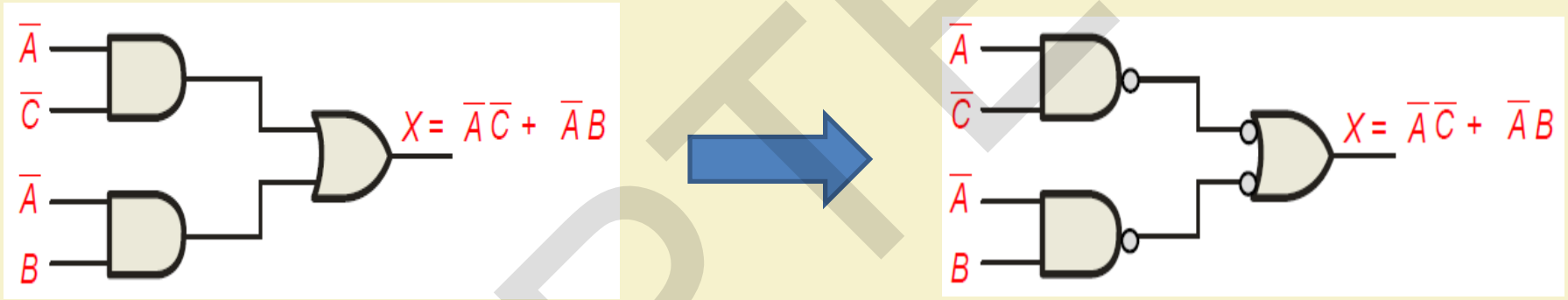
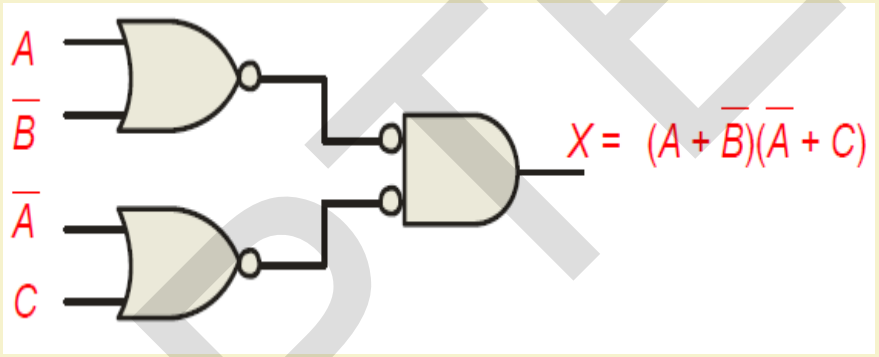
Conversion to NAND



Conversion to NOR



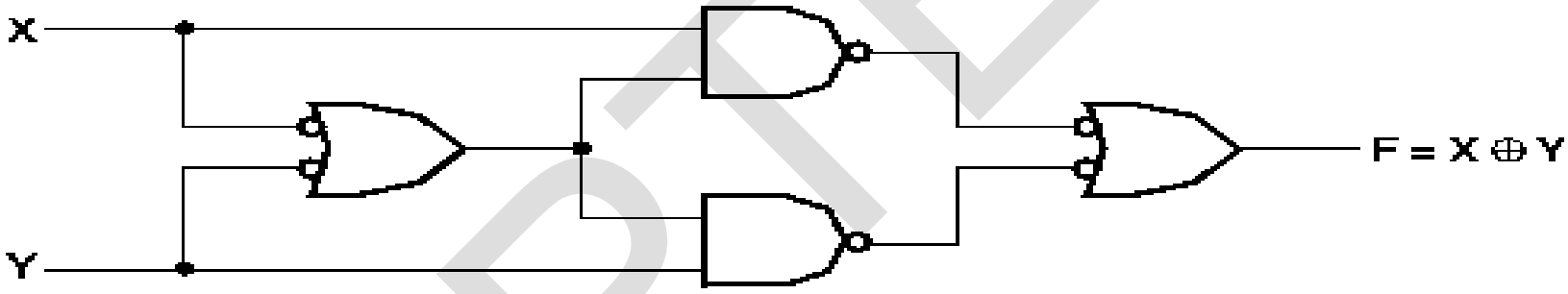
Exclusive-OR (XOR) Function

- XOR (also $\oplus$) : the "not-equal" function
- XOR(X,Y) = $X \oplus Y = X'Y + XY'$
- Identities:
  - $X \oplus 0 = X$
  - $X \oplus 1 = X'$
  - $X \oplus X = 0$
  - $X \oplus X' = 1$
- Properties:
  - $X \oplus Y = Y \oplus X$
  - $(X \oplus Y) \oplus W = X \oplus (Y \oplus W)$

XOR function implementation

- XOR(a,b) = ab’ + a’b
- Straightforward: 5 gates
  - 2 inverters, two 2-input ANDs, one 2-input OR
  - 2 inverters & 3 2-input NANDs
- Nonstraightforward:
  - 4 NAND gates

XOR circuit with 4 NANDs



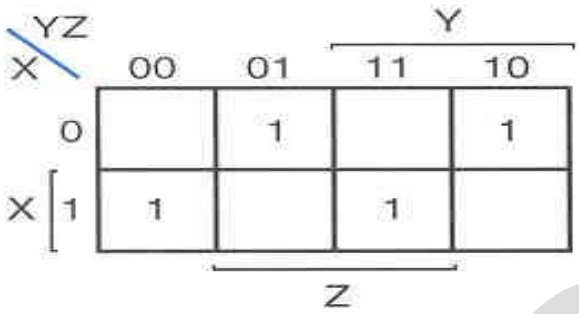
Exclusive-NOR (XNOR) Function

- XNOR: the "equality" function
- XNOR(a,b) = ab + a'b'
- Observe that XNOR(a,b) = (XOR(a,b))'
  - (a $\oplus$ b)' = (a'b + ab')'
    = (a'b)' (ab')'
    = (a + b') (a' +b)
    = ab + a'b'
- a $\oplus$ b' = (a $\oplus$ b)' = a' $\oplus$ b

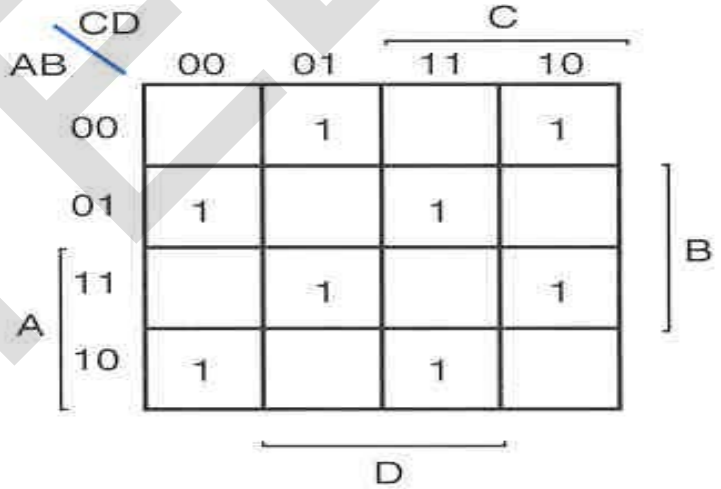
Odd Function

- $x \oplus y = x'y + xy'$
- $x \oplus y \oplus z = xy'z' + x'yz' + x'y'z + xyz$
- $x \oplus y \oplus z \oplus w = x'yzw + xy'zw + xyz'w + xyzw' + x'y'z'w + x'yz'w' + x'y'zw' + xy'z'w'$
- … Observe a pattern here?
- An n-input XOR function is implied (=1) by all the minterms that have an odd # of 1s
- Thus, XOR is also know as the ***odd function***

Odd Function (cont.)



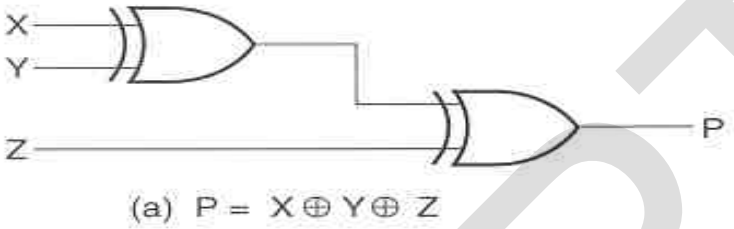
(a) $X \oplus Y \oplus Z$



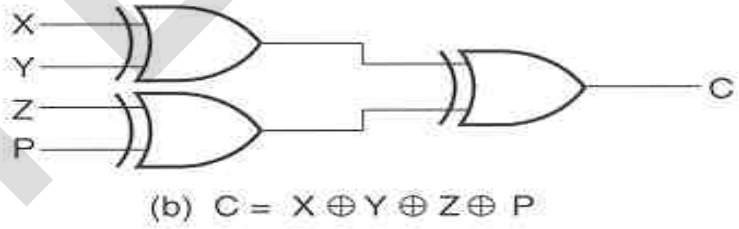
(b) $A \oplus B \oplus C \oplus D$

Minterms are ALWAYS ***distance two*** from each other

Odd Function (cont.)



(a) $P = X \oplus Y \oplus Z$



(b) $C = X \oplus Y \oplus Z \oplus P$

Even Function

- How would you implement an even ***function***?

The complement of XOR → XNOR

Parity Generation and Checking

- Odd and even functions can be used to implement parity checking circuits used for error detection and correction.
- Use even parity as example.
- *Parity generator:* the circuit that generates the parity bit before transmitting.
- *Parity checker:* the circuit that checks the parity in the receiver.

Even Parity Generation

Three-Bit Message			Parity Bit
X	Y	Z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- P(X,Y,Z) must produce a 1 for all the input combinations that contain an odd number of 1s
- Thus, it is a 3-input odd function $P = X \oplus Y \oplus Z$

Even Parity Checking

How would you implement a parity checker for the previous example?

Use a 4-input XOR circuit (odd function)

$C = X \oplus Y \oplus Z \oplus P$ → 1 indicates an error

OR

A 4-input XNOR circuit (even function)

$C = (X \oplus Y \oplus Z \oplus P)'$ → 1 indicates a pass

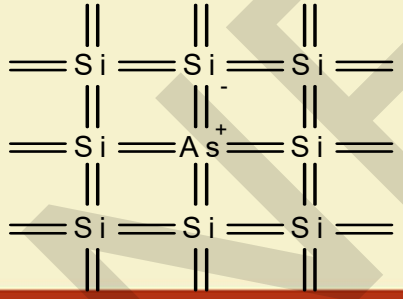
Logic Families

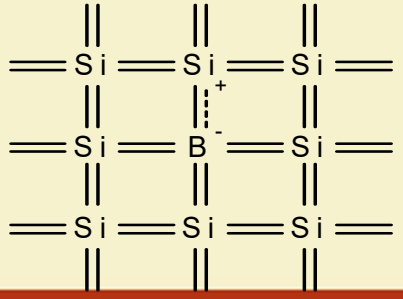
- DTL
- TTL
- ECL
- NMOS
- PMOS
- CMOS
- Etc.

Static CMOS Logic Family

Dopants

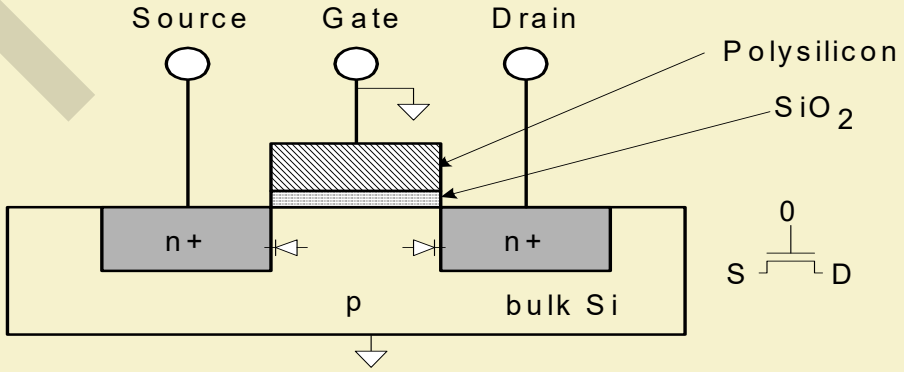
- Silicon is a semiconductor
- Pure silicon has no free carriers and conducts poorly
- Adding dopants increases the conductivity
- Group V: extra electron (n-type)
- Group III: missing electron, called hole (p-type)





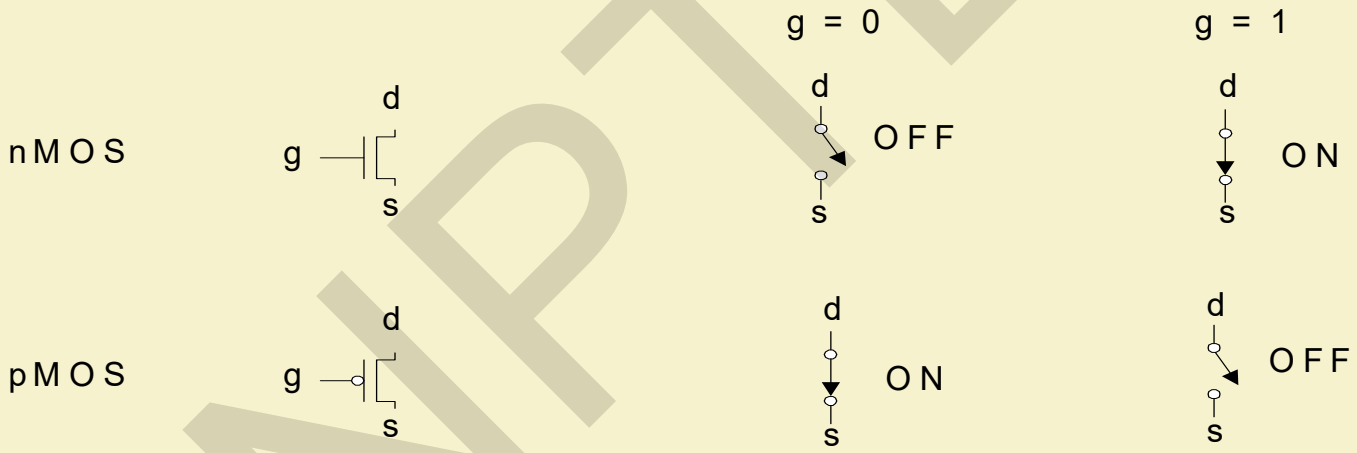
nMOS Operation

- Body is commonly tied to ground (0 V)
- When the gate is at a low voltage:
  - P-type body is at low voltage
  - Source-body and drain-body diodes are OFF
  - No current flows, transistor is OFF



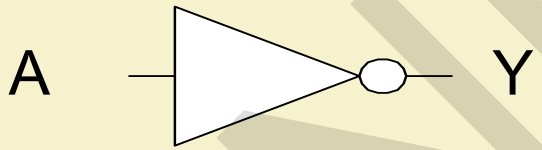
Transistors as Switches

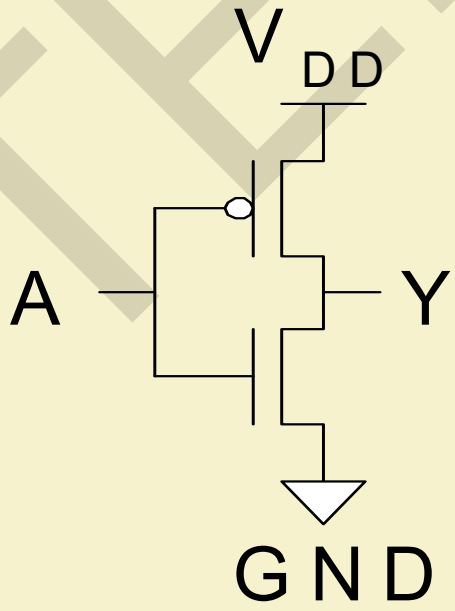
- We can view MOS transistors as electrically controlled switches
- Voltage at gate controls path from source to drain



CMOS Inverter

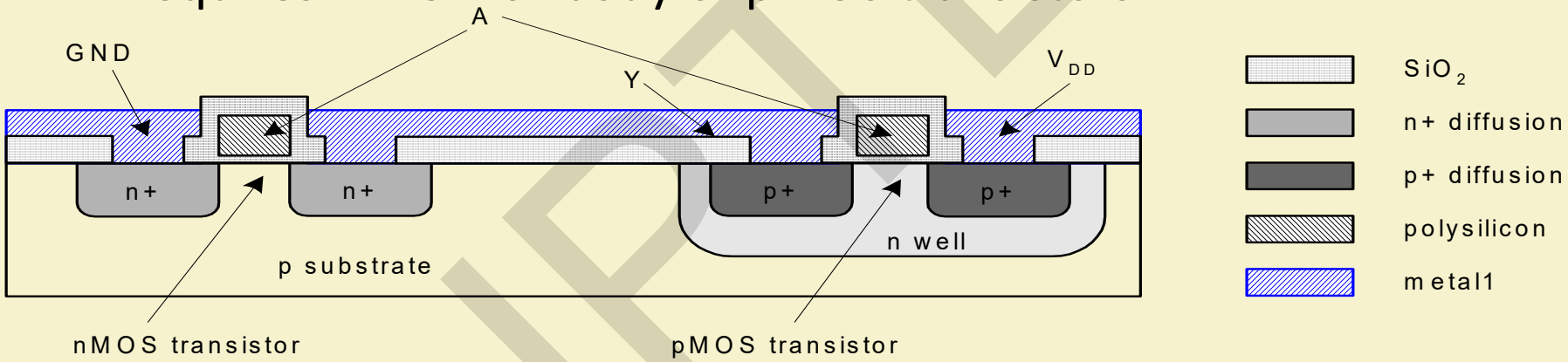
A	Y
0	
1	



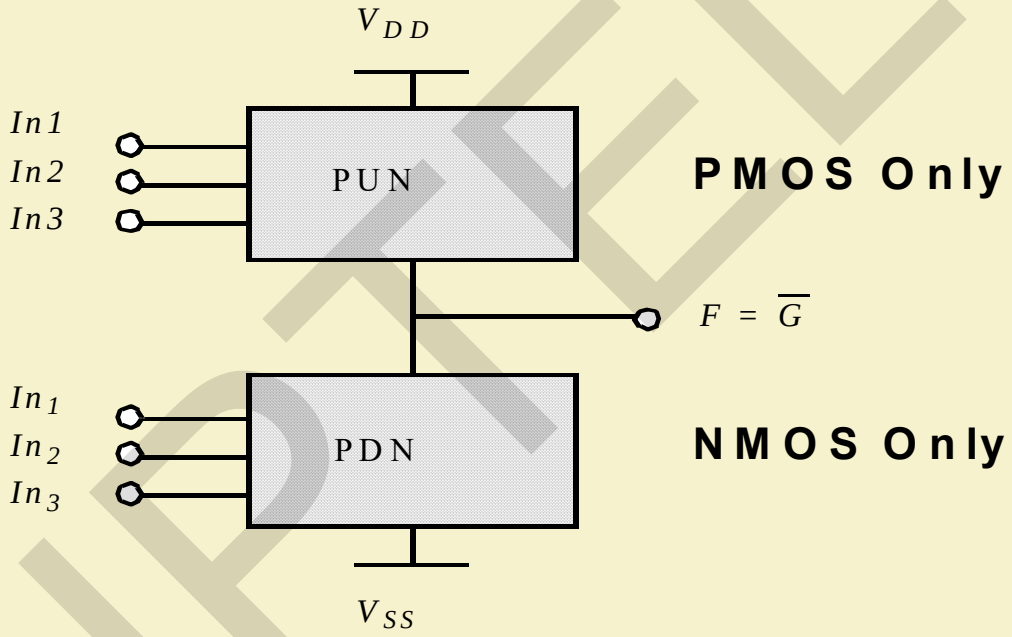


Inverter Cross-section

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



Static CMOS



PUN and PDN are Dual Networks

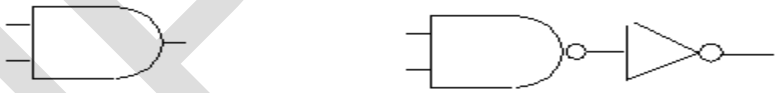
Complementary CMOS Logic Style Construction

- **PUP is the <u>DUAL</u> of PDN**
  (can be shown using DeMorgan's Theorem's)

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

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

- **The complementary gate is inverting**

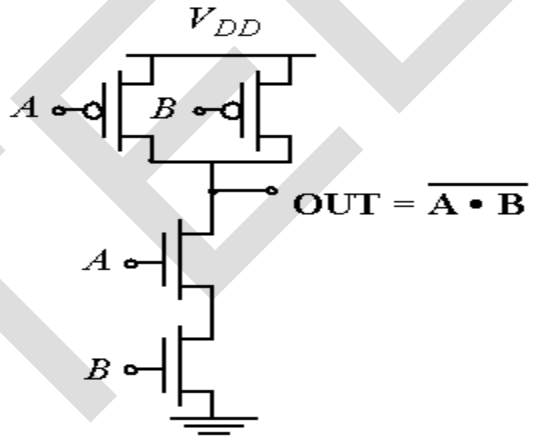


AND = NAND + INV

Example Gate: NAND

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Truth Table of a 2 input NAND gate



PDN: $G = A\,B \Rightarrow$ Conduction to GND

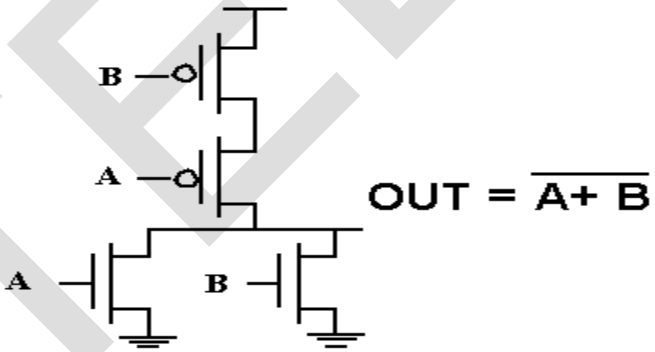
PUN: $F = \overline{A} + \overline{B} = \overline{AB} \Rightarrow$ Conduction to V_{DD}

$$\overline{G(In_1, In_2, In_3, \ldots)} \equiv F(\overline{In_1}, \overline{In_2}, \overline{In_3}, \ldots)$$

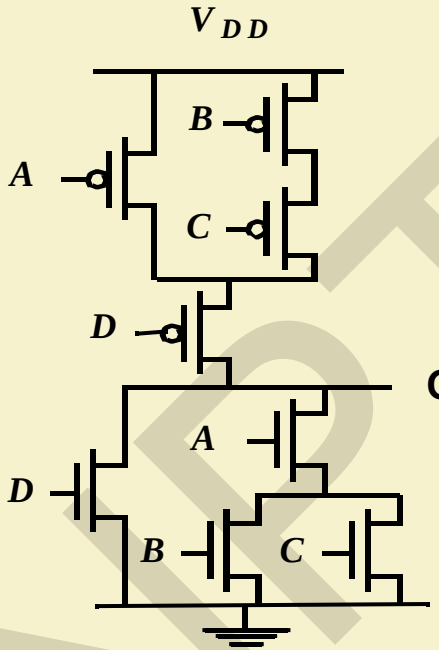
Example Gate: NOR

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Truth Table of a 2 input NOR gate



Example Gate: COMPLEX CMOS GATE



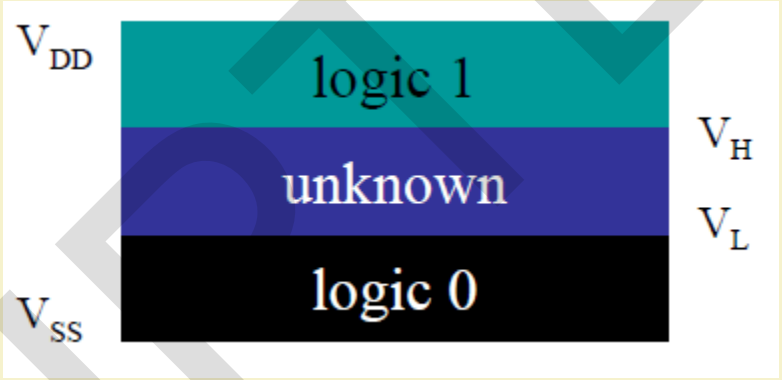
$$OUT = \overline{D + A \cdot (B + C)}$$

Properties of Complementary CMOS Gates

- High noise margin. V_{OH} and V_{OL} are at V_{DD} and GND respectively
- No static power consumption
- Comparable rise and fall times
- Highly compact structure

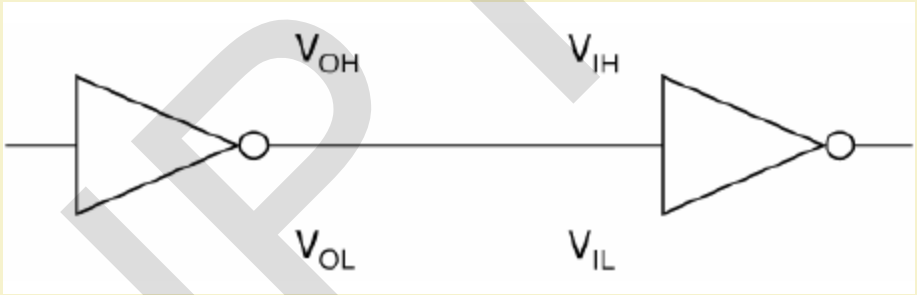
Timing diagrams and Hazards

Noise Margins



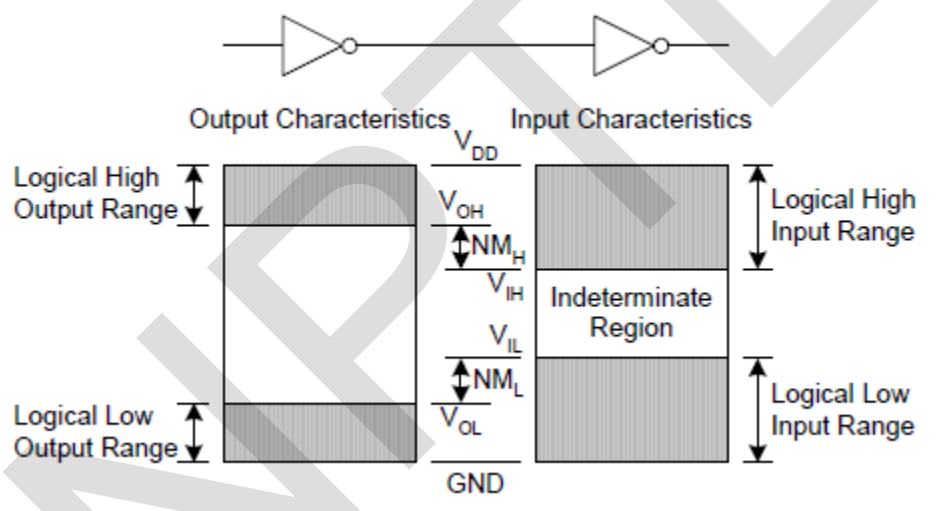
Logic Level Matching

- Levels at output of one gate must be sufficient to drive another gate



Noise Margins

- How much noise a gate input see before it does not recognize the input?

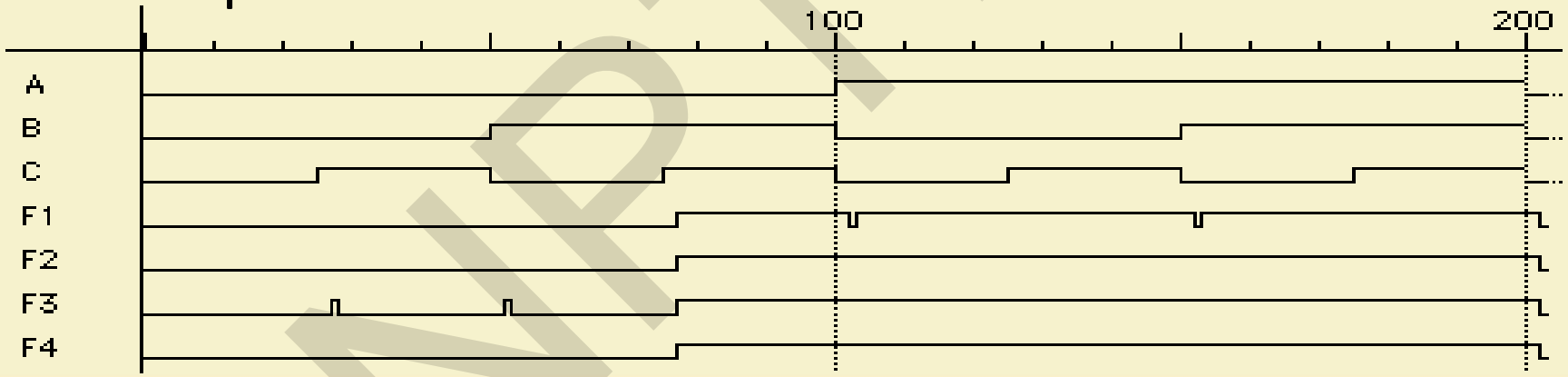


Noise Margin

- Noise Margin = Voltage difference between the output of one gate and input of next. Noise must exceed noise margin to make the second gate produce wrong output.

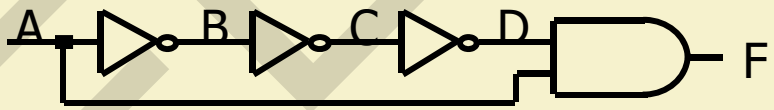
Timing diagrams (waveforms)

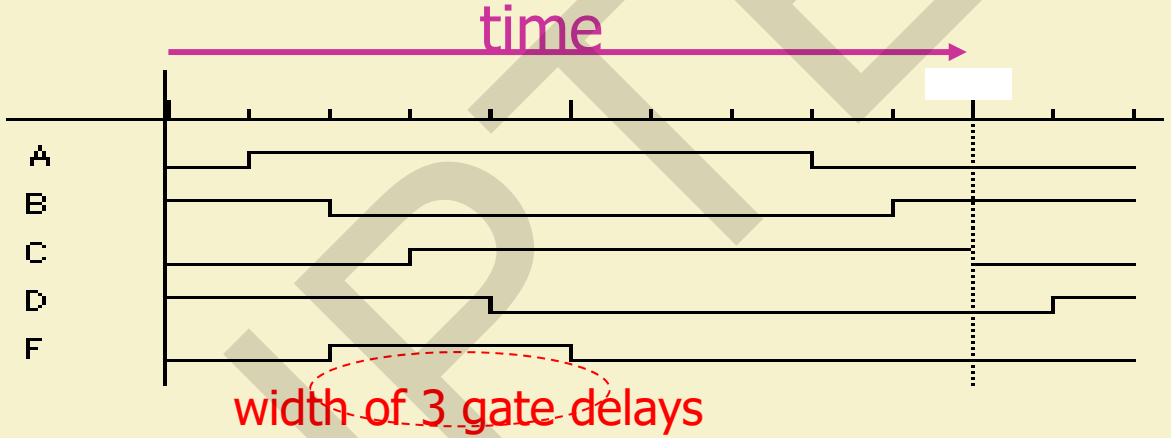
- Shows time-response of circuits
- Like a sideways truth table
- Example: F = A + BC



Timing diagrams

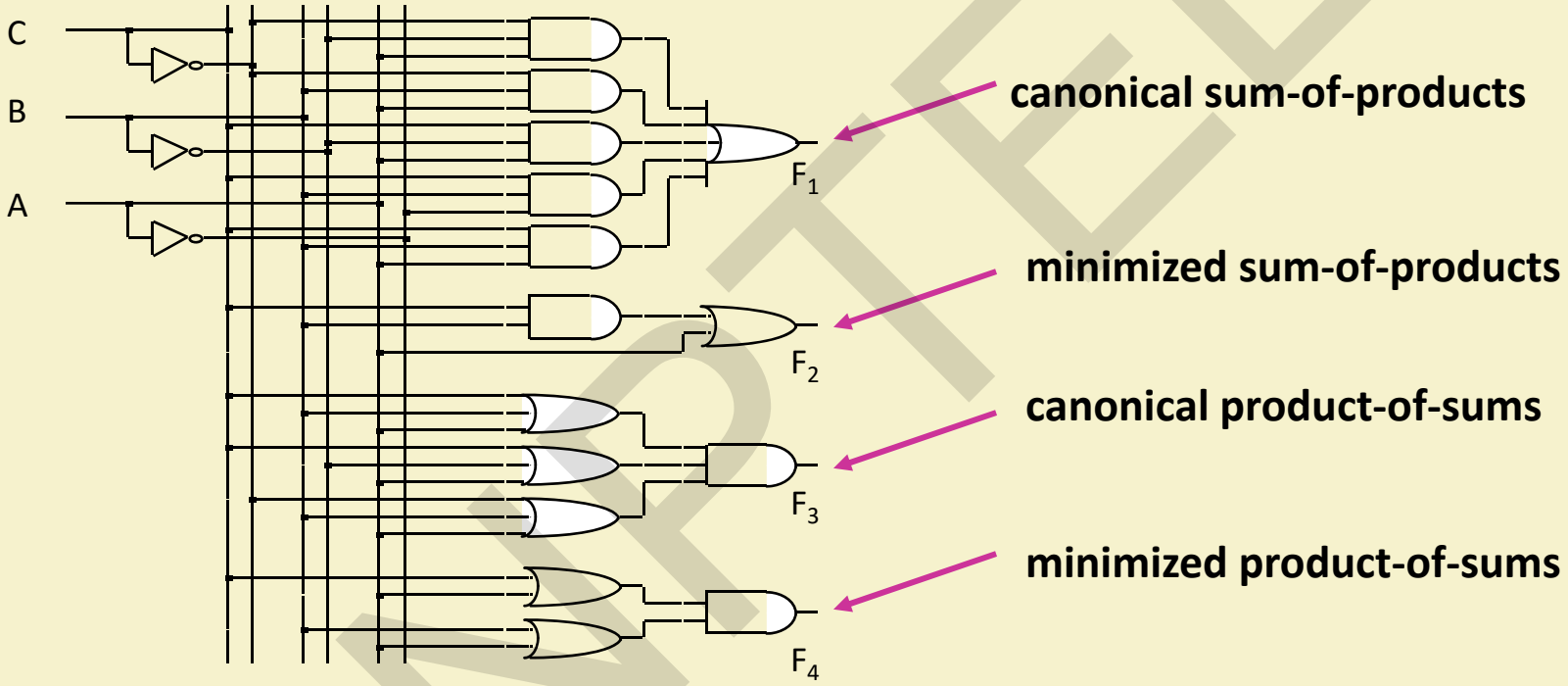
- Real gates have real delays
- Example: A' . A = 0?





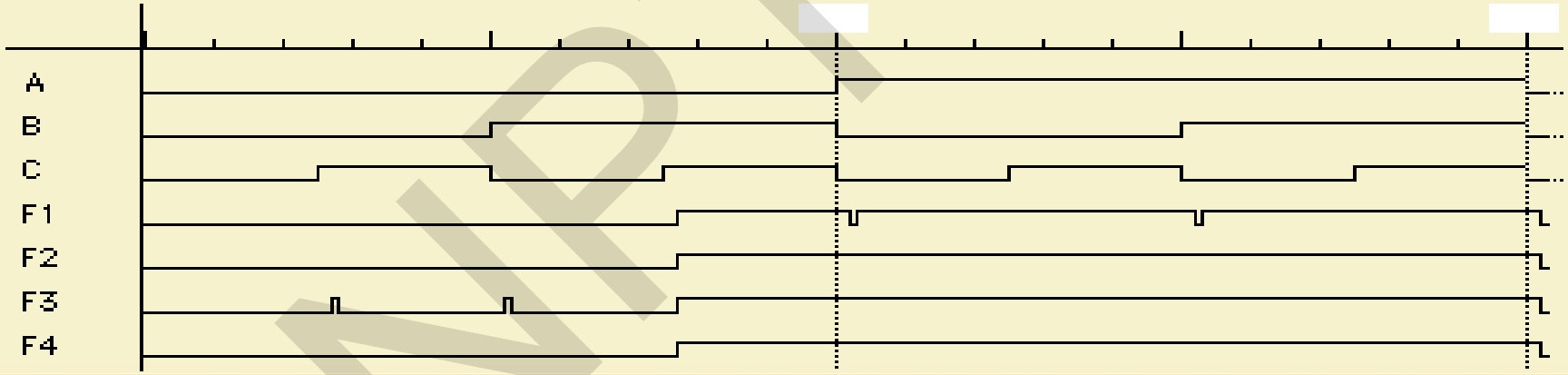
- Delays cause transient F=1

F = A + BC in 2-level logic



Timing diagram for F = A + BC

- Time waveforms for F_1 – F_4 are identical except for glitches



Hazards and glitches

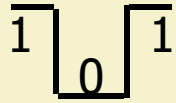
- **glitch**: unwanted output
- A circuit with the potential for a glitch has a **hazard**.
- Glitches occur when different pathways have different delays
  - Causes circuit noise
  - Dangerous if logic makes a decision while output is unstable

Hazards and glitches

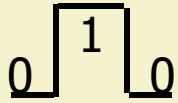
- Solutions
  - Design hazard-free circuits
    - Difficult when logic is multilevel
  - Wait until signals are stable

Types of hazards

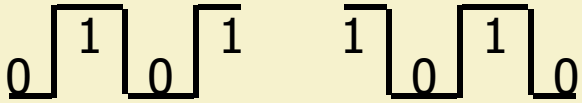
- Static 1-hazard
  - Output should stay logic 1
  - Gate delays cause brief glitch to logic 0



- Static 0-hazard
  - Output should stay logic 0
  - Gate delays cause brief glitch to logic 1



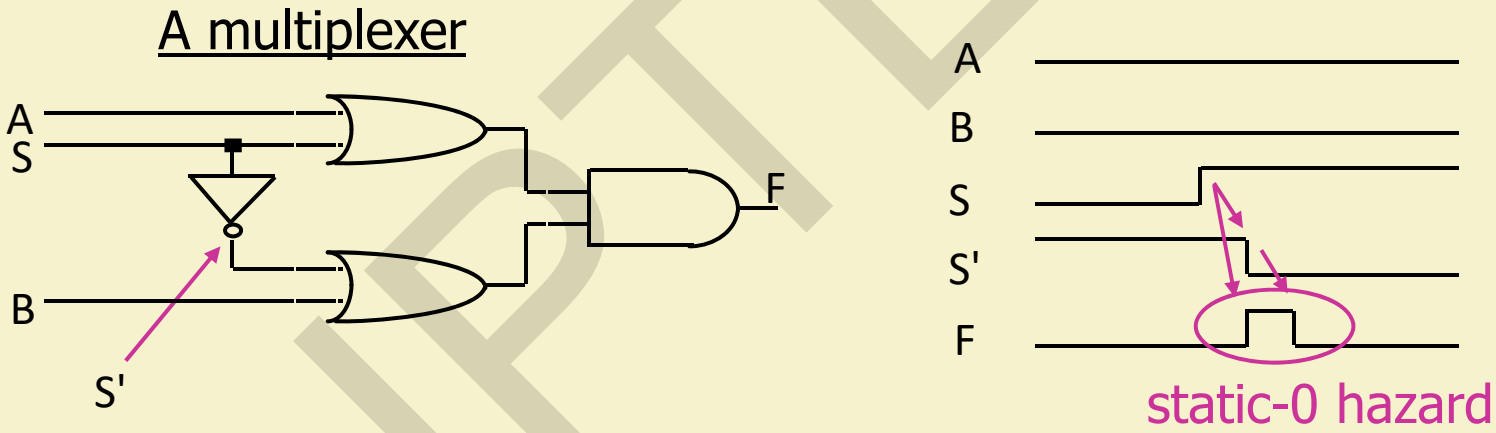
- Dynamic hazards
  - Output should toggle cleanly
  - Gate delays cause multiple transitions



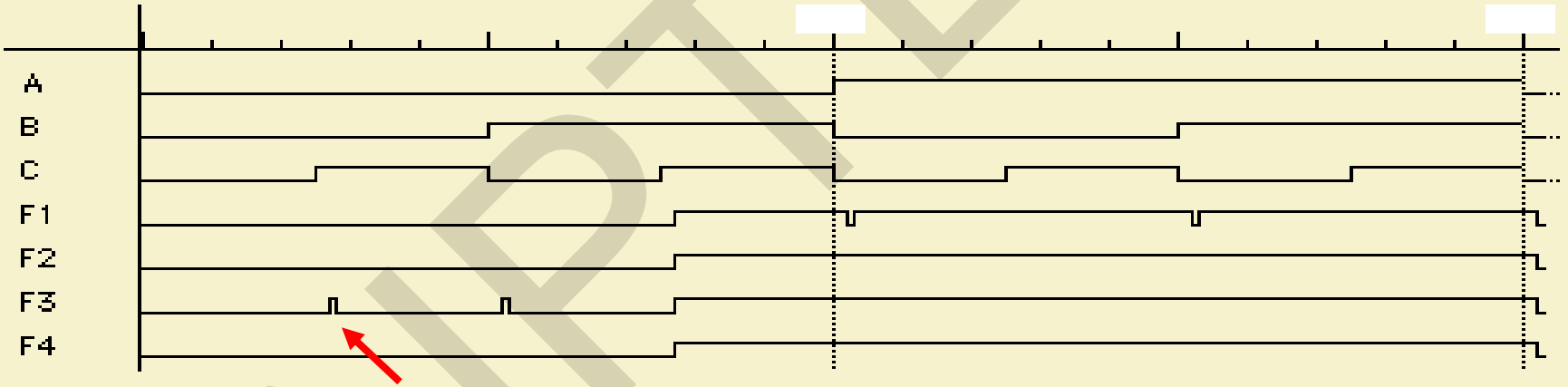
Static hazards

- Often occurs when a literal and its complement momentarily assume the same value
  - Through different paths with different delays
  - Causes an (ideally) static output to *glitch*

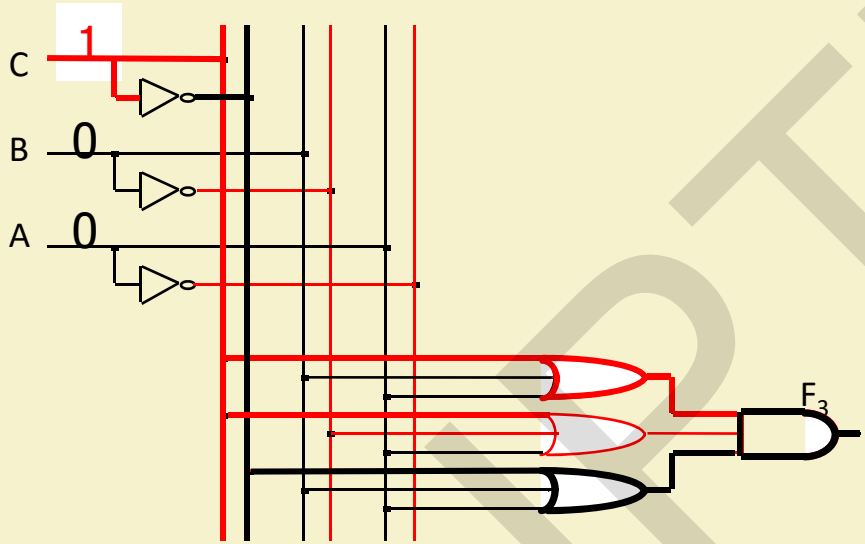
Static hazards



Timing diagram for F = A + BC

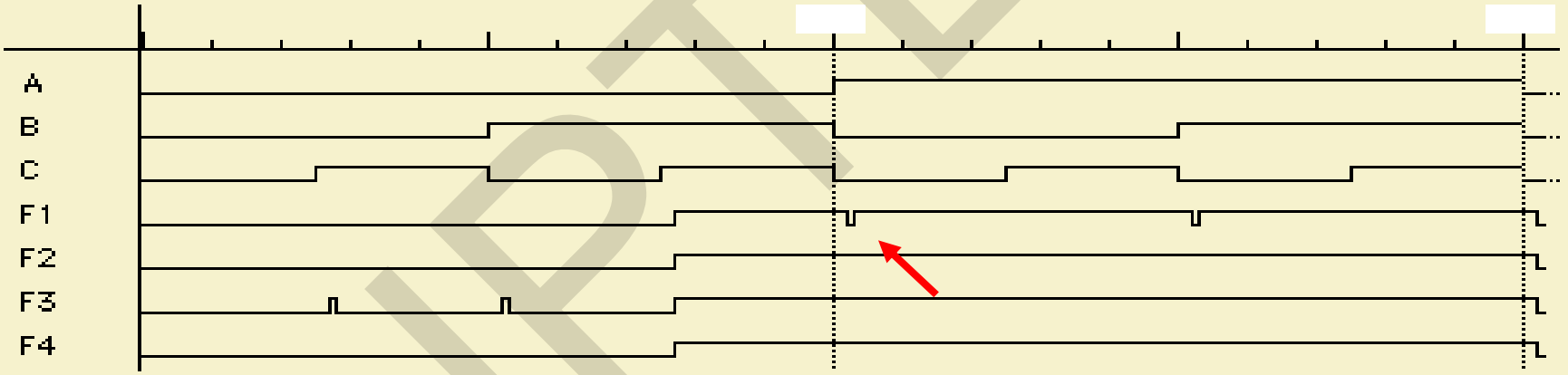


F = A + BC in 2-level logic

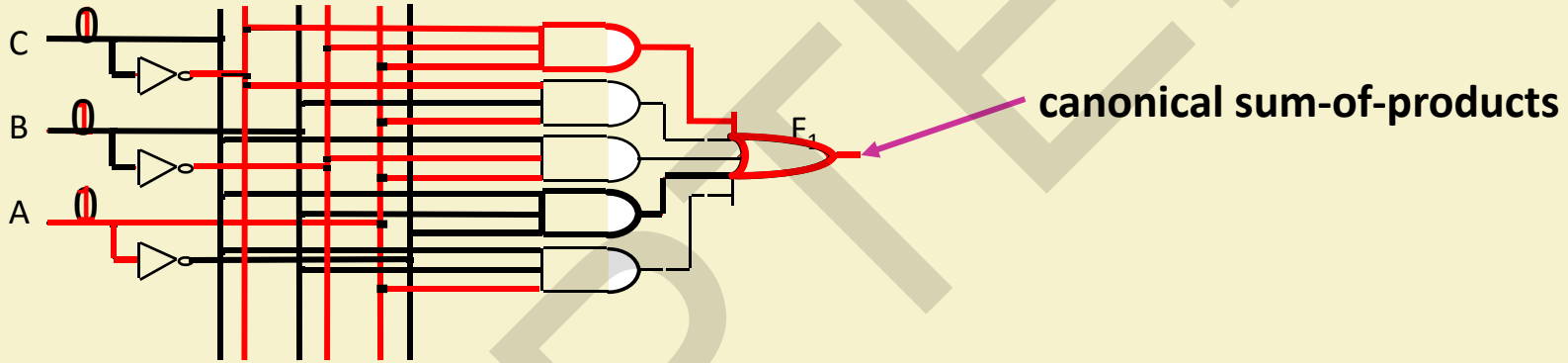


canonical product-of-sums

Timing diagram for F = A + BC

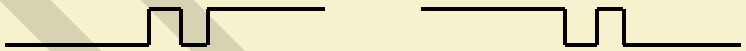


F = A + BC in 2-level logic



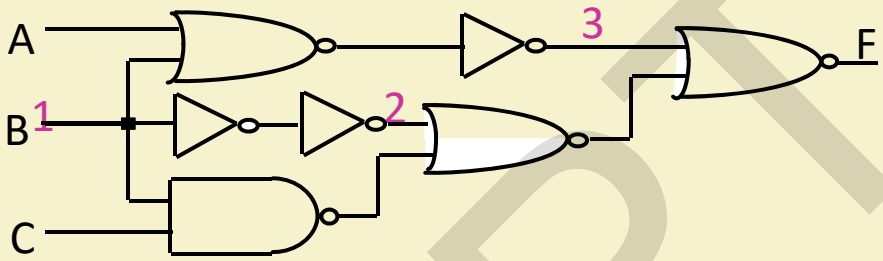
Dynamic hazards

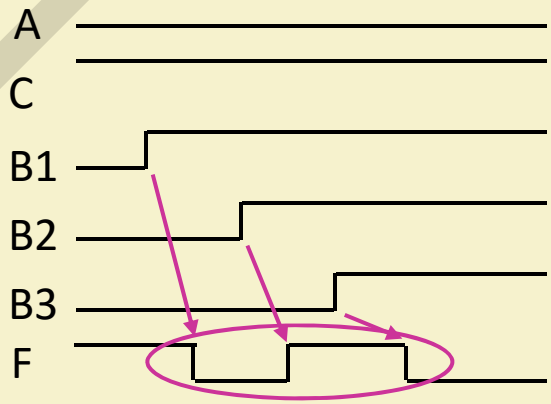
- Often occurs when a literal assumes multiple values
  - Through different paths with different delays
  - Causes an output to toggle multiple times



Dynamic hazards

Dynamic hazards

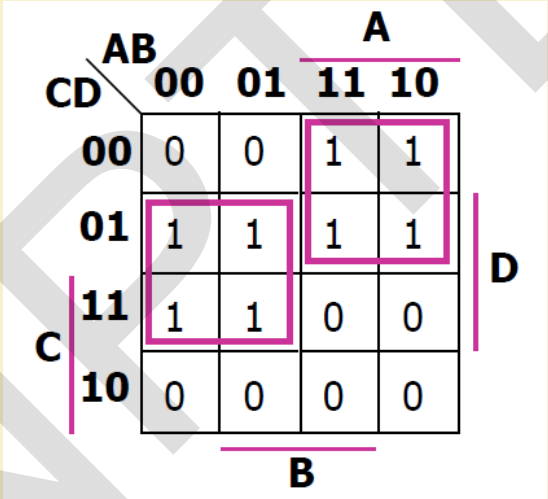




Dynamic hazard

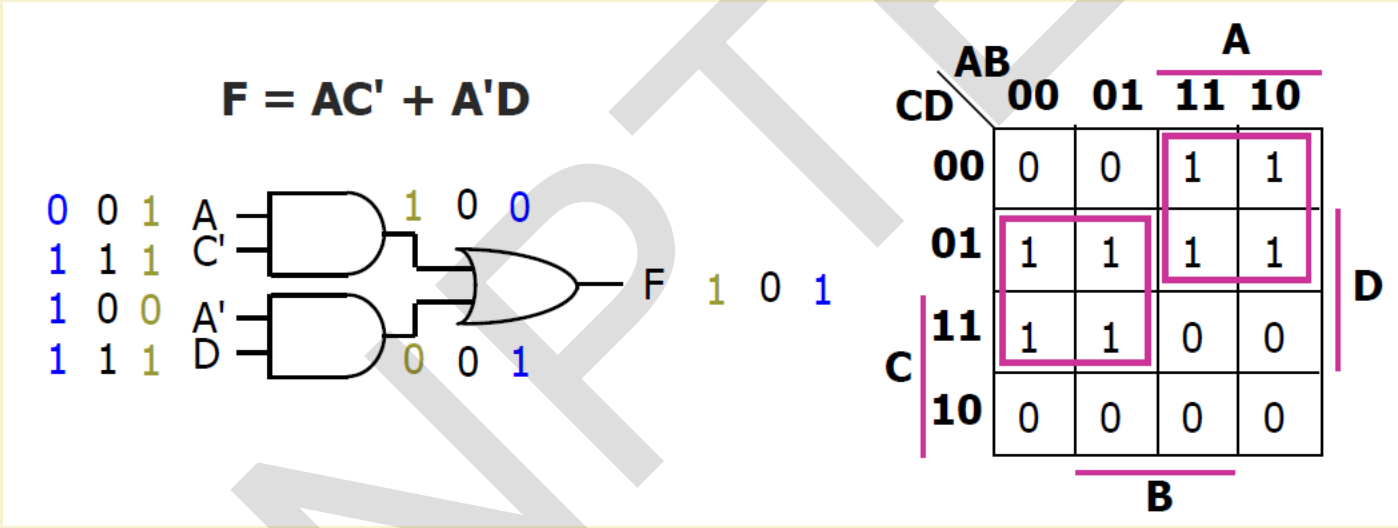
Eliminating static hazards

- Key idea: Glitches happen when a changing input spans separate K-map encirclements
  - Example: 1101 to 0101 change can cause a static-1 glitch



Eliminating static hazards

- ABCD: 1101 → 0101

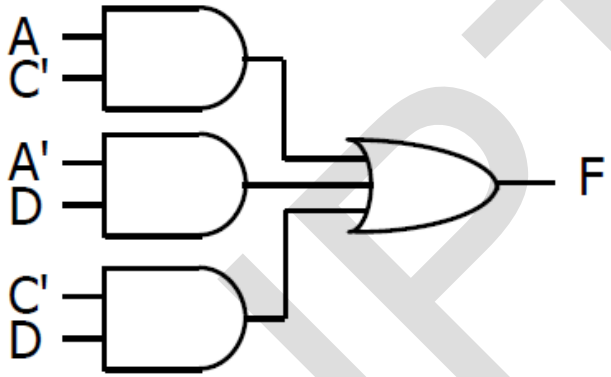


Eliminating static hazards

- Solution: Add redundant K-map encirclements
  - Ensure that all single-bit changes are covered by same block
  - First eliminate static-1 hazards: Use SOP form
  - If need to eliminate static-0 hazards, use POS form
- Technique only works for 2-level logic

Eliminating static hazards

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



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

A, B, C, D

Summary of hazards

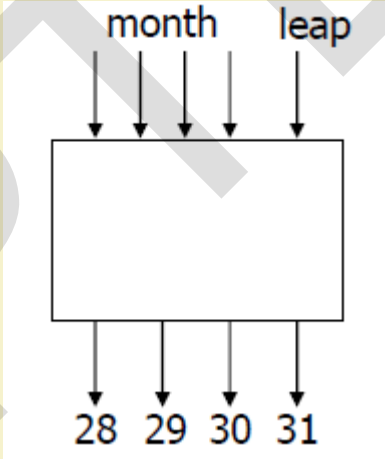
- We can eliminate static hazards in 2-level logic for **single-bit changes**
  - Eliminating static hazards also eliminates dynamic hazards
- Hazards are a difficult problem
  - Multiple-bit changes in 2-level logic are hard
  - Static hazards in multilevel logic are harder
  - Dynamic hazards in multilevel logic are harder yet

Example: Calendar System

- Determine number of days in a month (to control watch display)
  - Used in controlling the display of a wrist-watch LCD screen
  - Inputs: month, leap year flag
  - Outputs: number of days

Formalize the Problem

- Encoding:
  - Binary number for month: 4 bits
  - 4 wires for 28, 29, 30, and 31 one-hot – only one true at any time
- Block diagram



month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

Implementation

28 = m8′ m4′ m2 m1′ leap′

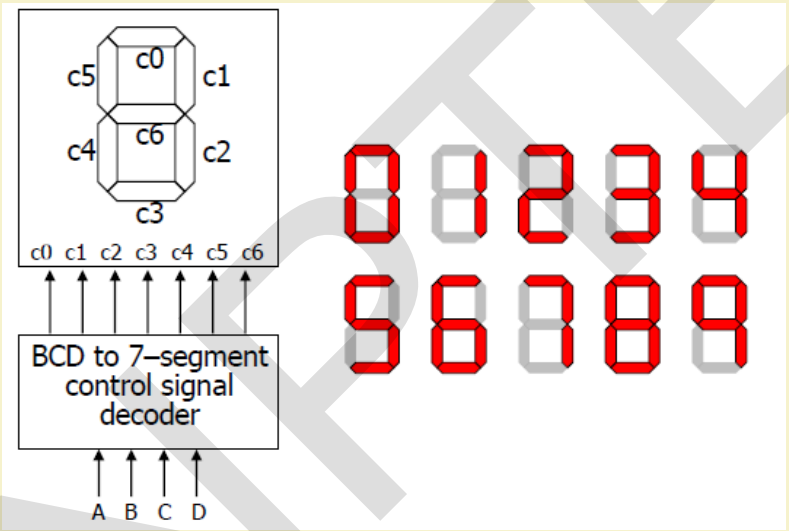
29 = m8′ m4′ m2 m1′ leap

30 = m8′ m4 m1′ + m8 m1

31 = m8′ m1 + m8 m1′

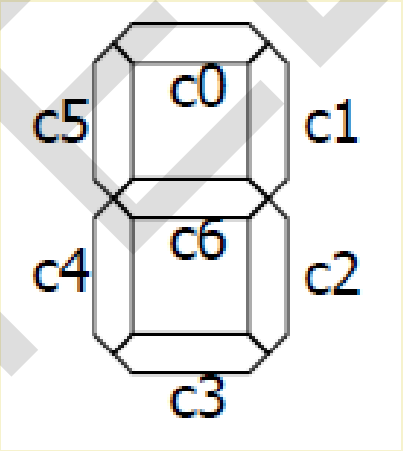
BCD to 7-Segment Display Controller

- Input: 4-bit BCD digit (A, B, C, D)
- Output: 7 control signals for the display (C0 to C6)



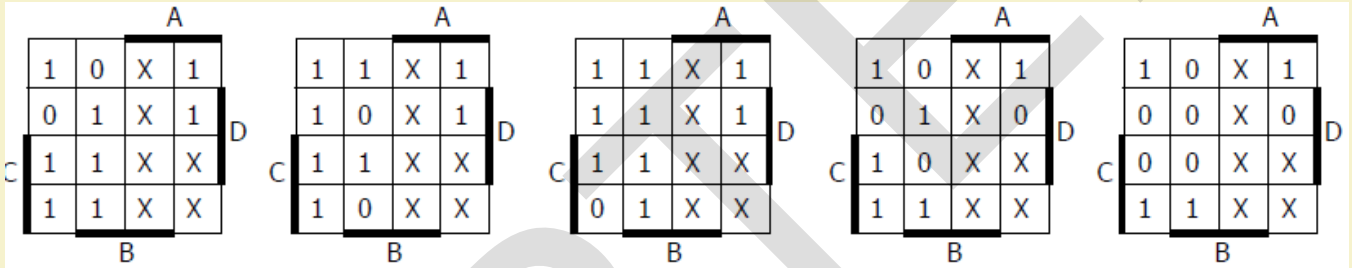
Truth Table

A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	–	–	–	–	–	–	–	–
1	1	–	–	–	–	–	–	–	–	–



Implementation

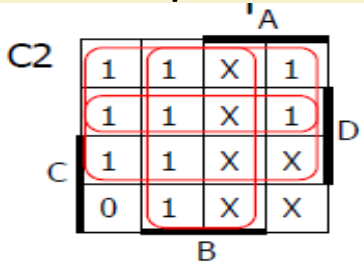
- 15 unique product terms when minimized individually



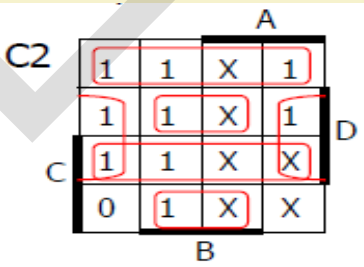
C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

Better Implementation

- 9 unique product terms
- Share terms among outputs
- Each output not necessarily in minimized form



C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C



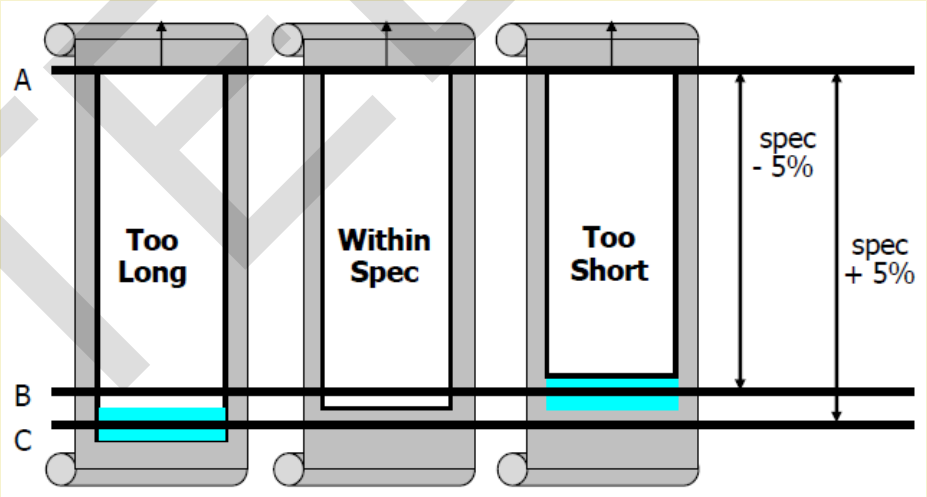
C0 = B C' D + C D + B' D' + B C D' + A
C1 = B' D + C' D' + C D + B' D'
C2 = B' D + B C' D + C' D' + C D + B C D'
C3 = B C' D + B' D + B' D' + B C D'
C4 = B' D' + B C D'
C5 = B C' D + C' D' + A + B C D'
C6 = B' C + B C' + B C D' + A

Production Line Control

- Rods of varying length (+/-10%) travel on conveyor belt
  - Mechanical arm pushes rods within spec (+/-5%) to one side
  - Second arm pushes rods too long to other side
  - Rods that are too short stay on belt
  - 3 light barriers (light source + photocell) as sensors
  - Design combinational logic to activate the arms
- Understanding the problem
  - Inputs are three sensors
  - Outputs are two arm control signals
  - Assume sensor reads "1" when tripped, "0" otherwise
  - Call sensors A, B, C

Sketch of the problem

- Position of Sensors
  - A to B distance = specification – 5%
  - A to C distance = specification + 5%



Problem formulation

- Truth Table
  - Show don't cares

A	B	C	Function
0	0	0	do nothing
0	0	1	do nothing
0	1	0	do nothing
0	1	1	do nothing
1	0	0	too short
1	0	1	don't care
1	1	0	in spec
1	1	1	too long

logic implementation now straightforward
just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

"too long" = A B C
(all three sensors tripped)

Logical Function Unit

- Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

C0	C1	C2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

Truth Table

C0	C1	C2	A	B	F
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

Arithmetic Circuits

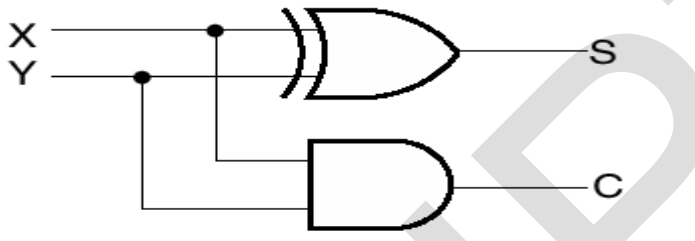
Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

Adder

- Great example of Iterative design
- Design a 1-bit adder circuit, then expand to n-bit adder
- Look at
  - Half adder – which is a 2-bit adder
  - Inputs are bits to be added
    - Outputs: result and possible carry
  - Full adder – includes carry in, really a 3-bit adder

Half Adder

- $S = X \oplus Y$
- $C = XY$



Logic Diagram of Half Adder

Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

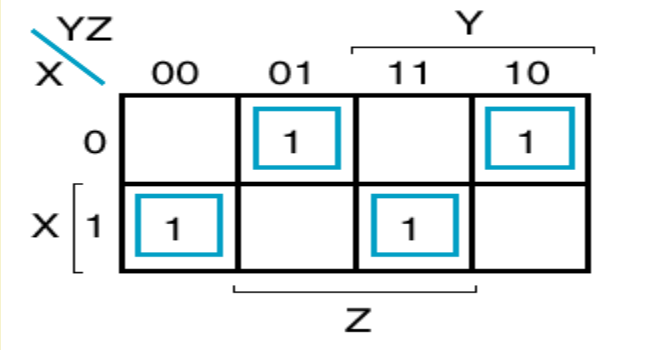
Full Adder

- Three inputs. Two are operand bits, third is C_{in}
- Two outputs: sum and carry

Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

K Map for S



- What is this?

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$

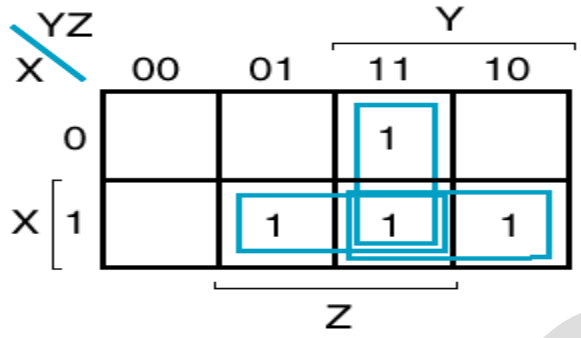
In a half adder the sum bit:

$S = X \oplus Y$

Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

K Map for C



$$C = XY + XZ + YZ$$
$$= XY + Z(X\overline{Y} + \overline{X}Y)$$
$$= XY + Z(X \oplus Y)$$

In a half adder the carry bit:

$C = XY$

Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Using two half Adders to Build a Full Adder

- Full adder functions

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$

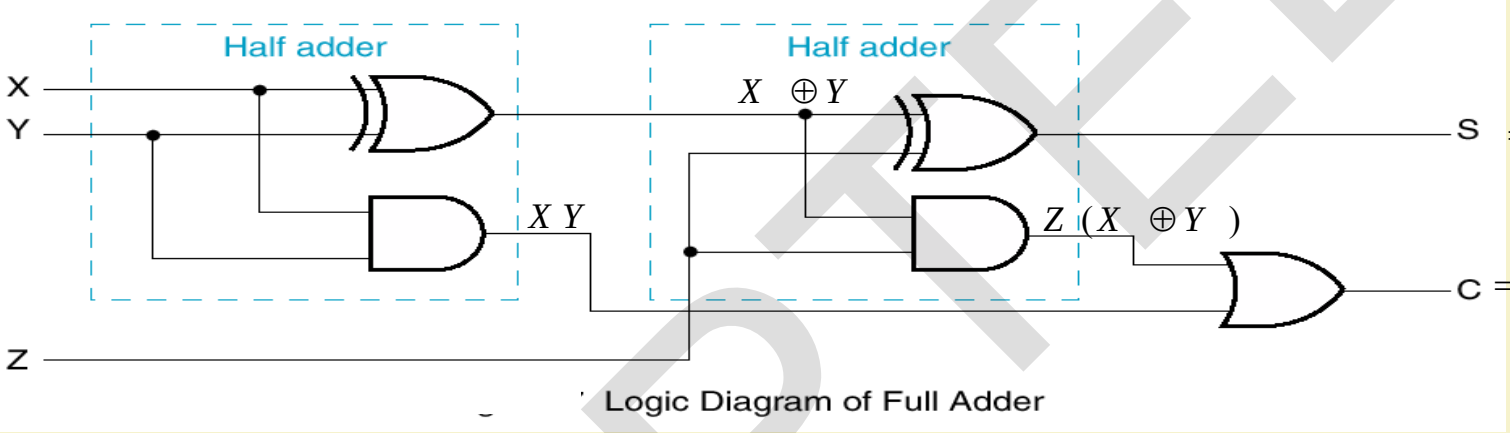
$$C = XY + XZ + YZ$$
$$= XY + Z(X\bar{Y} + \bar{X}Y)$$
$$= XY + Z(X \oplus Y)$$

- Half adder functions

$S = X \oplus Y$

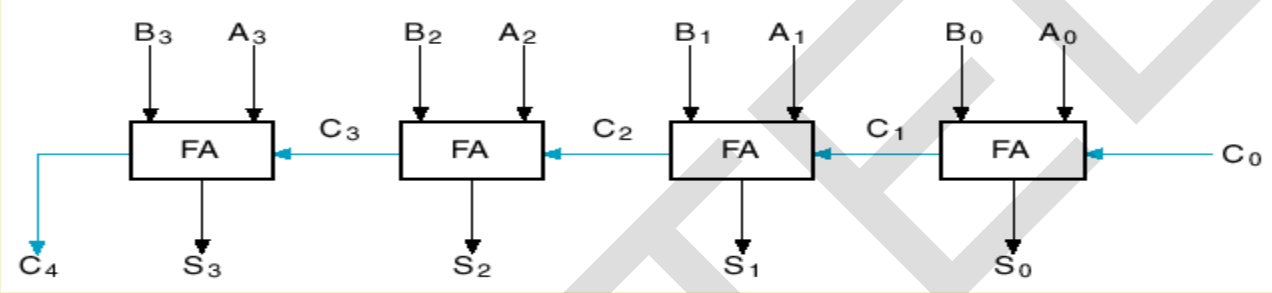
$C = XY$

Two Half Adders (and an OR)



Logic Diagram of Full Adder

Ripple-Carry Adder



- Straightforward – connect full adders
- C_4 : Chain carry-out to carry-in of FA of bits A_4 & B_4
  - C_0 in case this is part of larger chain
  - otherwise just set to zero

Hierarchical 4-Bit Adder

- We can easily use hierarchy here
- Design half adder
- Use in full adder
- Use full adder in 4-bit adder

Subtractor

- Compute M-N
  - Add 2's complement of N to M:

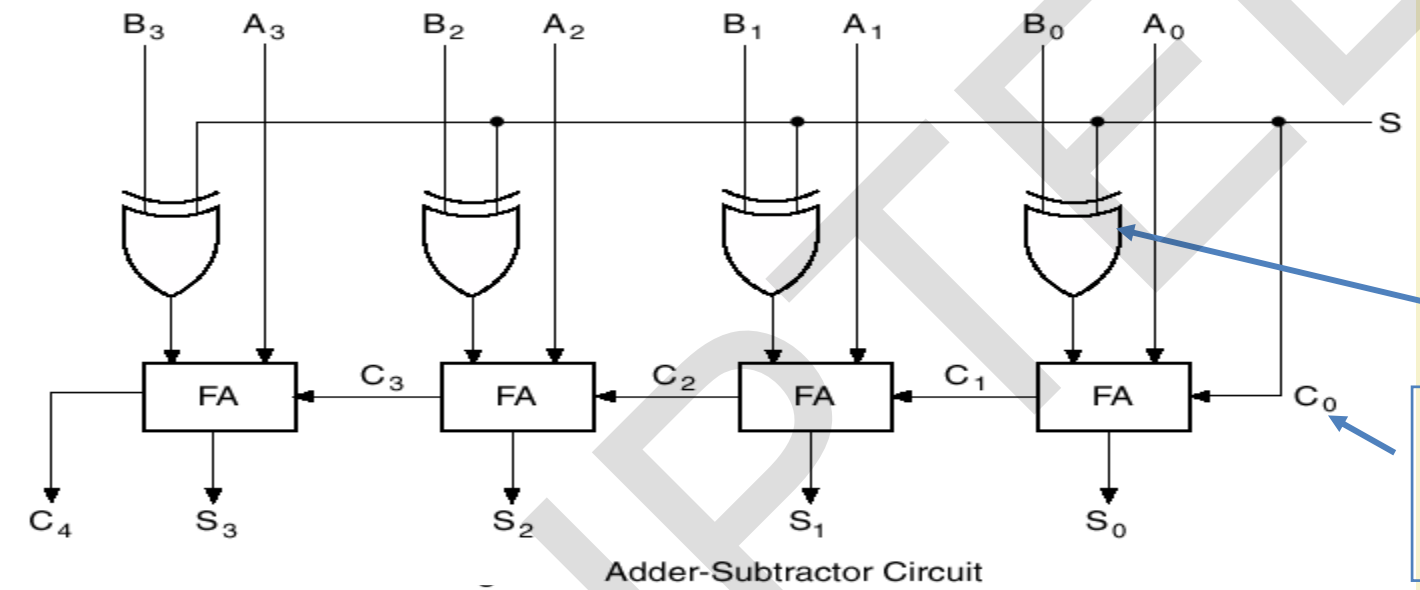
    $M + (2^n - N)$

    - If M >= N, need only "one adder" and a "complementer of N" to do the subtraction.

    - If M < N, we also need to take 2's complement of adder's output to produce magnitude of result

      $2^n - \{ M + (2^n - N)\} = N-M$

Adder-Subtractor Design: (A+B) OR (A-B)



Adder-Subtractor Circuit

S is low for add, high for subtract

Inverts each bit of B if S is 1

Add 1 to make 2's complement

- Output is result if A >= B; Discard carry
- Output is 2's complement of result if B > A

Overflow

- Overflow means that result cannot be represented with the number of bits used
- Two cases of overflow for addition of signed numbers
  - Two large positive numbers overflow into sign bit
    - Not enough bits for result
  - Two large negative numbers added
    - Same – Not enough bits for result

- Consider 7 + 7
- Overflow: cannot represent 14 using only 4 bits
- Generates NO CARRY, $C_4 = 0$

Example 2

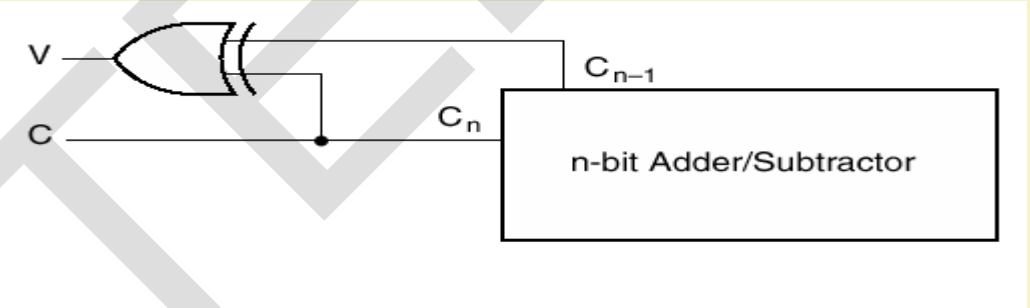
- Consider -7 – 7
- Overflow: cannot represent -14 using only 4 bits
- Generates CARRY, $C_4 = 1$

- Consider 4 + 4
- Overflow: cannot represent 8 using only 4 bits
- Generates NO CARRY, $C_4 = 0$

- Consider 7 – 7
- NO Overflow
- Generates CARRY, $C_4 = 1$

Overflow Detection

- Condition for overflow:
  - either C_{n-1} or C_n is high, but not both



  - 7 + 7; only C_{n-1} (C_3) is high ; overflow
  - -7 - 7; only C_n (C_4) is high ; overflow
  - 4 + 4; only C_{n-1} (C_3) is high ; overflow
  - 7 – 7; BOTH C_{n-1} & C_n (C_4 & C_3) are high; no overflow

Carry-Lookahead Adders

Possible solutions to the carry propagate problem

1. Detect the end of propagation rather than wait for the worst-case time
2. **Speed-up propagation via**
   - **look-ahead**
   - **carry select, etc**.
3. Limit carry propagation to within a small number of bits
4. Eliminate carry propagation through the redundant number representation

Basic Signals

Generate signal: $g_i = x_i y_i$

Propagate signal: $p_i = x_i \oplus y_i$

Anihilate (absorb) signal: $a_i = \overline{x_i}\,\overline{y_i} = \overline{x_i + y_i}$

Transfer signal: $t_i = g_i + p_i = \overline{a_i} = x_i + y_i$

$c_{out} = 1$ given $c_{in} = 1$

Carry recurrence

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

Unrolling Carry Recurrence

$$
\begin{aligned}
c_i &= g_{i-1} + c_{i-1}p_{i-1} = \\
&= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} = g_{i-1} + g_{i-2}\,p_{i-1} + c_{i-2}p_{i-2}p_{i-1} = \\
&= g_{i-1} + g_{i-2}\,p_{i-1} + (g_{i-3} + c_{i-3}p_{i-3})p_{i-2}p_{i-1} = \\
&= g_{i-1} + g_{i-2}\,p_{i-1} + g_{i-3}\,p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} = \\
&= \ldots\ldots = \\
&= g_{i-1} + g_{i-2}\,p_{i-1} + g_{i-3}\,p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + \ldots\ldots + \\
&\quad + g_0p_1p_2\cdots p_{i-2}p_{i-1} + c_0p_0p_1p_2\cdots p_{i-2}p_{i-1} = \\
&= \boxed{g_{i-1} + \sum_{k=0}^{i-2} g_k \prod_{j=k+1}^{i-1} p_j + c_0 \prod_{j=0}^{i-1} p_j}
\end{aligned}
$$

4-bit Carry-Lookahead Adder

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_1 = g_0 + c_0 p_0$$

$$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$$

$$s_1 = p_1 \oplus c_1$$

$$s_2 = p_2 \oplus c_2$$

$$s_3 = p_3 \oplus c_3$$

4-bit Carry-Lookahead Adder (2)

$c_4 = g_3 + \mathbf{c_3} p_3$

3 gates less

$\mathbf{c_3} = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$

$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$

$c_1 = g_0 + c_0 p_0$

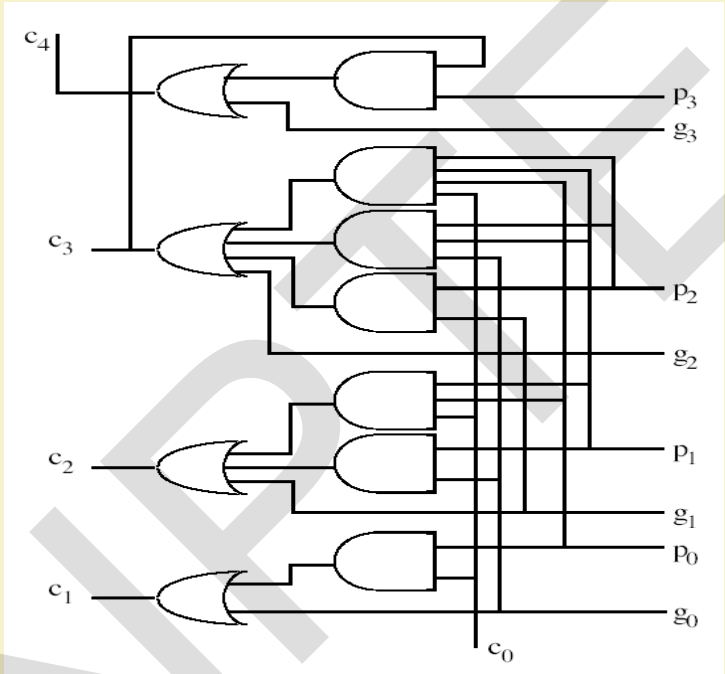
$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$

$s_1 = p_1 \oplus c_1$

$s_2 = p_2 \oplus c_2$

$s_3 = p_3 \oplus \mathbf{c_3}$

4-bit Carry Network with Full Lookahead



Equations for 4-bit Lookahead Carry Generation

$$c_{i+3} = g_{i+2} + g_{i+1}\, p_{i+2} + g_i\, p_{i+1} p_{i+2} + c_i p_i p_{i+1} p_{i+2}$$

$$c_{i+2} = g_{i+1} + g_i\, p_{i+1} + c_i p_i p_{i+1}$$

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

$$g_{[i..i+3]} = g_{i+3} + g_{i+2}\, p_{i+3} + g_{i+1}\, p_{i+2}\, p_{i+3} + g_i\, p_{i+1}\, p_{i+2}\, p_{i+3}$$

$$p_{[i..i+3]} = p_i\, p_{i+1}\, p_{i+2}\, p_{i+3}$$

4-bit Lookahead Carry Generator Schematic