} f_{\bar{x}_i} \oplus f_{\bar{x}_i} = 1$$

$$\underset{x_i}{x_i} f_{\bar{x}_i} \oplus \underset{\bar{x}_i}{\bar{x}_i} f_{\bar{x}_i} \oplus (x_i \oplus \bar{x}_i) \cdot f_{\bar{x}_i} = 1$$

$(A \oplus A = 0)$

$$\underset{x_i}{x_i} f_{\bar{x}_i} \oplus \underset{\bar{x}_i}{\bar{x}_i} f_{\bar{x}_i} \oplus \underset{x_i}{x_i} f_{\bar{x}_i} \oplus \underset{\bar{x}_i}{\bar{x}_i} f_{\bar{x}_i} = 1$$



# Test Generation

$$x_i f_{ni} \oplus x_i f_{\bar{n}} = 1$$

$$x_i (f_{ni} \oplus f_{\bar{n}}) = 1$$

$$\boxed{x_i \cdot \frac{\partial f}{\partial x_i} = 1}$$

$$\checkmark x_i = 1 \leftarrow$$

Fault sensitization

$$\boxed{x_i = 0}$$

$$\frac{\partial f}{\partial x_i} = \boxed{f_{ni} \oplus f_{\bar{n}}} = 1$$

Fault propagation condition



# Test Generation

$$x_i \leq 1 \quad f_x = f_{xi}$$

$$f(x) \oplus f_x(x) = 1$$

$$f(x) \oplus f_{\bar{x}_i}(x) = 1$$

$$x_i \cdot f_{xi} \oplus \bar{x}_i \cdot f_{\bar{x}_i} \oplus f_{\bar{x}_i} = 1$$

$$x_i \cdot f_{xi} \oplus \bar{x}_i \cdot f_{\bar{x}_i} \oplus (x_i \oplus \bar{x}_i) \cdot f_{\bar{x}_i} = 1$$

$$\cancel{x_i \cdot f_{xi}} \oplus \bar{x}_i \cdot f_{\bar{x}_i} \oplus \cancel{x_i \cdot f_{xi}} \oplus \bar{x}_i \cdot f_{\bar{x}_i} = 1 \quad \checkmark$$

$$\bar{x}_i \cdot f_{\bar{x}_i} \oplus \bar{x}_i \cdot f_{xi} = 1 \Rightarrow \bar{x}_i (f_{xi} \oplus f_{\bar{x}_i}) = 1$$

$$\bar{x}_i = 1 \Rightarrow x_i = 0 \rightarrow \text{Fault sensitization}$$

$$? \quad \frac{\partial f}{\partial x_i} = f_{xi} \oplus f_{\bar{x}_i} = 1 \rightarrow \text{Fault propagation.}$$

$$x = x_1 \cdot x_2 \cdot x_i \cdots x_n \rightarrow \checkmark$$



# Boolean Difference

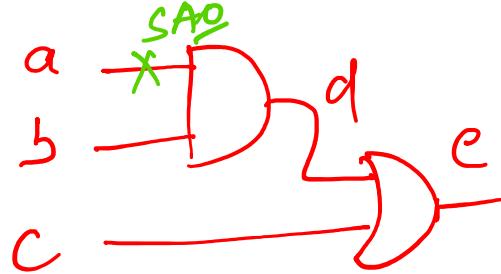
$$f(x_1, \dots, x_i = 0, \dots, x_n) \oplus f(x_1, \dots, x_i = 1, \dots, x_n) = 1$$

- Represented by the symbol  $\underline{df(x)/dx}$
- $df(x)/dx_i$  for  $x=0$  and  $df(x)/dx_i$  for  $x=1$  are called *the residues/co-factors* of the function for  $x = x_i$
- One of the residue is the good-circuit value and the other is the faulty-circuit value for  $x_i$
- To detect the fault, the two residues should be complementary
- Solving the equation yield the values of the primary inputs to detect a stuck-at fault on  $x_i$
- The test pattern is:  $x_i df(x)/dx_i = 1$  &  $x_i' df(x)/dx_i = 1$

Fault propagation



# Test Generation



$$e = f(a,b,c) = \underline{\overline{ab} + c}$$

$\text{TV}$

$$a \cdot \frac{\partial f}{\partial a} = 1$$

$$a=1 \text{ and } \underline{f_a \oplus f_{\bar{a}} = 1}$$

$$f_a = b+c$$

$$f_{\bar{a}} = c$$

$$a \cdot ((b+c) \oplus c) = 1$$

$$a \cdot ((b+c)\bar{c} + (\overline{b+c}) \cdot c) = 1$$

$$\Rightarrow a \cdot (b\bar{c} + 0 + \cancel{b \cdot \bar{c} \cdot c}) = 1 \Rightarrow \boxed{a \cdot b \cdot \bar{c} = 1}$$

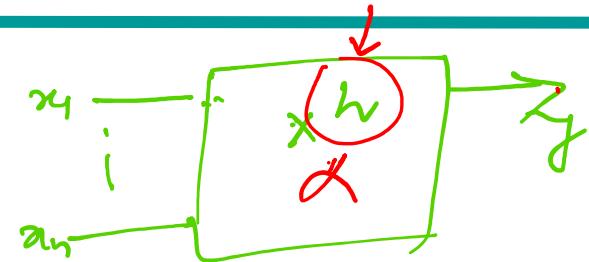
$$\left. \begin{array}{l} a=1 \\ b=1 \\ \bar{c}=1 \end{array} \right\} \left. \begin{array}{l} a=1 \\ b=1 \\ c=0 \end{array} \right\} \text{TV}$$



# Test Generation

$$f(x) = f(x_1, x_2, \dots, x_i, \dots, x_n, h) \rightarrow$$

$f_d \rightarrow$  faulted ~~or~~ function  
(fault at  $h$ )



$$\underline{h} = f(\underline{x}, \underline{x} \dots, \underline{x_n})$$

$$\underline{f(x,h)} \oplus \underline{f_d(x,h)} = 1$$

$$\text{for cert } h \text{ is } \underline{\text{so}} \quad f_d(x,h) = \underline{f_h(x,h)}$$

$$\underline{f(x,h)} \oplus \underline{f_h(x,h)} = 1$$

$$h f_h \oplus \bar{h} f_{\bar{h}} \oplus f_{\bar{h}} = 1$$

$$h f_h \oplus \bar{h} f_{\bar{h}} \oplus (h \oplus \bar{h}) f_{\bar{h}} = 1$$

$$h f_h \oplus \cancel{\bar{h} f_{\bar{h}}} \oplus \cancel{h f_{\bar{h}}} \oplus \cancel{\bar{h} f_h} = 1 \Rightarrow \boxed{h(f_h \oplus f_{\bar{h}}) = 1}$$

$$\begin{aligned} h &= 1 \checkmark \\ \frac{\partial f}{\partial h} &= 1 \checkmark \\ \uparrow & \end{aligned}$$



# Test Generation

$\lambda \text{ sao}$

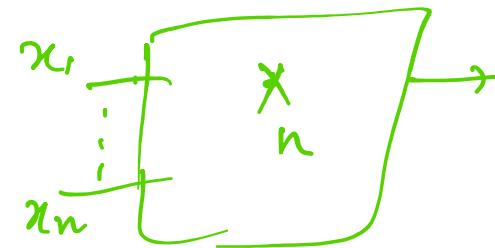
$$\lambda \cdot (f_h \oplus f_{\bar{h}}) = 1$$

if  $\lambda \text{ is sat}$   $f_x = f_h$

then

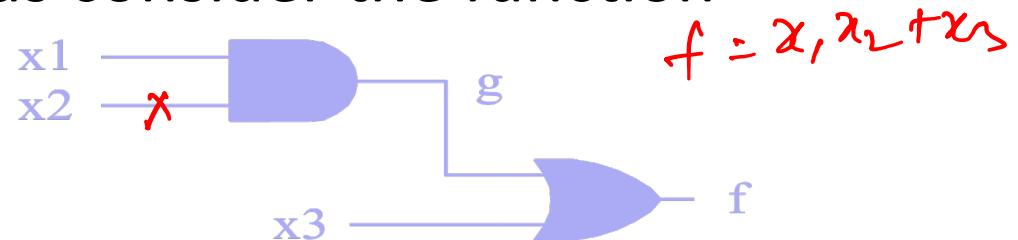
$$\bar{\lambda} (f_h \oplus f_{\bar{h}}) = \bar{\lambda} \frac{\partial f}{\partial h} = 1$$

$$\boxed{\begin{aligned}\bar{\lambda} &= 1 \\ \frac{\partial f}{\partial h} &= 1\end{aligned}}$$



# Fault Detection

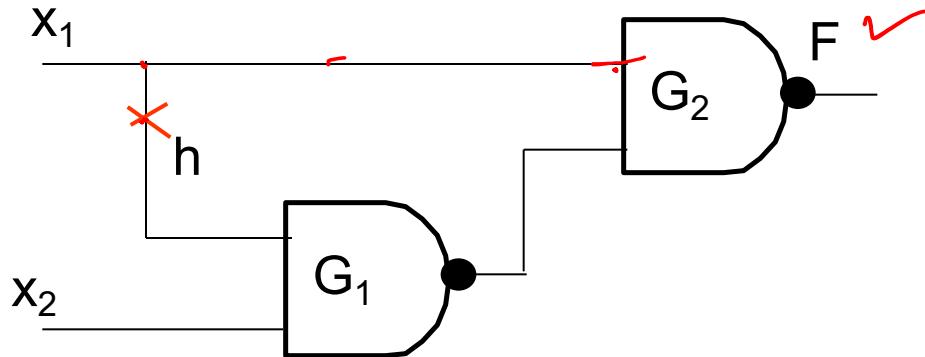
- ❖  $x_i, df(x)/dx_i = 1$  for s-a-0 at  $x_i$
- ❖  $x_i', df(x)/dx_i = 1$  for s-a-1 at  $x_i$
- ❖ As an example, let us consider the function



- ❖  $f(x) = x_1 x_2 + x_3$  ✓
- ❖ Thus  $\underline{df(x)/dx_2} = \underline{x_3} \oplus (\underline{x_1} + \underline{x_3}) = \underline{x_3}' \underline{x_1} = 1$ . Then
- ❖  $x_1 = 1$  and  $x_3 = 0$ .
- ❖ For the SA1 and SA0 faults on  $x_2$ , the patterns are then  $x_1 x_2 x_3 = (\underline{1}00)$  and  $(1\underline{1}0)$ , respectively.

# Fault Detection

$$h(x) \cdot \frac{\partial f}{\partial h} = 1$$



## S-a-0 fault at $h =$

# Test Vector

$$h(X) \frac{dF^*(X,h)}{dh} = 1$$

$$\underline{x_1} \cdot x_1 x_2 = x_1 \underline{x_2} = 1$$

$$\left. \begin{array}{l} x_1 = 1 \\ x_2 = 1 \end{array} \right\} \quad \text{Triv}$$

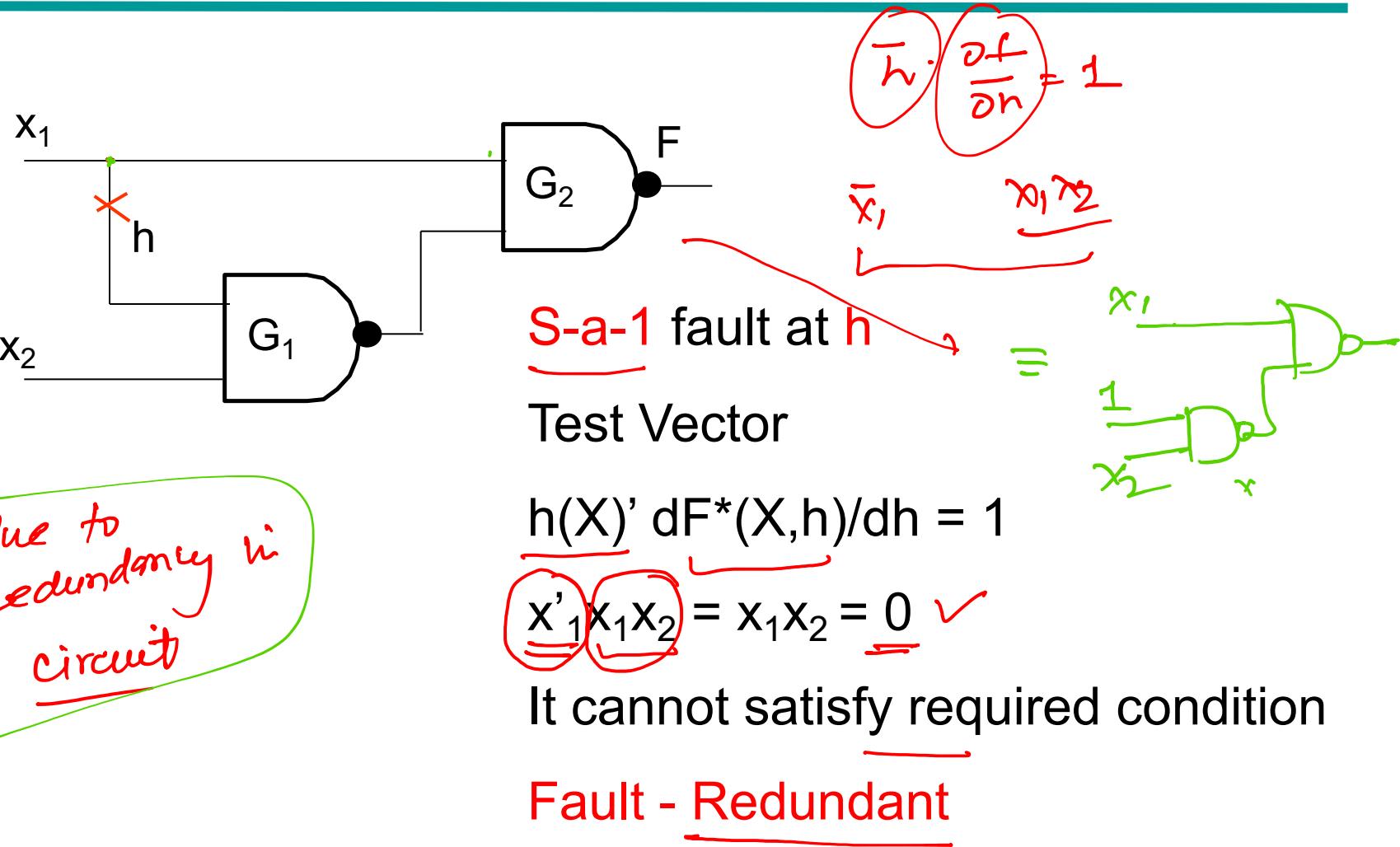
$$F(X, h) = x_1' + h \cdot x_2$$

$$h(X) = x_1 \bullet$$

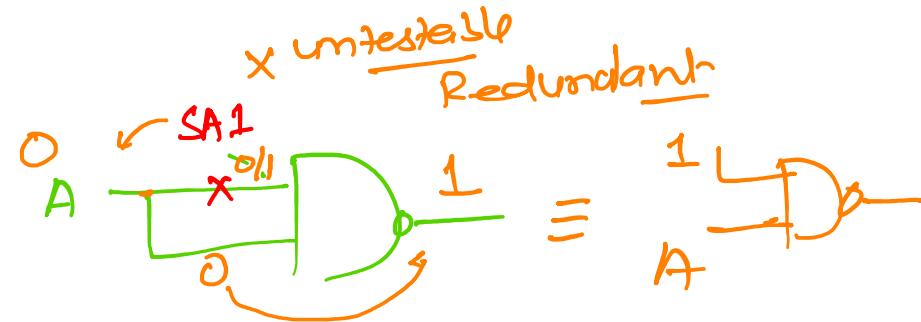
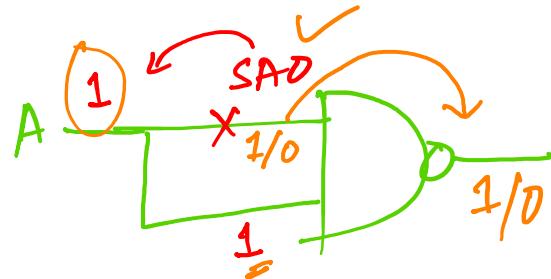
$$\begin{aligned} \underline{dF^*(X,h)/dh} &= \underline{x'_1} \bigoplus (\underline{x'_1} + \underline{x_2}) \\ &= x_1 x_2 \end{aligned}$$



# Fault Detection



-Do



## Redundant Faults

$$FC = \text{Fault Coverage} =$$

$$\frac{\# \text{ detectable faults}}{\# \text{ faults}}$$

$$\text{Fault Efficiency} = \frac{\# \text{ detectable faults}}{\# \text{ faults} - \# \text{ Redundant faults}}$$

faults: 1000

990 - detectable

10 Redundant

✓

$$FC = \frac{990 \times 100}{1000}$$

$$= 99\%$$

$$FE = \frac{990}{1000-10} = 100\%$$



Boolean Difference  $\rightarrow$  TV or Redundant  
— Test Efficiency = 100 %  
— Fault

Algorithmic method — Backtracking

# Test vectors must be small!

TE/PE  $\approx 100\%$

Reconvergent fanout.

Fanouts free  
circuit.



# Thank You

